

TCP/IP and the Linux Protocol Implementation
(Networking Council Series)

©Jon Crowcroft, Iain Phillips,
Wiley

July 23, 2001

Preface

This book is about the implementation of the Internet Protocol suite, in the Linux operating system.

Linux provides all the components necessary to build clients, servers and full-featured routers, and so it is possible to conceive of a world where we built almost all of the Internet out of Linux systems.

Of course, there is strength in diversity, and so luckily for the Internet software gene-pool, we have alternatives which keep each other honest. Thus there are many operating systems and hardware platforms on which the Internet is built in practice. What makes Linux particularly interesting, and what is relevant for readers of this book, is that the source code of the operating systems and many of the applications is freely available, and that the software runs on many hardware platforms ranging from the embedded microprocessor, to the multiprocessor and huge cluster systems that can function as core network routers, or major site servers.

Importance

There are a number of Fortune 100 companies now using Linux for servers, not just in the technology sector, but also in banking, healthcare and just about any other business you can think of. Some of these have started to use Linux as a router platform too. Given the open nature of the system, and its level of sophistication, both in terms of functionality and performance, this is entirely reasonable. We believe that this trend will increase over the next few years as the commoditization of the Internet rolls on.

Intended Audience

The material in this book is systems code. This means that it is tethered to hardware and other environmental specifics. While the Linux kernel is portable, it is not *generic*: It has a specific goal and purpose. The assumption we have made in writing this book is that a reader will be familiar with the general ideas of operating systems and communications protocols.

This is not a book for someone wishing to write an Internet application. There are several excellent books for that purpose. Indeed, most of the major operating systems share an API (“sockets”) for that purpose. The emphasis here is on the system specifics rather than the communications applications.

In fact we deliberately avoid most aspects of so-called *user land* or *user space* programming, and concentrate on the kernel internals.

As network researchers and educators, we have increasingly found it useful to explain (or create) ideas in communications by examples. Thus the goal of this book is to use the Linux kernel implementation of the Internet protocol suite as an exemplar. If you are learning or researching Internet Protocol protocols and services, transport protocols and services, or adapting either of these areas to novel communications media, then this is a book for you.

The level of the material assumes an engineering level knowledge of communication, operating and hardware system concepts. We assume you're acquainted with these already. If not, then there are some excellent books such as the classic work on hardware is the Patterson and Hennessy work[24]. There are many on communications (e.g. Kurose and Ross[25]) and Operating Systems (e.g. Silberschatz and Peterson[23]).

Unintended Audience

This book is not about system administration. For an excellent book on sys-admin for Linux routers, see the book by Tony Mancill[27]. We have also made a (somewhat arbitrary) decision to exclude non Internet mainstream protocols from the book (e.g. IPX, Appletalk, Decnet, ax25, X.25, ECNET, IRDA, ATM, are all not covered here).

Another criterion used to decide what went in and what didn't was whether the protocol required kernel support. Protocols that are entirely user-land based are really portable across a range of platforms, and so are not especially part of the Linux world exclusively. In general, this means that we exclude Internet applications altogether, except where they are close to the low level (e.g where the application's *raison d'être* is solely to manage a low level function).

Finally, we worked largely to the 2.4.0 kernel release from early 2001. Things that were in earlier releases, but are not here, or are in later releases, but weren't present here, are obviously not covered (volume 2, maybe!). We found our way around the code with help (of course) from the Linux Documentation Project, and from a variety of tools (ctags and vi, etags and emacs, the Red Hat source code navigator, grep, ls -R, find, etc etc - all those things that make Unix like systems the tool of choice for systems developers).

Roadmap, Outline and Organisation

The book is in ten chapters. Chapter one is a very brief reminder of the way that the Internet Protocol suite works. Chapter two is an introduction to the Linux Operating System kernel. Chapter three is a walk through of the life cycle of a single packet, from creation to transmission, and from reception to consumption. Chapter four is a recapitulation of Inter-Process Communication; in other words, an overview of socket programming. Chapter five is about the kernel infrastructure support for communications in general. Chapter six is a walkthrough of the implementation of the network layer code and support. Chapter seven is about transport protocol implementations. Chapter eight covers routing, while chapter nine is about the rich Linux support for different

packet forwarding treatment. We wrap up with chapter ten, which is about network security, covering various network filtering techniques and applications.

The approach taken is largely code-centric, but with explanations and illustrations of the way the code fits together and is used by other layers.

Each chapter covers the concepts and implementation from one or more of five perspectives:

Data structures We show by example the way that protocol state and packet data is stored

Wire format of packets We show the format of packets as transmitted/forwarded/received.

Algorithm/code The rules for processing user, network and timer events are detailed

Process The progression of a the set of tasks involved in communication are illustrated.

Operational We cover the set of actions carried out by the system manager, network ordinary user.

The structure of most chapter covers these five perspectives, and then connects them together through a story - the life of some specific, or general component activity of the Internet protocol architecture as exemplified by the Linux implementation.

Acknowledgements

This book is dedicated to the followers of *li*.

Thanks also to the editors and reviewers, as well as to our colleagues Orion Hodson, Tristan Henderson, Panos Gevros, and other denizens of (virtual) G11 and to Martin Hamilton and Adam Greenhalgh, Julia Schnabul, and John Andrews and Tony Mancill for feedback. Thanks specially to Carsten Heinz, for his excellent listings style and assistance fixing my overly naive use of said. Thanks to Jamal Hadi Salim for information about SMP forwarding, and to T'so for helpful comments on security.

Of course, we would have nothing if not for the valiant efforts of the authors of the code in implementing and bugfixing and performance tweaking. A list of implementors who signed code that we look at is appended.

Contents

1	Overview of the Internet Protocols	2
1.1	Roadmap	2
1.2	Introduction	2
1.3	Protocols	5
1.3.1	A Note on Programming Languages and Methodologies	5
1.4	Protocol Stacks	8
1.4.1	The Basics	9
1.4.2	IP	9
1.4.3	Routes	9
1.4.4	TCP	10
1.4.5	Packet exchange patterns	11
1.4.6	Performance trends	13
1.5	Security	13
1.6	Performance	14
1.6.1	Integrated Services	14
1.6.2	Differentiated Services	15
1.7	Summary	16
2	Introduction to the Linux Operating System	17
2.1	Roadmap	17
2.2	What is an Operating System?	18
2.2.1	Multi-user	19
2.2.2	Multi-tasking	20
2.2.3	Multi-device	22
2.2.4	From input/output to Interprocess Communication	24
2.2.5	Processor Independence	25
2.2.6	Concurrency Control	26
2.2.7	Memory Management	28
2.3	What more could you ask for?	29
2.4	The Source Code Organisation	31
2.5	A day in the life of a Process	32
2.6	A day in the life of a File	33
2.7	A day in the life of an operating system	34
2.7.1	The start of time	34
2.7.2	As time goes by	35
2.7.3	The end of time	35
2.8	A day in the life of a device driver	35
2.9	A day in the life of a kernel module	35

2.10	SMP	36
2.10.1	Application Support	36
2.11	A Day in the Life of Time	37
2.12	Summary	38
3	The Brief Life of a Packet	39
3.1	Roadmap	39
3.1.1	User, Application and Protocol Viewpoints	40
3.1.2	Source Code, Execution and Wire Viewpoints	42
3.1.3	State, Memory etc	43
3.2	TCP Example	43
3.2.1	Socket Level	43
3.2.2	TCP Output	43
3.2.3	IP Output Work	44
3.2.4	Link Level Output	44
3.2.5	Link Level Input	44
3.2.6	IP Input Work	45
3.2.7	TCP Input Work	45
3.2.8	Socket Input Work	45
3.2.9	On the wire	45
3.3	DNS/UDP Example	46
3.3.1	UDP Output	46
3.3.2	UDP Input	46
3.3.3	On the wire	46
3.4	RTP/UDP (multicast) Example	46
3.4.1	On the wire	47
3.4.2	...but with a router in the path...	47
3.5	Three Views of the Traces of Ping	47
4	Inter-Process Communication	50
4.1	Roadmap	50
4.2	Socket API	51
4.3	The actual socket API	52
4.4	Using the Socket API	53
4.4.1	Those System Include Files	55
4.5	Summary	55
5	Protocol Implementation Framework	56
5.1	Roadmap	56
5.2	A Day in the Life of a software interrupt	56
5.2.1	Concurrency in the Linux Communications Stack	56
5.2.2	Producer/Consumer Chain	57
5.2.3	What about real Multi-processor Systems?	59
5.3	Bottom Up: Interrupts, DMA etc	59
5.4	Device Level, bottom up, one level up	61
5.4.1	Network Device Driver API	62
5.4.2	Network Reception	63
5.4.3	A specific device driver - 3c501	63
5.4.4	link driver level config	65
5.5	Top Down: Socket Glue Level	66

5.5.1	Virtual File System Socket instance functions	67
5.6	Sideways: A day in the life of a socket buffer	74
5.6.1	Allocating a Socket Buffer	76
5.6.2	Allocating and freeing a socket buffer from the global pool	77
5.6.3	Manipulating Socket Buffers	78
5.7	A day in the life of a Protocol Control Block	79
5.7.1	Socket and Route related transport state	79
5.7.2	Future of the socket structure	80
5.7.3	Reference COunting	81
5.7.4	Sockets and Routes	82
5.7.5	Socket Credentials Manager Interface	83
5.7.6	A brief with comparison BSD Style Stack Infrastructure	84
5.8	Summary	85
6	Infrastructure Protocols	86
6.1	Roadmap	86
6.2	IP	86
6.2.1	Internet Protocol Output	87
6.2.2	Fragmentation	88
6.2.3	Internet Protocol Input	92
6.2.4	Internet Protocol Forwarding	93
6.2.5	Internet Protocol Fragmentation Support Functions	94
6.2.6	Internet Protocol Options	97
6.2.7	Internet Protocol Socket Glue	99
6.3	Addressing	101
6.4	Network Level Debugging - ICMP	107
6.5	Group Management - IGMP	129
6.5.1	IGMP	129
6.5.2	Mangeing the group/interface state	133
6.5.3	Multicast Forwarding	138
6.5.4	Multicast Transmission	145
6.5.5	PIM Specifics	146
6.5.6	Starting Multicast	147
6.6	NATs, Tunnels and Other Hacks	147
6.7	Network Address Translation	148
6.8	Tunnels	149
6.8.1	IP in IP Tunnels	149
6.8.2	GRE Tunnels	154
6.9	IP version 6	160
6.9.1	IPv6 Header Definitions	161
6.9.2	IPv6 API	162
6.9.3	IPv6 Address (Auto)-Configuration	163
6.9.4	IPv6 Neighbour Discovery	166
6.9.5	IPv6 Input	168
6.9.6	IPv6 Output	169
6.9.7	IPv6 Forward Function	170
6.9.8	IPv6 Socket Glue	171
6.9.9	IPv6 Flow Label	172
6.9.10	Extension Headers	173
6.9.11	Simple Internet Transition	174

6.9.12	Effect on transport of IPv6 of note	174
6.10	Summary	174
7	Transport	175
7.1	Roadmap	175
7.2	Introduction	175
7.3	UDP—User Datagram Protocol	176
7.3.1	Introduction	176
7.3.2	Multiplexing	176
7.3.3	UDP packet header anatomy	176
7.3.4	Transport Layer linkage	177
7.3.5	The <code>sock</code> structure	179
7.3.6	DNS look-up	179
7.4	TCP—Transmission Control Protocol	181
7.4.1	TCP Transport Functionality	181
7.4.2	HTTP request	183
7.4.3	TCP header	184
7.4.4	TCP implementation	184
7.4.5	TCP states	185
7.4.6	TCP Structures	185
7.4.7	TCP Code	185
7.4.8	TCP behaviour	192
7.5	Management	193
7.6	TCP vs UDP	193
7.7	RTP—Real Time Protocol	193
7.8	Writing a new Transport Layer	194
7.9	Summary	194
8	Routing	195
8.1	Roadmap	195
8.2	Introduction	195
8.3	Forwarding Table Computation	200
8.3.1	IP Addressing	200
8.3.2	FIB Purpose	203
8.3.3	FIB Implementation	204
8.3.4	FIB Interface	204
8.3.5	FIB Code	205
8.4	Multicast	229
8.4.1	Dense Mode, Data-driven Multicast Routing	230
8.4.2	Sparse and Single Source Multicast	231
8.5	QoS Routing	231
8.6	Routing Table Construction	232
8.6.1	GateD	232
8.6.2	Route Control and Traffic Control Maintenance	233
8.7	Host routing	238
8.8	Managing IP level state information - the <code>IP</code> command	239
8.9	Summary	241

9 Forwarding	242
9.1 Roadmap	242
9.2 Introduction	243
9.2.1 Integrated Services	246
9.2.2 Admission Test for Integrated Services Guaranteed Service	247
9.2.3 Network Architecture	247
9.3 Mixing Traffic onto the wire	248
9.3.1 Performance Parameters	249
9.3.2 Probing or measurement based admission control?	251
9.4 Elastic and Inelastic Applications - tolerance for variation?	251
9.4.1 Network Source behaviour Characterization: Leaky and Token Buckets	253
9.5 Packet Forward Scheduling and Queue Management	253
9.6 Queueing Disciplines	253
9.6.1 Common Scheduling Disciplines	254
9.6.2 Class Based Queueing	256
9.7 Queue Management	256
9.7.1 Random Early Detection	257
9.8 Linux Forwarding Treatment Architecture	258
9.8.1 Output Interface	258
9.8.2 Queue Discipline	259
9.8.3 Class/Type	259
9.8.4 filter, or classifier	260
9.9 The Classifier Code	260
9.9.1 Classifiers	261
9.10 queueing discipline	273
9.11 Scheduler Framework	274
9.11.1 Scheduling API	274
9.11.2 Scheduler Framework	286
9.12 Class Based Queueing	297
9.13 Queue Management	318
9.13.1 Best Effort (a.k.a FIFO) Scheduler	318
9.13.2 Ingress Queueing	323
9.13.3 Differentiated Services Marker	326
9.13.4 RED and Friends	327
9.13.5 Explicit Congestion Notification	328
9.13.6 GRED	330
9.13.7 Token Bucket Queue	331
9.13.8 Ingress Queueing	332
9.13.9 Priority Queueing	334
9.13.10 Stochastic Fairness Queueing	335
9.13.11 Priority Queueing	336
9.13.12 Stochastic Fairness Queueing	337
9.13.13 Link Equalization	338
9.13.14 Profiling	339
9.14 Managing Forwarding Treatment Information - the TC Command	344
9.14.1 Traffic Control - Using The Tool	344
9.14.2 Traffic Control - How the Tool is implemented	345
9.14.3 TC Command Usage	345
9.15 BSD	348

9.16 Summary	349
10 Security	350
10.1 Roadmap	350
10.2 Security Scenarios	350
10.2.1 Firewalls	350
10.2.2 NAT—Network Address Translation	350
10.3 Netfilter	352
10.3.1 Introduction	352
10.3.2 IP Tables	353
10.3.3 Summary	353
10.4 IPSec	353
10.5 SSL	353
10.6 Summary	353

List of Figures

1	Comments, what comments: <code>linux/net/khttpd/waitheaders.c</code>	384
2	LAN	385
3	Layers of Abstraction	386
4	The RFC style IP Header Definition	387
5	The C style IP Header Definition	388
6	The RFC style IP Header Definition	389
7	The TCP Header Definition: <code>linux/include/linux/tcp.h</code>	390
8	trends	391
9	Differentiated Services Metering and Marking	392
10	POSIX C Type Definitions: <code>linux/include/linux/posix_types.h</code>	393
11	i386 ASM header interface C Type Definitions: <code>linux/include/asm{i386}/types.h</code>	394
12	Kernel C Type Definitions: <code>linux/include/linux/types.h</code>	395
13	Kernel C Type Definitions: <code>linux/include/linux/types.h</code>	396
14	Kernel C Type Definitions: <code>linux/include/linux/types.h</code>	397
15	System Calls	398
16	The Scheduler i: <code>linux/kernel/sched.c</code>	399
17	The Scheduler ii: <code>linux/kernel/sched.c</code>	400
18	The Scheduler iii: <code>linux/kernel/sched.c</code>	401
19	The Scheduler iv: <code>linux/kernel/sched.c</code>	402
20	The Scheduler iv: <code>linux/kernel/sched.c</code>	403
21	The Scheduler v: <code>linux/kernel/sched.c</code>	404
22	The Scheduler Tail: <code>linux/kernel/sched.c</code>	405
23	The Wakeup Common Code: <code>linux/kernel/sched.c</code>	406
24	The Sleepon Code: <code>linux/kernel/sched.c</code>	407
25	The Linux Kernel System Call Entry Code -save and restore over- head: <code>linux/arch/i386/kernel/entry.S</code>	408
26	The Linux Kernel System Call Entry Code - example of entry: <code>linux/arch/i386/kernel/entry.S</code>	409
27	The Linux Kernel Startup Code i: <code>linux/init/main.c</code>	410
28	The Linux Kernel Startup Code ii: <code>linux/init/main.c</code>	411
29	The Linux Kernel Startup Code iii: <code>linux/init/main.c</code>	412
30	Linux Character Device Registration: <code>linux/fs/devices.c</code>	413
31	Linux Block Device Registration : <code>linux/fs/block_dev.c</code>	414
32	Linux Network Device Registration - i: <code>linux/net/core/dev.c</code>	415
33	Linux Network Device Registration - ii: <code>linux/net/core/dev.c</code>	416
34	Linux Network Device Registration - iii: <code>linux/net/core/dev.c</code>	417
35	Linux Network Device Registration - iv: <code>linux/net/core/dev.c</code>	418
36	Linux Network Device Registration - v: <code>linux/net/core/dev.c</code>	419
37	Linux Network Device Registration - 2: <code>linux/net/core/dev.c</code>	420

38	Linux Network Device Registration - 2: <code>linux/net/core/dev.c</code>	421
39	Linux Network Device Registration - 2: <code>linux/net/core/dev.c</code>	422
40	Monolithic, Micro-kernel or Hybrid Kernel?	423
41	SMP netdevice scheduling: <code>linux/include/linux/netdevice.h</code>	424
42	Linux Timer Structure: <code>linux/include/linux/timer.h</code>	425
43	Updating the Linux Timer Structure: <code>linux/include/linux/timer.h</code>	426
44	Linux Timer Calls - <code>add_timer</code> : <code>linux/kernel/timer.c</code>	427
45	Linux Timer Calls - <code>internal_add_timer</code> : <code>linux/kernel/timer.c</code>	428
46	The overall transmission process: from Application to Wire	429
47	The overall reception process: from Wire to Application	430
48	Filling in Some Key TCP fields: <code>linux/net/ipv4/tcp_output.c</code>	431
49	Filling in Some Key IP bits: <code>linux/net/ipv4/ip_output.c</code>	432
50	Network Device Driver Rx i: <code>linux/net/core/dev.c</code>	433
51	Network Device Driver Rx ii: <code>linux/net/core/dev.c</code>	434
52	Network Device Driver Rx i: <code>linux/net/core/dev.c</code>	435
53	Network Device Driver Rx ii: <code>linux/net/core/dev.c</code>	436
54	Socket Receive wake up process: <code>linux/net/core/sock.c</code>	437
55	UDP Output i: <code>linux/net/ipv4/udp.c</code>	438
56	UDP Output ii: <code>linux/net/ipv4/udp.c</code>	439
57	UDP Output iii: <code>linux/net/ipv4/udp.c</code>	440
58	UDP Input i: <code>linux/net/ipv4/udp.c</code>	441
59	UDP Input iii: <code>linux/net/ipv4/udp.c</code>	442
60	UDP Receive Multicast: <code>linux/net/ipv4/udp.c</code>	443
61	Use of ping	444
62	Wire view of ping	445
63	Strace of ping	446
64	<code>socket()</code>	447
65	Socket Programming 1 - main: <code>pix/sock.c</code>	448
66	Socket Programming 2 - client: <code>pix/sock.c</code>	449
67	Socket Programming 3 - server: <code>pix/sock.c</code>	450
68	Select: <code>pix/poll.c</code>	451
69	<code>poll()</code>	452
70	<code>pollfd</code>	453
71	NBIO: <code>pix/nbio.c</code>	454
72	Asynchronous Reception	455
73	SIGIO: <code>pix/asynch.c</code>	456
74	IP user view of address structure: <code>linux/include/linux/in.h</code>	457
75	IP user view of the IPv6 address structure: <code>linux/include/linux/in6.h</code>	458
76	Producer Consumer Chain	459
77	Deadlock	460
78	Livelock	461
79	Task Queue structure- copyright statement: <code>linux/include/linux/tqueue.h</code>	462
80	Task Queue (TQ) structures: <code>linux/include/linux/tqueue.h</code>	463
81	TQ mapping to deferred processing/bh_: <code>linux/include/linux/tqueue.h</code>	464
82	TQ inline functions, queue, and run: <code>linux/include/linux/tqueue.h</code>	465
83	Soft IRQs - main copyright etc: <code>linux/kernel/softirq.c</code>	466
84	Soft IRQs - main comment: <code>linux/kernel/softirq.c</code>	467
85	Soft IRQs - handler i: <code>linux/kernel/softirq.c</code>	468
86	Soft IRQs - handler ii: <code>linux/kernel/softirq.c</code>	469
87	Soft IRQs - setup: <code>linux/kernel/softirq.c</code>	470

88	Soft IRQs - tasklet handler: <code>linux/kernel/softirq.c</code>	471
89	Soft IRQs - tasklet hi priority handler: <code>linux/kernel/softirq.c</code>	472
90	Soft IRQs - tasklet initialisation and destruction: <code>linux/kernel/softirq.c</code>	473
91	Soft IRQs - handler for legacy bottom half code: <code>linux/kernel/softirq.c</code>	474
92	Soft IRQs - handler for legacy bottom half code: <code>linux/kernel/softirq.c</code>	475
93	Soft IRQs - initialise softirq and run tasklets: <code>linux/kernel/softirq.c</code>	476
94	Network/OS Device Level Abstraction 0: <code>linux/include/linux/netdevice.h</code>	477
95	Network/OS Device Level Abstraction 1: <code>linux/include/linux/netdevice.h</code>	478
96	Network/OS Device Level Abstraction 2: <code>linux/include/linux/netdevice.h</code>	479
97	Network/OS Device Level Abstraction 3: <code>linux/include/linux/netdevice.h</code>	480
98	Network/OS Device Level Abstraction 4: <code>linux/include/linux/netdevice.h</code>	481
99	Network Device Driver Rx: <code>linux/net/core/dev.c</code>	482
100	Ethernet Driver 0 - data: <code>linux/drivers/net/3c501.c</code>	483
101	Ethernet Driver 1 - data: <code>linux/drivers/net/3c501.c</code>	484
102	Ethernet Driver 2 - init a: <code>linux/drivers/net/3c501.c</code>	485
103	Ethernet Driver 2 - init b: <code>linux/drivers/net/3c501.c</code>	486
104	Ethernet Driver 2 - init c: <code>linux/drivers/net/3c501.c</code>	487
105	Ethernet Driver 2 - init d: <code>linux/drivers/net/3c501.c</code>	488
106	Ethernet Driver 2 - txstart a: <code>linux/drivers/net/3c501.c</code>	489
107	Ethernet Driver 2 - txstart a: <code>linux/drivers/net/3c501.c</code>	490
108	Ethernet Driver 2 - txstart b: <code>linux/drivers/net/3c501.c</code>	491
109	Ethernet Driver 2 - txstart b: <code>linux/drivers/net/3c501.c</code>	492
110	Ethernet Driver 3 - interrupt a1: <code>linux/drivers/net/3c501.c</code>	493
111	Ethernet Driver 3 - interrupt a2: <code>linux/drivers/net/3c501.c</code>	494
112	Ethernet Driver 3 - interrupt b1: <code>linux/drivers/net/3c501.c</code>	495
113	Ethernet Driver 3 - interrupt b2: <code>linux/drivers/net/3c501.c</code>	496
114	Ethernet Driver 3 - interrupt c: <code>linux/drivers/net/3c501.c</code>	497
115	Ethernet Driver 3 - interrupt d: <code>linux/drivers/net/3c501.c</code>	498
116	Ethernet Driver 4 - receive: <code>linux/drivers/net/3c501.c</code>	499
117	Ethernet Driver 4 - receive: <code>linux/drivers/net/3c501.c</code>	500
118	Ethernet Driver 4 - stats: <code>linux/drivers/net/3c501.c</code>	501
119	Netstat -i	502
120	ifconfig	503
121	Ethernet Driver 4 - multicast: <code>linux/drivers/net/3c501.c</code>	504
122	Ethernet Driver 4 - modules: <code>linux/drivers/net/3c501.c</code>	505
123	Socket Interface Code - authors etc: <code>linux/net/socket.c</code>	506
124	Socket Interface Code - main copyright etc: <code>linux/net/socket.c</code>	507
125	Socket Interface Code - Prototypes: <code>linux/net/socket.c</code>	508
126	Socket Interface Code - file operation mappings etc: <code>linux/net/socket.c</code>	509
127	Socket Interface Code - Protocol Families supported: <code>linux/net/socket.c</code>	510
128	Socket Interface Code - lock code: <code>linux/net/socket.c</code>	511
129	Socket Interface Code - Global Counters and statistics: <code>linux/net/socket.c</code>	512
130	Check buffer addresses passed between user/kernel: <code>linux/net/socket.c</code>	513
131	More legality checks: <code>linux/net/socket.c</code>	514
132	Socket Glue - socket statfs: <code>linux/net/socket.c</code>	515
133	Socket Glue - socket fs read: <code>linux/net/socket.c</code>	516
134	Socket Glue - socket fs mnt, delete, dentry: <code>linux/net/socket.c</code>	517
135	Socket Glue - seek (and ye shall not find!): <code>linux/net/socket.c</code>	518
136	Socket Glue - mmap: <code>linux/net/socket.c</code>	519
137	Socket Glue - find a spare file descriptor: <code>linux/net/socket.c</code>	520

138	Socket Glue - get a file descriptor for a sock fd: <code>linux/net/socket.c</code>	521
139	Socket Glue - get a file descriptor for a sock fd: <code>linux/net/socket.c</code>	522
140	Socket Glue - lookup a file descriptor to a sock fd: <code>linux/net/socket.c</code>	523
141	Socket Glue - lookup a file descriptor to a sock fd: <code>linux/net/socket.c</code>	524
142	Socket Glue - allocate a socket <i>inode</i> : <code>linux/net/socket.c</code>	525
143	Socket Glue - free a socket <i>inode</i> : <code>linux/net/socket.c</code>	526
144	Socket Glue - block “opening” a socket <i>inode</i> : <code>linux/net/socket.c</code>	527
145	Socket Glue - shutdown: <code>linux/net/socket.c</code>	528
146	Socket Glue - send and receive messages on a socket: <code>linux/net/socket.c</code>	529
147	Socket Glue - read: <code>linux/net/socket.c</code>	530
148	Socket Glue - write: <code>linux/net/socket.c</code>	531
149	Socket Glue - readv/writev service routine: <code>linux/net/socket.c</code>	532
150	Socket Glue - readv and writev: <code>linux/net/socket.c</code>	533
151	Socket Glue - sendto i: <code>linux/net/socket.c</code>	534
152	Socket Glue - sendto ii: <code>linux/net/socket.c</code>	535
153	Socket Glue - recvfrom: <code>linux/net/socket.c</code>	536
154	Socket Glue - recvfrom: <code>linux/net/socket.c</code>	537
155	Socket Glue - recvfrom: <code>linux/net/socket.c</code>	538
156	Socket Glue - ioctl: <code>linux/net/socket.c</code>	539
157	Socket Glue - poll: <code>linux/net/socket.c</code>	540
158	Socket Glue - close: <code>linux/net/socket.c</code>	541
159	Socket Glue - asynch events on sockets i: <code>linux/net/socket.c</code>	542
160	Socket Glue - asynch events on sockets, ii: <code>linux/net/socket.c</code>	543
161	Socket Glue - asynch events on sockets, iii: <code>linux/net/socket.c</code>	544
162	Socket Glue - asynch events on sockets else off, ii: <code>linux/net/socket.c</code>	545
163	Socket Glue - finish asynch events on sockets: <code>linux/net/socket.c</code>	546
164	Socket Glue - create a socket i: <code>linux/net/socket.c</code>	547
165	Socket Glue - create a socket ii: <code>linux/net/socket.c</code>	548
166	Socket Glue - create a socket: <code>linux/net/socket.c</code>	549
167	Socket Glue - asm glue to create a socket: <code>linux/net/socket.c</code>	550
168	Socket Glue - asm glue to create a socket pair: <code>linux/net/socket.c</code>	551
169	Socket Glue - ... to create a socket pair: <code>linux/net/socket.c</code>	552
170	Socket Glue - bind a socket: <code>linux/net/socket.c</code>	553
171	Socket Glue - listen on a socket: <code>linux/net/socket.c</code>	554
172	Socket Glue - accept on a socket: <code>linux/net/socket.c</code>	555
173	Socket Glue - accept on a socket i: <code>linux/net/socket.c</code>	556
174	Socket Glue - accept on a socket ii: <code>linux/net/socket.c</code>	557
175	Socket Glue - connect on a socket: <code>linux/net/socket.c</code>	558
176	Socket Glue - get the local name (“address”) of a socket: <code>linux/net/socket.c</code>	559
177	Socket Glue - get the far end’s name (“address”) for a socket: <code>linux/net/socket.c</code>	560
178	Socket Glue - setsockopt: <code>linux/net/socket.c</code>	561
179	Socket Glue - getsockopt: <code>linux/net/socket.c</code>	562
180	Socket Glue - sendmsg i: <code>linux/net/socket.c</code>	563
181	Socket Glue - sendmsg ii: <code>linux/net/socket.c</code>	564
182	Socket Glue - recvmsg i: <code>linux/net/socket.c</code>	565
183	Socket Glue - recvmsg ii: <code>linux/net/socket.c</code>	566
184	Socket Glue - file control (fcntl): <code>linux/net/socket.c</code>	567
185	Socket Glue - system call vector: <code>linux/net/socket.c</code>	568
186	Socket Glue - system call vector: <code>linux/net/socket.c</code>	569

187	Socket Glue - system call vector: <code>linux/net/socket.c</code>	570
188	Socket Glue - registering a protocol family with the socket module: <code>linux/net/socket.c</code>	571
189	Socket Glue - un-registering a protocol family with the socket module: <code>linux/net/socket.c</code>	572
190	Socket Glue - initialising the socket VFS module: <code>linux/net/socket.c</code>	573
191	Socket Glue - initialising the socket VFS module: <code>linux/net/socket.c</code>	574
192	Socket Glue - socket stats per CPU: <code>linux/net/socket.c</code>	575
193	<code>sk_buff</code> structure layout	576
194	Memory Allocation:- Socket Buffers i: <code>linux/include/linux/skbuff.h</code>	577
195	Memory Allocation:- Socket Buffers ii: <code>linux/include/linux/skbuff.h</code>	578
196	Memory Allocation:- Socket Buffers Routines: <code>linux/include/linux/skbuff.h</code>	579
197	Memory Allocation:- Socket Buffers Code	580
198	Memory Allocation:- Socket Buffers Code	581
199	TCP Memory Allocation:- Socket Buffers Code	582
200	Socket Buffer Kernel Globals: <code>linux/net/core/skbuff.c</code>	583
201	Socket Buffer Head from/to Pool: <code>linux/net/core/skbuff.c</code>	584
202	Allocate Socket Buffer - comment: <code>linux/net/core/skbuff.c</code>	585
203	Allocate Socket Buffer: <code>linux/net/core/skbuff.c</code>	586
204	Initialise Socket Buffer: <code>linux/net/core/skbuff.c</code>	587
205	Free Socket Buffer: <code>linux/net/core/skbuff.c</code>	588
206	Clone Socket Buffer: <code>linux/net/core/skbuff.c</code>	589
207	Copy Socket Buffer Structure Header: <code>linux/net/core/skbuff.c</code>	590
208	Copy Socket Buffer Structure <i>and</i> data: <code>linux/net/core/skbuff.c</code>	591
209	Copy and Expand Socket Buffer Structure <i>and</i> data: <code>linux/net/core/skbuff.c</code>	592
210	Initialise Socket Buffer Global data: <code>linux/net/core/skbuff.c</code>	593
211	TCP States: <code>linux/include/net/tcp.h</code>	594
212	TCP State flags: <code>linux/include/net/tcp.h</code>	595
213	TCP Conenction State: <code>linux/include/net/tcp.h</code>	596
214	TCP State variables: <code>linux/include/net/tcp.h</code>	597
215	Socket TCP State i, start of...: <code>linux/include/net/sock.h</code>	598
216	Socket relevant IP option State: <code>linux/include/net/sock.h</code>	599
217	Socket IPv6 State: <code>linux/include/net/sock.h</code>	600
218	Socket State i, demultiplex data: <code>linux/include/net/sock.h</code>	601
219	Socket State ii, resource data: <code>linux/include/net/sock.h</code>	602
220	Socket State iii, socket state itself: <code>linux/include/net/sock.h</code>	603
221	Socket State iv, routing, erroring and filtering: <code>linux/include/net/sock.h</code>	604
222	Socket State v, private protocol info: <code>linux/include/net/sock.h</code>	605
223	Socket State vi, timers and callbacks: <code>linux/include/net/sock.h</code>	606
224	Socket State vi, timers and callbacks: <code>linux/include/net/sock.h</code>	607
225	Proposal to Tidy Up Socket State: <code>linux/include/net/sock.h</code>	608
226	Socket Reference Counting Comment: <code>linux/include/net/sock.h</code>	609
227	Inline functions for Socket Reference Management: <code>linux/include/net/sock.h</code>	610
228	Inline functions for Socket Reference Management: <code>linux/include/net/sock.h</code>	611
229	Inline functions for Socket Reference Management: <code>linux/include/net/sock.h</code>	612
230	Socket Route State Management: <code>linux/include/net/sock.h</code>	613
231	Socket Route State Management: <code>linux/include/net/sock.h</code>	614
232	Socket Route State Management: <code>linux/include/net/sock.h</code>	615
233	Socket Queue Management Interface: <code>linux/include/net/sock.h</code>	616
234	Socket Queue Management Interface: <code>linux/include/net/sock.h</code>	617

235	Socket Queue Management Interface: <code>linux/include/net/sock.h</code>	618
236	Socket Credential Function: <code>linux/net/core/scm.c</code>	619
237	Socket Credential Interface: <code>linux/include/net/scm.h</code>	620
238	BSD producer consumer chain - transport out	621
239	BSD producer consumer chain - network out	622
240	BSD producer consumer chain - legend	623
241	IP Output - add header and send: <code>linux/net/ipv4/ip_output.c</code>	624
242	IP Output - finish sending - note two stage process: <code>linux/net/ipv4/ip_output.c</code>	625
243	Internet Protocol - output - checksum: <code>linux/net/ipv4/ip_output.c</code>	626
244	IP Output - loopback: <code>linux/net/ipv4/ip_output.c</code>	627
245	IP Output - reroute?: <code>linux/net/ipv4/ip_output.c</code>	628
246	IP Output - multicast replication...: <code>linux/net/ipv4/ip_output.c</code>	629
247	IP Output - multicast replication...: <code>linux/net/ipv4/ip_output.c</code>	630
248	IP Output - drive output: <code>linux/net/ipv4/ip_output.c</code>	631
249	IP Output - Queue for sending - part 2i: <code>linux/net/ipv4/ip_output.c</code>	632
250	IP Output - Queue for sending - part 2ii: <code>linux/net/ipv4/ip_output.c</code>	633
251	IP Output - Queue packet for sending - part 1: <code>linux/net/ipv4/ip_output.c</code>	634
252	IP Output - Queue packet for sending - part 1: <code>linux/net/ipv4/ip_output.c</code>	635
253	IP Output - classic building of IP packet: <code>linux/net/ipv4/ip_output.c</code>	636
254	IP Output - classic building of IP packet: <code>linux/net/ipv4/ip_output.c</code>	637
255	IP Output - classic building of IP packet: <code>linux/net/ipv4/ip_output.c</code>	638
256	IP Output - classic building of IP packet: <code>linux/net/ipv4/ip_output.c</code>	639
257	IP Output - classic building of IP packet: <code>linux/net/ipv4/ip_output.c</code>	640
258	IP Output - classic building of IP packet: <code>linux/net/ipv4/ip_output.c</code>	641
259	IP Output - classic building of IP packet: <code>linux/net/ipv4/ip_output.c</code>	642
260	IP Output - classic building of IP packet: <code>linux/net/ipv4/ip_output.c</code>	643
261	IP Output - classic building of IP packet: <code>linux/net/ipv4/ip_output.c</code>	644
262	IP Output - fast path for non fragmented packets: <code>linux/net/ipv4/ip_output.c</code>	645
263	IP Output - fast path for non fragmented packets: <code>linux/net/ipv4/ip_output.c</code>	646
264	IP Output - fast path for non fragmented packets: <code>linux/net/ipv4/ip_output.c</code>	647
265	IP Output - fragment: <code>linux/net/ipv4/ip_output.c</code>	648
266	IP Output - fragment: <code>linux/net/ipv4/ip_output.c</code>	649
267	IP Output - fragment: <code>linux/net/ipv4/ip_output.c</code>	650
268	IP Output - fragment: <code>linux/net/ipv4/ip_output.c</code>	651
269	IP Output - fragment: <code>linux/net/ipv4/ip_output.c</code>	652
270	IP Output - deal with iovec and checksum for next function: <code>linux/net/ipv4/ip_output.c</code>	653
271	IP Output - build a packet that is a response to something: <code>linux/net/ipv4/ip_output.c</code>	654
272	IP Output - initialise IP layer: <code>linux/net/ipv4/ip_output.c</code>	655
273	Internet Protocol - Input- preamble: <code>linux/net/ipv4/ip_input.c</code>	656
274	IP Input - stats for network managers: <code>linux/net/ipv4/ip_input.c</code>	657
275	IP Input - Router Alter Handler: <code>linux/net/ipv4/ip_input.c</code>	658
276	IP Input - find ip proto - not inline...: <code>linux/net/ipv4/ip_input.c</code>	659
277	IP Input - deliver IP packets 2nd stage: <code>linux/net/ipv4/ip_input.c</code>	660
278	IP Input - deliver IP packets 1st stage: <code>linux/net/ipv4/ip_input.c</code>	661
279	IP Input - finish IP reception: <code>linux/net/ipv4/ip_input.c</code>	662
280	IP Input - finish IP reception: <code>linux/net/ipv4/ip_input.c</code>	663
281	IP Input - Main IP reception routine: <code>linux/net/ipv4/ip_input.c</code>	664
282	IP Input - Main IP reception routine: <code>linux/net/ipv4/ip_input.c</code>	665

283	Internet Protocol - Forward last part:	linux/net/ipv4/ip_forward.c	666
284	Internet Protocol - Forward first bit:	linux/net/ipv4/ip_forward.c	667
285	Internet Protocol - Forward first bit:	linux/net/ipv4/ip_forward.c	668
286	Internet Protocol - Fragmentation- Data:	linux/net/ipv4/ip_fragment.c	669
287	Fragmentation - Hash Data:	linux/net/ipv4/ip_fragment.c	670
288	Fragmentation - memory tracking:	linux/net/ipv4/ip_fragment.c	671
289	Fragmentation - free up ipq:	linux/net/ipv4/ip_fragment.c	672
290	Fragmentation - free up space starting with oldest frags:	linux/net/ipv4/ip_fragment.c	673
291	Fragmentation - reassembly timeout, expire a packet:	linux/net/ipv4/ip_fragment.c	674
292	Fragmentation - create fragment:	linux/net/ipv4/ip_fragment.c	675
293	Fragmentation - queue whole packet:	linux/net/ipv4/ip_fragment.c	676
294	Fragmentation - match a fragment to rest of datagram in partial queue:	linux/net/ipv4/ip_fragment.c	677
295	Fragmentation - add segment to queue:	linux/net/ipv4/ip_fragment.c	678
296	Fragmentation - add segment to queue:	linux/net/ipv4/ip_fragment.c	679
297	Fragmentation - add segment to queue:	linux/net/ipv4/ip_fragment.c	680
298	Fragmentation - add segment to queue:	linux/net/ipv4/ip_fragment.c	681
299	Fragmentation - add segment to queue:	linux/net/ipv4/ip_fragment.c	682
300	Fragmentation - put fragments together:	linux/net/ipv4/ip_fragment.c	683
301	Fragmentation - put fragments together:	linux/net/ipv4/ip_fragment.c	684
302	Fragmentation - put fragments together:	linux/net/ipv4/ip_fragment.c	685
303	Fragmentation - deal with arrival of a fragment:	linux/net/ipv4/ip_fragment.c	686
304	Internet Protocol - Options - Build IP option header:	linux/net/ipv4/ip_options.c	687
305	IP Options - Echo:	linux/net/ipv4/ip_options.c	688
306	IP Options - Echo:	linux/net/ipv4/ip_options.c	689
307	IP Options - Echo:	linux/net/ipv4/ip_options.c	690
308	IP Options - Echo:	linux/net/ipv4/ip_options.c	691
309	IP Options - zero out disallowed options in frags:	linux/net/ipv4/ip_options.c	692
310	IP Options - Validate inbound packet header options:	linux/net/ipv4/ip_options.c	693
311	IP Options - Validate inbound packet header options:	linux/net/ipv4/ip_options.c	694
312	IP Options - Validate inbound packet header options:	linux/net/ipv4/ip_options.c	695
313	IP Options - Validate inbound packet header options:	linux/net/ipv4/ip_options.c	696
314	IP Options - Validate inbound packet header options:	linux/net/ipv4/ip_options.c	697
315	IP Options - Validate inbound packet header options:	linux/net/ipv4/ip_options.c	698
316	IP Options - end of options processing:	linux/net/ipv4/ip_options.c	699
317	IP Options - unmangle what previous function did!:	linux/net/ipv4/ip_options.c	700
318	IP Options - make space for options:	linux/net/ipv4/ip_options.c	701
319	IP Options - forwarding options - e.g. route record:	linux/net/ipv4/ip_options.c	702
320	IP Options - receive SSR and process:	linux/net/ipv4/ip_options.c	703
321	IP Options - receive SSR and process:	linux/net/ipv4/ip_options.c	704
322	Internet Protocol - Socket Glue -some spare constants:	linux/net/ipv4/ip_sockglue.c	705
323	IP Socket Glue - various CMSGs:	linux/net/ipv4/ip_sockglue.c	706
324	IP Socket Glue - various CMSGs:	linux/net/ipv4/ip_sockglue.c	707
325	IP Socket Glue - field cmsgs to right place:	linux/net/ipv4/ip_sockglue.c	708
326	IP Socket Glue - send cmsgs:	linux/net/ipv4/ip_sockglue.c	709
327	IP Socket Glue - Router Alert handler for user level:	linux/net/ipv4/ip_sockglue.c	710
328	IP Socket Glue - Router Alert handler for user level:	linux/net/ipv4/ip_sockglue.c	711
329	IP Socket Glue - ICMP error to socket error mapping:	linux/net/ipv4/ip_sockglue.c	712
330	IP Socket Glue - local error problem:	linux/net/ipv4/ip_sockglue.c	713
331	IP Socket Glue - map IP reception errors to socket:	linux/net/ipv4/ip_sockglue.c	714

332	IP Socket Glue - map IP reception errors to socket: linux/net/ipv4/ip_sockglue.c	715
333	IP Socket Glue - setsockopt entry for IP: linux/net/ipv4/ip_sockglue.c	716
334	IP Socket Glue - setsockopt entry for IP: linux/net/ipv4/ip_sockglue.c	717
335	IP Socket Glue - setsockopt entry for IP: linux/net/ipv4/ip_sockglue.c	718
336	IP Socket Glue - setsockopt entry for IP: linux/net/ipv4/ip_sockglue.c	719
337	IP Socket Glue - setsockopt entry for IP: linux/net/ipv4/ip_sockglue.c	720
338	IP Socket Glue - setsockopt entry for IP: linux/net/ipv4/ip_sockglue.c	721
339	IP Socket Glue - setsockopt entry for IP: linux/net/ipv4/ip_sockglue.c	722
340	IP Socket Glue - setsockopt entry for IP: linux/net/ipv4/ip_sockglue.c	723
341	IP Socket Glue - getsockopt entry for IP: linux/net/ipv4/ip_sockglue.c	724
342	IP Socket Glue - getsockopt entry for IP: linux/net/ipv4/ip_sockglue.c	725
343	IP Socket Glue - getsockopt entry for IP: linux/net/ipv4/ip_sockglue.c	726
344	IP Socket Glue - getsockopt entry for IP: linux/net/ipv4/ip_sockglue.c	727
345	IP Socket Glue - getsockopt entry for IP: linux/net/ipv4/ip_sockglue.c	728
346	Address Resolution Protocol - Data : linux/net/ipv4/arp.c . . .	729
347	ARP Neighbour Operations : linux/net/ipv4/arp.c	730
348	ARP Neighbour Operations : linux/net/ipv4/arp.c	731
349	ARP Table Structure : linux/net/ipv4/arp.c	732
350	ARP Map MAC type/IEEE type: linux/net/ipv4/arp.c . . .	733
351	ARP Hash Function: linux/net/ipv4/arp.c	734
352	ARP Build Entry: linux/net/ipv4/arp.c	735
353	ARP Build Entry: linux/net/ipv4/arp.c	736
354	ARP Error Report: linux/net/ipv4/arp.c	737
355	ARP Solicit (Who Has): linux/net/ipv4/arp.c	738
356	ARP - even russians have Built in Obsolescence: linux/net/ipv4/arp.c	739
357	ARP - even russians...: linux/net/ipv4/arp.c	740
358	ARP - Bind address to neighbour: linux/net/ipv4/arp.c . . .	741
359	ARP - Send an ARP Message: linux/net/ipv4/arp.c	742
360	ARP - Send an ARP Message: linux/net/ipv4/arp.c	743
361	ARP - Send an ARP Message: linux/net/ipv4/arp.c	744
362	ARP - Receive an ARP Message: linux/net/ipv4/arp.c	745
363	ARP - Receive an ARP Message: linux/net/ipv4/arp.c	746
364	ARP - Receive an ARP Message: linux/net/ipv4/arp.c	747
365	ARP - Receive an ARP Message: linux/net/ipv4/arp.c	748
366	ARP - Receive an ARP Message: linux/net/ipv4/arp.c	749
367	ARP - Receive an ARP Message: linux/net/ipv4/arp.c	750
368	ARP - Receive an ARP Message: linux/net/ipv4/arp.c	751
369	ARP - User Interface (proc, ioctl) - Create a cache entry: linux/net/ipv4/arp.c	752
370	ARP - User Interface (proc, ioctl) - Create a cache entry: linux/net/ipv4/arp.c	753
371	ARP - User Interface - change ARP entry flags: linux/net/ipv4/arp.c	754
372	ARP - User Interface - Get an ARP entry: linux/net/ipv4/arp.c	755
373	ARP - User Interface - Delete an ARP entry: linux/net/ipv4/arp.c	756
374	ARP - User Interface - Delete an ARP entry: linux/net/ipv4/arp.c	757
375	ARP - User Interface - ioctl entry point: linux/net/ipv4/arp.c	758
376	ARP - User Interface - ioctl entry point: linux/net/ipv4/arp.c	759
377	ARP - User Interface - procfs entry point - write table: linux/net/ipv4/arp.c	760
378	ARP - User Interface - procfs entry point - write table: linux/net/ipv4/arp.c	761
379	ARP - User Interface - procfs entry point - write table: linux/net/ipv4/arp.c	762
380	ARP - Interface Down: linux/net/ipv4/arp.c	763
381	ARP - Initialise: linux/net/ipv4/arp.c	764

382	ARP - Amateur Radio X.25 ARP String Address Conversion: linux/net/ipv4/arp.c	765
383	ICMP, Hosts Requirement Conformance:-): linux/net/ipv4/icmp.c	766
384	ICMP, Hosts Requirement Conformance:-): linux/net/ipv4/icmp.c	767
385	ICMP, Router Requirement Conformance:-): linux/net/ipv4/icmp.c	768
386	ICMP, Router Requirement Conformance:-): linux/net/ipv4/icmp.c	769
387	ICMP, Router Requirement Conformance:-): linux/net/ipv4/icmp.c	770
388	CMP, Router Requirement Conformance:-): linux/net/ipv4/icmp.c	771
389	ICMP Global Data: linux/net/ipv4/icmp.c	772
390	ICMP Global Data: linux/net/ipv4/icmp.c	773
391	ICMP - transmit locks: linux/net/ipv4/icmp.c	774
392	ICMP transmit rate limiter - token bucket: linux/net/ipv4/icmp.c	775
393	ICMP transmit rate limit computation: linux/net/ipv4/icmp.c	776
394	ICMP/SNMP statistics counters: linux/net/ipv4/icmp.c	777
395	ICMP Checksum: linux/net/ipv4/icmp.c	778
396	ICMP Reply: linux/net/ipv4/icmp.c	779
397	ICMP Send: linux/net/ipv4/icmp.c	780
398	ICMP Send: linux/net/ipv4/icmp.c	781
399	ICMP Send: linux/net/ipv4/icmp.c	782
400	ICMP Send: linux/net/ipv4/icmp.c	783
401	ICMP Unreachables: linux/net/ipv4/icmp.c	784
402	ICMP Unreachables: linux/net/ipv4/icmp.c	785
403	ICMP Unreachables: linux/net/ipv4/icmp.c	786
404	ICMP Unreachables: linux/net/ipv4/icmp.c	787
405	ICMP Redirects: linux/net/ipv4/icmp.c	788
406	ICMP Echo: linux/net/ipv4/icmp.c	789
407	ICMP Timestamp: linux/net/ipv4/icmp.c	790
408	ICMP Address Mask (not!): linux/net/ipv4/icmp.c	791
409	ICMP Address Reply: linux/net/ipv4/icmp.c	792
410	ICMP discard: linux/net/ipv4/icmp.c	793
411	ICMP Receive General Packet: linux/net/ipv4/icmp.c	794
412	ICMP Statistics: linux/net/ipv4/icmp.c	795
413	ICMP Statistics: linux/net/ipv4/icmp.c	796
414	ICMP Initialisation: linux/net/ipv4/icmp.c	797
415	Internet Group Management Protocol- Data: linux/net/ipv4/igmp.c	798
416	IGMP Configure multicast interface: linux/net/ipv4/igmp.c	799
417	IGMP Timer Management: linux/net/ipv4/igmp.c	800
418	IGMP Timer Expired: linux/net/ipv4/igmp.c	801
419	IGMP Send Report: linux/net/ipv4/igmp.c	802
420	IGMP Send Report: linux/net/ipv4/igmp.c	803
421	IGMP Heard a Report: linux/net/ipv4/igmp.c	804
422	IGMP Heard a Query: linux/net/ipv4/igmp.c	805
423	IGMP Receive: linux/net/ipv4/igmp.c	806
424	IGMP Add and Delete Filter from interface: linux/net/ipv4/igmp.c	807
425	IGMP Group added or dropped: linux/net/ipv4/igmp.c	808
426	IGMP - socket joins group: linux/net/ipv4/igmp.c	809
427	IGMP Socket leaves group: linux/net/ipv4/igmp.c	810
428	IGMP join socket to group: linux/net/ipv4/igmp.c	811
429	IGMP join socket to group: linux/net/ipv4/igmp.c	812
430	IGMP: linux/net/ipv4/igmp.c	813

431	IGMP - socket closing:	linux/net/ipv4/igmp.c	814
432	IGMP - Device Goes Down or Comes Up:	linux/net/ipv4/igmp.c	815
433	IGMP - Destroy Device:	linux/net/ipv4/igmp.c	816
434	IGMP Find Device:	linux/net/ipv4/igmp.c	817
435	IPMR - device notifier:	linux/net/ipv4/ipmr.c	818
436	IGMP Check if address is in use on any interface:	linux/net/ipv4/igmp.c	819
437	IGMP - procfs - report membership etc:	linux/net/ipv4/igmp.c	820
438	IGMP - procfs - report membership etc:	linux/net/ipv4/igmp.c	821
439	Internet Multicast Routing Support - Data:	linux/net/ipv4/ipmr.c	822
440	Internet Multicast Routing Support - Data:	linux/net/ipv4/ipmr.c	823
441	IPMR - new vif/tunnel:	linux/net/ipv4/ipmr.c	824
442	IPMR - register a vif:	linux/net/ipv4/ipmr.c	825
443	IPMR - register a vif:	linux/net/ipv4/ipmr.c	826
444	IPMR - delete a vif:	linux/net/ipv4/ipmr.c	827
445	IPMR - Add a VIF:	linux/net/ipv4/ipmr.c	828
446	IPMR - Add a VIF:	linux/net/ipv4/ipmr.c	829
447	IPMR - cleanup vifs after socket close:	linux/net/ipv4/ipmr.c	830
448	IPMR - Destroy stale cache entry:	linux/net/ipv4/ipmr.c	831
449	IPMR - Timer process:	linux/net/ipv4/ipmr.c	832
450	IPMR - update output interface lists:	linux/net/ipv4/ipmr.c	833
451	IPMR - allocate cache entries:	linux/net/ipv4/ipmr.c	834
452	IPMR - resolve cache entries:	linux/net/ipv4/ipmr.c	835
453	IPMR - multicast route cache query passed up to mrouterd:	linux/net/ipv4/ipmr.c	836
454	IPMR - multicast route cache query passed up to mrouterd:	linux/net/ipv4/ipmr.c	837
455	IPMR - queue a packet to send awaiting resolution:	linux/net/ipv4/ipmr.c	838
456	IPMR - queue a packet to send awaiting resolution:	linux/net/ipv4/ipmr.c	839
457	IPMR - queue a packet to send awaiting resolution:	linux/net/ipv4/ipmr.c	840
458	IPMR - user space calls MFC delete:	linux/net/ipv4/ipmr.c	841
459	IPMR - user space calls MFC add:	linux/net/ipv4/ipmr.c	842
460	IPMR - user space calls MFC add:	linux/net/ipv4/ipmr.c	843
461	IPMR - mrouterd socket dies:	linux/net/ipv4/ipmr.c	844
462	IPMR - setsockopt entry handler:	linux/net/ipv4/ipmr.c	845
463	IPMR - setsockopt entry handler:	linux/net/ipv4/ipmr.c	846
464	IPMR - setsockopt entry handler:	linux/net/ipv4/ipmr.c	847
465	IPMR - getsockopt entry handler:	linux/net/ipv4/ipmr.c	848
466	IPMR - :	linux/net/ipv4/ipmr.c	849
467	IPMR - transmit encapsulated:	linux/net/ipv4/ipmr.c	850
468	IPMR - finish forwarding:	linux/net/ipv4/ipmr.c	851
469	IPMR - transmission handler:	linux/net/ipv4/ipmr.c	852
470	IPMR - transmission handler:	linux/net/ipv4/ipmr.c	853
471	IPMR - transmission handler:	linux/net/ipv4/ipmr.c	854
472	IPMR - find a vif by device:	linux/net/ipv4/ipmr.c	855
473	IPMR - forwarding:	linux/net/ipv4/ipmr.c	856
474	IPMR - forwarding:	linux/net/ipv4/ipmr.c	857
475	IPMR - handle multicast packets to be forwarded:	linux/net/ipv4/ipmr.c	858
476	IPMR - handle multicast packets to be forwarded:	linux/net/ipv4/ipmr.c	859
477	IPMR - PIM v1 input handler:	linux/net/ipv4/ipmr.c	860
478	IPMR - PIM v1 input handler:	linux/net/ipv4/ipmr.c	861
479	IPMR - PIM v2 input handler:	linux/net/ipv4/ipmr.c	862
480	IPMR - PIM v2 input handler:	linux/net/ipv4/ipmr.c	863

481	IPMR - RT Netlink fill route: <code>linux/net/ipv4/ipmr.c</code>	864
482	IPMR - RT Netlink get route: <code>linux/net/ipv4/ipmr.c</code>	865
483	IPMR - PROC FS vif reporting: <code>linux/net/ipv4/ipmr.c</code>	866
484	IPMR - PROC FS MFC reporting: <code>linux/net/ipv4/ipmr.c</code>	867
485	IPMR - PROC FS MFC reporting: <code>linux/net/ipv4/ipmr.c</code>	868
486	IPMR - initialise multicast routing: <code>linux/net/ipv4/ipmr.c</code>	869
487	Internet Protocol - Simple NAT: <code>linux/net/ipv4/ip_nat_dumb.c</code>	870
488	Internet Protocol - Simple NAT: <code>linux/net/ipv4/ip_nat_dumb.c</code>	871
489	Internet Protocol - Simple NAT: <code>linux/net/ipv4/ip_nat_dumb.c</code>	872
490	Internet Protocol - Simple NAT: <code>linux/net/ipv4/ip_nat_dumb.c</code>	873
491	IP in IP Tunnel - Opening Remarks: <code>linux/net/ipv4/ipip.c</code>	874
492	Internet Protocol - room for improvement: <code>linux/net/ipv4/ipip.c</code>	875
493	IP in IP - Data : <code>linux/net/ipv4/ipip.c</code>	876
494	IP in IP - Lookup Tunnel based on local and remote addresses: <code>linux/net/ipv4/ipip.c</code>	877
495	IP in IP - Tunnel hash bucket: <code>linux/net/ipv4/ipip.c</code>	878
496	IP in IP - unlink/link tunnel hash entry: <code>linux/net/ipv4/ipip.c</code>	879
497	IP in IP - locate a tunnel: <code>linux/net/ipv4/ipip.c</code>	880
498	IP in IP - locate a tunnel: <code>linux/net/ipv4/ipip.c</code>	881
499	IP in IP - uninitialise and destroy a tunnel: <code>linux/net/ipv4/ipip.c</code>	882
500	IP in IP - ICMP Error handling: <code>linux/net/ipv4/ipip.c</code>	883
501	IP in IP - ICMP Error handling: <code>linux/net/ipv4/ipip.c</code>	884
502	IP in IP - ICMP Error handling: <code>linux/net/ipv4/ipip.c</code>	885
503	IP in IP - ICMP Error handling: <code>linux/net/ipv4/ipip.c</code>	886
504	IP in IP - ICMP Error handling: <code>linux/net/ipv4/ipip.c</code>	887
505	IP in IP - ICMP Error handling: <code>linux/net/ipv4/ipip.c</code>	888
506	IP in IP - ECN handling: <code>linux/net/ipv4/ipip.c</code>	889
507	IP in IP - Receive Code: <code>linux/net/ipv4/ipip.c</code>	890
508	IP in IP - Tx Wrapper for NF Hook: <code>linux/net/ipv4/ipip.c</code>	891
509	IP in IP - Transmit Code: <code>linux/net/ipv4/ipip.c</code>	892
510	IP in IP - Transmit Code: <code>linux/net/ipv4/ipip.c</code>	893
511	IP in IP - Transmit Code: <code>linux/net/ipv4/ipip.c</code>	894
512	IP in IP - Transmit Code: <code>linux/net/ipv4/ipip.c</code>	895
513	IP in IP - Ioctl entry: <code>linux/net/ipv4/ipip.c</code>	896
514	IP in IP - Ioctl entry: <code>linux/net/ipv4/ipip.c</code>	897
515	IP in IP - Ioctl entry: <code>linux/net/ipv4/ipip.c</code>	898
516	IP in IP - stats: <code>linux/net/ipv4/ipip.c</code>	899
517	IP in IP - general initialisation: <code>linux/net/ipv4/ipip.c</code>	900
518	IP in IP - initialise a tunnel: <code>linux/net/ipv4/ipip.c</code>	901
519	IP in IP - module setup/teardown: <code>linux/net/ipv4/ipip.c</code>	902
520	IP in IP - module init: <code>linux/net/ipv4/ipip.c</code>	903
521	GRE Tunnels - Opening Salvo: <code>linux/net/ipv4/ip_gre.c</code>	904
522	GRE Tunnels - Opening Salvo: <code>linux/net/ipv4/ip_gre.c</code>	905
523	GRE Tunnel Data: <code>linux/net/ipv4/ip_gre.c</code>	906
524	GRE Tunnels - Hash Fucntion: <code>linux/net/ipv4/ip_gre.c</code>	907
525	GRE Tunnels - Hash Bucket: <code>linux/net/ipv4/ip_gre.c</code>	908
526	GRE Tunnels - link/unlink hash entry: <code>linux/net/ipv4/ip_gre.c</code>	909
527	GRE Tunnels - locate hash entry: <code>linux/net/ipv4/ip_gre.c</code>	910
528	GRE Tunnels - uninitialise and destory entry: <code>linux/net/ipv4/ip_gre.c</code>	911
529	GRE Tunnels- Receive Case: <code>linux/net/ipv4/ip_gre.c</code>	912

530	GRE Tunnels- Receive Case: linux/net/ipv4/ip_gre.c	913
531	GRE Tunnels- Receive Case: linux/net/ipv4/ip_gre.c	914
532	GRE Tunnels - Tx Wrapper: linux/net/ipv4/ip_gre.c	915
533	GRE Tunnels - Transmit Case: linux/net/ipv4/ip_gre.c	916
534	GRE Tunnels - Transmit Case: linux/net/ipv4/ip_gre.c	917
535	GRE Tunnels - Transmit Case: linux/net/ipv4/ip_gre.c	918
536	GRE Tunnels - Transmit Case: linux/net/ipv4/ip_gre.c	919
537	GRE Tunnels - Transmit Case: linux/net/ipv4/ip_gre.c	920
538	GRE Tunnels - Transmit Case: linux/net/ipv4/ip_gre.c	921
539	GRE Tunnels - Error/ICMP cases: linux/net/ipv4/ip_gre.c	922
540	GRE Tunnels - Error/ICMP cases: linux/net/ipv4/ip_gre.c	923
541	GRE Tunnels - Error/ICMP cases: linux/net/ipv4/ip_gre.c	924
542	GRE Tunnels - Error/ICMP cases: linux/net/ipv4/ip_gre.c	925
543	GRE Tunnels - Error/ICMP cases: linux/net/ipv4/ip_gre.c	926
544	GRE Tunnels - Error/ICMP cases: linux/net/ipv4/ip_gre.c	927
545	GRE Tunnels - Error/ICMP cases: linux/net/ipv4/ip_gre.c	928
546	GRE Tunnels - ECN Handling: linux/net/ipv4/ip_gre.c	929
547	GRE Tunnels - ioctl entry handler: linux/net/ipv4/ip_gre.c	930
548	GRE Tunnels - ioctl entry handler: linux/net/ipv4/ip_gre.c	931
549	GRE Tunnels - ioctl entry handler: linux/net/ipv4/ip_gre.c	932
550	GRE Tunnels - ioctl entry handler: linux/net/ipv4/ip_gre.c	933
551	GRE Tunnels - Broadcast Support!: linux/net/ipv4/ip_gre.c	934
552	GRE Tunnels - Broadcast Support!: linux/net/ipv4/ip_gre.c	935
553	GRE Tunnels - Broadcast Support!: linux/net/ipv4/ip_gre.c	936
554	GRE Tunnels - stats: linux/net/ipv4/ip_gre.c	937
555	GRE Tunnels - general initialisation: linux/net/ipv4/ip_gre.c	938
556	GRE Tunnels - tunnel initialisation: linux/net/ipv4/ip_gre.c	939
557	GRE Tunnels - tunnel initialisation: linux/net/ipv4/ip_gre.c	940
558	GRE Tunnels - module initialisation: linux/net/ipv4/ip_gre.c	941
559	GRE Tunnels - Protocol setup: linux/net/ipv4/ip_gre.c	942
560	IPv6 packet format: linux/include/linux/ipv6.h	943
561	Next Header Types: linux/include/net/ipv6.h	944
562	IPv6 address family information: linux/include/net/ipv6.h	945
563	IPv6 fragmentation header: linux/include/net/ipv6.h	946
564	IPv6 network interface structures: linux/include/net/if_inet6.h947	
565	IPv6 multicast interface structures: linux/include/net/if_inet6.h948	
566	IPv6 device configuration struct: linux/include/net/if_inet6.h949	
567	IPv6 device data: linux/include/net/if_inet6.h	950
568	IPv6 example of ethernet to ipv6 map: linux/include/net/if_inet6.h951	
569	IPv6 API - address family: linux/include/linux/in6.h	952
570	IPv6 API - multicast programmer interface: linux/include/linux/in6.h953	
571	IPv6 API - flow label programmer interface structure: linux/include/linux/in6.h954	
572	IP Address Configurator - Determine Address Type: linux/net/ipv6/addrconf.c955	
573	IP Address Configurator - Determine Address Type: linux/net/ipv6/addrconf.c956	
574	Select IPv6 Source Address given Destination to get to: linux/net/ipv6/addrconf.c957	
575	Select IPv6 Source Address given Destination to get to: linux/net/ipv6/addrconf.c958	
576	Select IPv6 Source Address given Destination to get to: linux/net/ipv6/addrconf.c959	
577	Select IPv6 Source Address given Destination to get to: linux/net/ipv6/addrconf.c960	
578	Deriving EUI64 bits of IPv6 Address: linux/net/ipv6/addrconf.c961	
579	Adding prefix route: linux/net/ipv6/addrconf.c	962

580	Adding prefix route: <code>linux/net/ipv6/addrconf.c</code>	963
581	Adding prefix route: <code>linux/net/ipv6/addrconf.c</code>	964
582	Adding prefix route: <code>linux/net/ipv6/addrconf.c</code>	965
583	Adding prefix route: <code>linux/net/ipv6/addrconf.c</code>	966
584	Adding prefix route: <code>linux/net/ipv6/addrconf.c</code>	967
585	Adding prefix route: <code>linux/net/ipv6/addrconf.c</code>	968
586	IPv6 <code>addrconf</code> sysctl: <code>linux/net/ipv6/addrconf.c</code>	969
587	IPv6 <code>addrconf</code> sysctl: <code>linux/net/ipv6/addrconf.c</code>	970
588	IPv6 Neighbour Discovery Operations: <code>linux/net/ipv6/ndisc.c</code>	971
589	IPv6 Neighbour Advertisement: <code>linux/net/ipv6/ndisc.c</code>	972
590	IPv6 Neighbour Advertisement: <code>linux/net/ipv6/ndisc.c</code>	973
591	IPv6 Router Discovery: <code>linux/net/ipv6/ndisc.c</code>	974
592	IPv6 Router Discovery: <code>linux/net/ipv6/ndisc.c</code>	975
593	IPv6 Router Discovery: <code>linux/net/ipv6/ndisc.c</code>	976
594	IPv6 Router Discovery: <code>linux/net/ipv6/ndisc.c</code>	977
595	IPv6 Router Discovery: <code>linux/net/ipv6/ndisc.c</code>	978
596	IPv6 Receive: <code>linux/net/ipv6/ip6_input.c</code>	979
597	IPv6 Receive: <code>linux/net/ipv6/ip6_input.c</code>	980
598	IPv6 Receive Finish: <code>linux/net/ipv6/ip6_input.c</code>	981
599	IPv6 - Output: <code>linux/net/ipv6/ip6_output.c</code>	982
600	IPv6 - Xmt: <code>linux/net/ipv6/ip6_output.c</code>	983
601	IPv6 - Output: <code>linux/net/ipv6/ip6_output.c</code>	984
602	IPv6 - Output: <code>linux/net/ipv6/ip6_output.c</code>	985
603	IPv6 - Output: <code>linux/net/ipv6/ip6_output.c</code>	986
604	IPv6 - Output: <code>linux/net/ipv6/ip6_output.c</code>	987
605	IPv6 - Output: <code>linux/net/ipv6/ip6_output.c</code>	988
606	IPv6 - Output: <code>linux/net/ipv6/ip6_output.c</code>	989
607	IPv6 - Output: <code>linux/net/ipv6/ip6_output.c</code>	990
608	IPv6 - Forward: <code>linux/net/ipv6/ip6_output.c</code>	991
609	IPv6 - Forward: <code>linux/net/ipv6/ip6_output.c</code>	992
610	IPv6 - Forward: <code>linux/net/ipv6/ip6_output.c</code>	993
611	IPv6 - Socket Glue Set Socket Options: <code>linux/net/ipv6/ipv6_sockglue.c</code>	994
612	IPv6 - Socket Glue Set Socket Options: <code>linux/net/ipv6/ipv6_sockglue.c</code>	995
613	IPv6 - Socket Glue Set Socket Options: <code>linux/net/ipv6/ipv6_sockglue.c</code>	996
614	IPv6 - Socket Glue Set Socket Options: <code>linux/net/ipv6/ipv6_sockglue.c</code>	997
615	IPv6 - Socket Glue Set Socket Options: <code>linux/net/ipv6/ipv6_sockglue.c</code>	998
616	IPv6 - Socket Glue Set Socket Options: <code>linux/net/ipv6/ipv6_sockglue.c</code>	999
617	IPv6 - Socket Glue Set Socket Options: <code>linux/net/ipv6/ipv6_sockglue.c</code>	1000
618	IPv6 Flow Label Creation: <code>linux/net/ipv6/ip6_flowlabel.c</code>	1001
619	IPv6 Flow Label Creation: <code>linux/net/ipv6/ip6_flowlabel.c</code>	1002
620	IPv6 Flow Label Creation: <code>linux/net/ipv6/ip6_flowlabel.c</code>	1003
621	IPv6 Extension Headers: <code>linux/net/ipv6/exthdrs.c</code>	1004
622	IPv6 Extension Headers: <code>linux/net/ipv6/exthdrs.c</code>	1005
623	SIT: <code>linux/net/ipv6/sit.c</code>	1006
624	SIT: <code>linux/net/ipv6/sit.c</code>	1007
625	A sample from the <code>/etc/services</code> file	1008
626	linkage structures and functions (i)	1010
627	linkage structures and functions (ii)	1011
628	start of the sock structure <code>linux{2.4.4/include/net/sock.h}</code>	1013
629	Strace of a DNS lookup	1014

630	UDP connect function <code>linux{2.4.4/net/ipv4/udp.c}</code>	1015
631	UDP lookup disconnect function <code>linux{2.4.4/net/ipv4/udp.c}</code>	1016
632	Building and sending a udp packet (from <code>udp_sendmsg</code>)	1017
633	UDP lookup socket function <code>linux{2.4.4/net/ipv4/udp.c}</code>	1018
634	UDP actions after lookup <code>linux{2.4.4/net/ipv4/udp.c}</code>	1019
635	Sliding window operation	1020
636	Strace output from a web browse with lynx	1021
637	tcpflow output from a web browsing session	1022
638	strace of a server process	1024
639	The TCP header	1025
640	TCP States (from RFC793)	1026
641	The TCP Connection creation state diagram (from RFC793)	1027
642	Assignment to <code>tp</code>	1028
643	<code>tcp_opt</code> (i)	1029
644	<code>tcp_opt</code> (ii)	1030
645	<code>tcp_opt</code> (iii)	1031
646	<code>tcp_opt</code> (iv)	1032
647	<code>tcp_opt</code> (v)	1033
648	<code>tcp_skb_cb</code> structure	1034
649	Start of <code>tcp_sendmsg</code>	1035
650	Determine MSS	1036
651	Start of send loop	1037
652	Finding space to copy the data to (i)	1038
653	Finding space to copy the data to (ii)	1039
654	Rest of <code>tcp_send</code>	1040
655	Rest of <code>tcp_send</code>	1041
656	Writing socket buffers to the network	1042
657	Should we send	1043
658	Building a TCP header (i)	1044
659	Building a TCP header (ii)	1045
660	Building a TCP header (ii)	1046
661	Checksumming TCP packets	1047
662	Receiving a packet by the TCP layer (i)	1048
663	Receiving a packet by the TCP layer (ii)	1049
664	An optimisation	1050
665	Receiving a socket buffer on an established connection (i)	1051
666	Receiving a socket buffer on an established connection (ii)	1052
667	Receiving a socket buffer on an established connection (iii)	1053
668	Receiving a socket buffer on an established connection (iv)	1054

669	Receiving a socket buffer on an established connection (iv)	1055
670	TCP sequence number check	1056
671	TCP receive window determination	1057
672	Ack sending code	1058
673	Processing an acknowledgement (i)	1059
674	Processing an acknowledgment (ii)	1060
675	processing and queuing incoming data (i)	1061
676	processing and queuing incoming data (ii)	1062
677	processing and queuing incoming data (iii)	1063
678	processing and queuing incoming data (iv)	1064
679	Timer Initialisation	1065
680	TCP congestion avoidance	1066
681	Recording the need to selectively acknowledge an out-of-order segment (i)	1067
682	Recording the need to selectively acknowledge an out-of-order segment (ii)	1068
683	Recording the need to send a dsack	1069
684	Building TCP option fields (timestamp and sack)	1070
685	Checking for a sack renegeing receiver	1071
686	PAWS checking	1072
687	Round Trip Estimation (i)	1073
688	Round Trip Estimation (ii)	1074
689	Round Trip Estimation (iii)	1075
690	Building options on SYN packets	1076
691	Parsing the WINDOW option	1077
692	Hash Tables	1078
693	IP Addresses and Masks and Route Advertisements: <code>linux{2.4.4}/net/ipv4/tcp_input.c</code>	1079
694	The FIB	1080
695	FIB Route Table Attribute Entry: <code>linux/include/net/ip_fib.h</code>	1081
696	FIB Next Hop Data Structure: <code>linux/include/net/ip_fib.h</code>	1082
697	The Main FIB Structure: <code>linux/include/net/ip_fib.h</code>	1083
698	FIB Lookup Result Structure: <code>linux/include/net/ip_fib.h</code>	1084
699	FIB Generic Interface: <code>linux/include/net/ip_fib.h</code>	1085
700	FIB Frontend i setting up the table: <code>linux/net/ipv4/fib_frontend.c</code>	1086
701	FIB Frontend i - flushing table: <code>linux/net/ipv4/fib_frontend.c</code>	1087
702	FIB Frontend ProcFS interface: <code>linux/net/ipv4/fib_frontend.c</code>	1088
703	FIB Front end - find device by address: <code>linux/net/ipv4/fib_frontend.c</code>	1089
704	FIB Frontend - find by address type: <code>linux/net/ipv4/fib_frontend.c</code>	1090
705	FIB Frontend check address makes sense i: <code>linux/net/ipv4/fib_frontend.c</code>	1091
706	FIB Frontend check address makes sense ii: <code>linux/net/ipv4/fib_frontend.c</code>	1092
707	FIB Frontend ioctl handler: <code>linux/net/ipv4/fib_frontend.c</code>	1093
708	FIB Frontend route find/add/delete: <code>linux/net/ipv4/fib_frontend.c</code>	1094
709	FIB Frontend dump FIB: <code>linux/net/ipv4/fib_frontend.c</code>	1095
710	FIB Frontend - intra-kernel route message: <code>linux/net/ipv4/fib_frontend.c</code>	1096
711	FIB Frontend - intra-kernel route message: <code>linux/net/ipv4/fib_frontend.c</code>	1097
712	FIB Frontend - intra-kernel route message: <code>linux/net/ipv4/fib_frontend.c</code>	1098
713	FIB Frontend add entry by interface address: <code>linux/net/ipv4/fib_frontend.c</code>	1099
714	FIB Frontend delete entry by interface address i: <code>linux/net/ipv4/fib_frontend.c</code>	1100
715	FIB Frontend delete entry by interface address ii: <code>linux/net/ipv4/fib_frontend.c</code>	1101
716	FIB Frontend Event Handler: <code>linux/net/ipv4/fib_frontend.c</code>	1102

717	FIB Frontend Event Handler:	linux/net/ipv4/fib_frontend.c	1103
718	FIB Frontend Event Handler:	linux/net/ipv4/fib_frontend.c	1104
719	FIB Frontend initialisation:	linux/net/ipv4/fib_frontend.c	1105
720	FIB Semantics - looping macros over fib entries:	linux/net/ipv4/fib_semantics.c	1106
721	FIB Semantics - initial Route Attribute properties:	linux/net/ipv4/fib_semantics.c	1107
722	FIB Semantics - find and release next hop info:	linux/net/ipv4/fib_semantics.c	1108
723	FIB Semantics - lookup:	linux/net/ipv4/fib_semantics.c	1109
724	FIB Semantics - set defaults:	linux/net/ipv4/fib_semantics.c	1110
725	FIB Semantics - add up next hops and get nhfs:	linux/net/ipv4/fib_semantics.c	1111
726	FIB Semantics - next hops match:	linux/net/ipv4/fib_semantics.c	1112
727	FIB Semantics - next hops match:	linux/net/ipv4/fib_semantics.c	1113
728	FIB Semantics - the Big Picture:	linux/net/ipv4/fib_semantics.c	1114
729	FIB Semantics - Check Next Hop:	linux/net/ipv4/fib_semantics.c	1115
730	FIB Semantics - Check Next Hop:	linux/net/ipv4/fib_semantics.c	1116
731	FIB Semantics - Create FIB Entry - i:	linux/net/ipv4/fib_semantics.c	1117
732	FIB Semantics - Create FIB Entry - i:	linux/net/ipv4/fib_semantics.c	1118
733	FIB Semantics - Create FIB Entry - ii:	linux/net/ipv4/fib_semantics.c	1119
734	FIB Semantics - Create FIB Entry - iii:	linux/net/ipv4/fib_semantics.c	1120
735	FIB Semantics - Create FIB Entry - iv:	linux/net/ipv4/fib_semantics.c	1121
736	FIB Semantics - Meaningful Match?:	linux/net/ipv4/fib_semantics.c	1122
737	FIB Semantics - Meaningful Match?:	linux/net/ipv4/fib_semantics.c	1123
738	FIB Semantics - Dump utilities i:	linux/net/ipv4/fib_semantics.c	1124
739	FIB Semantics - Dump utilities ii:	linux/net/ipv4/fib_semantics.c	1125
740	FIB Semantics - Dump utilities iii:	linux/net/ipv4/fib_semantics.c	1126
741	FIB Semantics - convert RT request:	linux/net/ipv4/fib_semantics.c	1127
742	FIB Semantics - export to RT entry i:	linux/net/ipv4/fib_semantics.c	1128
743	FIB Semantics - export to RT entry ii:	linux/net/ipv4/fib_semantics.c	1129
744	FIB Semantics - update fib if address or interface down i:	linux/net/ipv4/fib_semantics.c	1130
745	FIB Semantics - update fib if address or interface down ii:	linux/net/ipv4/fib_semantics.c	1131
746	FIB Semantics - update fib if address or interface up!:	linux/net/ipv4/fib_semantics.c	1132
747	FIB Semantics - multipath case i:	linux/net/ipv4/fib_semantics.c	1133
748	FIB Semantics - multipath case ii:	linux/net/ipv4/fib_semantics.c	1134
749	FIB Semantics - PROC FS Interface:	linux/net/ipv4/fib_semantics.c	1135
750	FIB Rules - globals:	linux/net/ipv4/fib_rules.c	1136
751	FIB Rules i:	linux/net/ipv4/fib_rules.c	1137
752	FIB Rules - utils:	linux/net/ipv4/fib_rules.c	1138
753	FIB Rules - new rule:	linux/net/ipv4/fib_rules.c	1139
754	FIB Rules - new rule:	linux/net/ipv4/fib_rules.c	1140
755	FIB Rules - new rule:	linux/net/ipv4/fib_rules.c	1141
756	FIB Rules - more utils:	linux/net/ipv4/fib_rules.c	1142
757	FIB Rules - attach and detach rule:	linux/net/ipv4/fib_rules.c	1143
758	FIB Rules -lookup rule:	linux/net/ipv4/fib_rules.c	1144
759	FIB Rules -lookup rule:	linux/net/ipv4/fib_rules.c	1145
760	FIB Rules Events:	linux/net/ipv4/fib_rules.c	1146
761	FIB Rules - initialisation:	linux/net/ipv4/fib_rules.c	1147
762	FIB Rules - initialisation:	linux/net/ipv4/fib_rules.c	1148
763	Fast Accessing the FIB through hashing - types i:	linux/net/ipv4/fib_hash.c	1149
764	Fast Accessing the FIB through hashing - types ii:	linux/net/ipv4/fib_hash.c	1150
765	Fast Accessing the FIB through hashing - inline functions i:	linux/net/ipv4/fib_hash.c	1151

766	Fast Accessing the FIB through hashing - inline functions ii: linux/net/ipv4/fib_hash.c	1152
767	Fast Accessing the FIB through hashing - inline functions iii: linux/net/ipv4/fib_hash.c	1153
768	Fast Accessing the FIB through hashing - zone lookup functions: linux/net/ipv4/fib_hash.c	1154
769	Fast Accessing the FIB through hashing - hash lookup function: linux/net/ipv4/fib_hash.c	1155
770	Fast Accessing the FIB through hashing - dead entry: linux/net/ipv4/fib_hash.c	1156
771	Fast Accessing the FIB through hashing - select default entry: linux/net/ipv4/fib_hash.c	1157
772	Fast Accessing the FIB through hashing - select default entry: linux/net/ipv4/fib_hash.c	1158
773	Fast Accessing the FIB through hashing - insert entry i: linux/net/ipv4/fib_hash.c	1159
774	Fast Accessing the FIB through hashing - insert entry ii: linux/net/ipv4/fib_hash.c	1160
775	Fast Accessing the FIB through hashing - insert entry iii: linux/net/ipv4/fib_hash.c	1161
776	Fast Accessing the FIB through hashing - insert entry iv: linux/net/ipv4/fib_hash.c	1162
777	Fast Accessing the FIB through hashing - insert entry v: linux/net/ipv4/fib_hash.c	1163
778	Fast Accessing the FIB through hashing - insert entry v: linux/net/ipv4/fib_hash.c	1164
779	Fast Accessing the FIB through hashing - delete entry i: linux/net/ipv4/fib_hash.c	1165
780	Fast Accessing the FIB through hashing - delete entry ii: linux/net/ipv4/fib_hash.c	1166
781	Fast Accessing the FIB through hashing - delete entry iii: linux/net/ipv4/fib_hash.c	1167
782	Fast Accessing the FIB through hashing - flush entries: linux/net/ipv4/fib_hash.c	1168
783	Accessing the FIB -Proc FS access : linux/net/ipv4/fib_hash.c	1169
784	Accessing the FIB - RT access i: linux/net/ipv4/fib_hash.c .	1170
785	Accessing the FIB - RT access ii: linux/net/ipv4/fib_hash.c .	1171
786	Accessing the FIB - RT access iii: linux/net/ipv4/fib_hash.c	1172
787	Accessing the FIB - RT access iv: linux/net/ipv4/fib_hash.c	1173
788	Accessing the FIB - Initialisation: linux/net/ipv4/fib_hash.c	1174
789	igmp	1175
790	PIM SM	1176
791	The Route Interface i: linux/include/net/route.h	1177
792	The Route Interface ii: linux/include/net/route.h	1178
793	The Route Interface iii: linux/include/net/route.h	1179
794	Basic Route Maintenance: linux/net/ipv4/route.c	1180
795	Basic Route Maintenance Ops: linux/net/ipv4/route.c	1181
796	API for Route Maintenance: linux/net/core/rtnetlink.c	1182
797	The Route API Hash Key: linux/include/net/route.h	1183
798	The Route API Table: linux/include/net/route.h	1184
799	Route Output Lookup: linux/net/ipv4/route.c	1185
800	Route Lookup i : linux/net/ipv4/route.c	1186
801	Route Lookup ii - source case: linux/net/ipv4/route.c	1187
802	Route Lookup iii - oif case: linux/net/ipv4/route.c	1188
803	Route Lookup - loopback case: linux/net/ipv4/route.c	1189
804	Route Lookup: oh, need a full FIB lookup linux/net/ipv4/route.c	1190
805	Route Lookup: corner cases linux/net/ipv4/route.c	1191
806	Route Lookup route request: linux/net/ipv4/route.c	1192
807	Route Lookup route request tidy up refs ("put"): linux/net/ipv4/route.c	1193
808	Route Lookup route request tidy up refs ("put"): linux/net/ipv4/route.c	1194
809	Guaranteed Bounded Delay and Utilisation	1195

810	Network Hierarchy	1196
811	mixing traffic onto one wire	1197
812	fifo sharing	1198
813	Leaky Bucket Description	1199
814	Token Bucket Description	1200
815	Router Schematic	1201
816	Class Based Queueing	1202
817	qdisc call graph structure	1203
818	queueing discipline fine grain structure: classification via filters .	1204
819	queueing discipline applying a filter	1205
820	Classifier: linux/net/sched/cls_api.c	1206
821	Classifier i: linux/net/sched/cls_api.c	1207
822	Classifier ii: linux/net/sched/cls_api.c	1208
823	Classifier iii: linux/net/sched/cls_api.c	1209
824	Classifier iv: linux/net/sched/cls_api.c	1210
825	Classifier iv: linux/net/sched/cls_api.c	1211
826	Classifier v: linux/net/sched/cls_api.c	1212
827	Classifier v: linux/net/sched/cls_api.c	1213
828	Classifier vi: linux/net/sched/cls_api.c	1214
829	Classifier vi: linux/net/sched/cls_api.c	1215
830	Classifier vii: linux/net/sched/cls_api.c	1216
831	Classifier viii: linux/net/sched/cls_api.c	1217
832	Classifier - Dump Output: linux/net/sched/cls_api.c	1218
833	Classifier - Dump Output: linux/net/sched/cls_api.c	1219
834	Classifier - Dump Output: linux/net/sched/cls_api.c	1220
835	Classifier- Initialisation: linux/net/sched/cls_api.c	1221
836	Universal 32 bit Classifier API- Global Data: linux/net/sched/cls_u32.c	1222
837	u32 API- the Classifier i: linux/net/sched/cls_u32.c	1223
838	u32 API- the Classifier ii: linux/net/sched/cls_u32.c	1224
839	u32 API- lookup fucntions: linux/net/sched/cls_u32.c	1225
840	u32 API- Misc Support: linux/net/sched/cls_u32.c	1226
841	u32 API- Initialisation: linux/net/sched/cls_u32.c	1227
842	u32 API- Destroy/Delete Key: linux/net/sched/cls_u32.c	1228
843	u32 API- Destroy/Delete Classifier: linux/net/sched/cls_u32.c	1229
844	u32 API- New Key Id: linux/net/sched/cls_u32.c	1230
845	u32 API- Set Parameters: linux/net/sched/cls_u32.c	1231
846	u32 API- Change Parameters: linux/net/sched/cls_u32.c	1232
847	u32 API- Change Parameters: linux/net/sched/cls_u32.c	1233
848	u32 API- Walk Tree: linux/net/sched/cls_u32.c	1234
849	u32 API- RTNETLINK Comms: linux/net/sched/cls_u32.c	1235
850	u32 API- TCF Operationlinux/net/sched/Registration: linux/net/sched/cls_u32.c	1236
851	RSVP Classifier Interface- Opening Comment: linux/net/sched/cls_rsvp.h	1237
852	RSVP - Structures: linux/net/sched/cls_rsvp.h	1238
853	RSVP - Police function: linux/net/sched/cls_rsvp.h	1239
854	RSVP - Classifier Function i: linux/net/sched/cls_rsvp.h	1240
855	RSVP - Classifier Function ii: linux/net/sched/cls_rsvp.h	1241
856	RSVP - Misc Functions: linux/net/sched/cls_rsvp.h	1242
857	RSVP - Delete: linux/net/sched/cls_rsvp.h	1243
858	RSVP - Destroy: linux/net/sched/cls_rsvp.h	1244
859	RSVP - general bits: linux/net/sched/cls_rsvp.h	1245

860	RSVP - recycle or generate tunnel:	linux/net/sched/cls_rsvp.h	1246
861	RSVP - Change Parameters i:	linux/net/sched/cls_rsvp.h	1247
862	RSVP - Change Parameters ii:	linux/net/sched/cls_rsvp.h	1248
863	RSVP - Change Parameters iii:	linux/net/sched/cls_rsvp.h	1249
864	RSVP - Change Parameters iv:	linux/net/sched/cls_rsvp.h	1250
865	RSVP - Change Parameters v:	linux/net/sched/cls_rsvp.h	1251
866	RSVP - Walk RSVP tree:	linux/net/sched/cls_rsvp.h	1252
867	RSVP Classifier API:	linux/net/sched/cls_rsvp.c	1253
868	Route Classifier API - Structures:	linux/net/sched/cls_route.c	1254
869	Route Classifier API - Functions including Policer:	linux/net/sched/cls_route.c	1255
870	Route Classifier API - Classifier Function i:	linux/net/sched/cls_route.c	1256
871	Route Classifier API - Classifier Function ii:	linux/net/sched/cls_route.c	1257
872	Route Classifier API - Classifier Function iii:	linux/net/sched/cls_route.c	1258
873	Route Classifier API - Misc:	linux/net/sched/cls_route.c	1259
874	Route Classifier API - Destroy Route Classifier:	linux/net/sched/cls_route.c	1260
875	Route Classifier API - Delete Entry:	linux/net/sched/cls_route.c	1261
876	Firewall Classifier API - Structures:	linux/net/sched/cls_fw.c	1262
877	Firewall Classifier API - Classifier:	linux/net/sched/cls_fw.c	1263
878	Firewall Classifier API - Misc:	linux/net/sched/cls_fw.c	1264
879	Firewall Classifier API - Destroy Classifier:	linux/net/sched/cls_fw.c	1265
880	Firewall Classifier API Delete Classifier:	linux/net/sched/cls_fw.c	1266
881	Firewall Classifier API - Change Classifier:	linux/net/sched/cls_fw.c	1267
882	Firewall Classifier API - Change Classifier ii:	linux/net/sched/cls_fw.c	1268
883	Firewall Classifier API - Change Classifier iii:	linux/net/sched/cls_fw.c	1269
884	Firewall Classifier API - Change Classifier iv:	linux/net/sched/cls_fw.c	1270
885	TC Classifier API - Defines:	linux/net/sched/cls_tcindex.c	1271
886	TC Classifier API - lookup:	linux/net/sched/cls_tcindex.c	1272
887	TC Classifier API - Classifier:	linux/net/sched/cls_tcindex.c	1273
888	TC Classifier API - Misc:	linux/net/sched/cls_tcindex.c	1274
889	TC Classifier API - delete classifier:	linux/net/sched/cls_tcindex.c	1275
890	TC Classifier API - Destroy Classifier:	linux/net/sched/cls_tcindex.c	1276
891	TC Classifier API - change classifier i:	linux/net/sched/cls_tcindex.c	1277
892	TC Classifier API - change classifier ii:	linux/net/sched/cls_tcindex.c	1278
893	TC Classifier API - change classifier iii:	linux/net/sched/cls_tcindex.c	1279
894	TC Classifier API - change classifier iii:	linux/net/sched/cls_tcindex.c	1280
895	Scheduler API i:	linux/net/sched/sch_api.c	1281
896	Scheduler API auxilliary routines:	linux/net/sched/sch_api.c	1282
897	Scheduler API all the disciplines:	linux/net/sched/sch_api.c	1283
898	Scheduler API - register/unregister a discipline:	linux/net/sched/sch_api.c	1284
899	Scheduler API - find a discipline by handle:	linux/net/sched/sch_api.c	1285
900	Scheduler API - find a discipline by name:	linux/net/sched/sch_api.c	1286
901	Scheduler API - find a discipline by rate:	linux/net/sched/sch_api.c	1287
902	Scheduler API - free rate table entry:	linux/net/sched/sch_api.c	1288
903	Scheduler API - allocate discipline a unique handle:	linux/net/sched/sch_api.c	1289
904	Scheduler API - graft queue discipline to a device:	linux/net/sched/sch_api.c	1290
905	Scheduler API - graft queue discipline child to parent class:	linux/net/sched/sch_api.c	1291
906	Scheduler API - allocate and initialise new queue discipline:	linux/net/sched/sch_api.c	1292
907	Scheduler API - allocate and initialise new queue discipline:	linux/net/sched/sch_api.c	1293
908	Scheduler API - allocate and initialise new queue discipline:	linux/net/sched/sch_api.c	1294
909	Scheduler API - modify new queue discipline:	linux/net/sched/sch_api.c	1295

910	Scheduler API - detect loops in queue operations graph:	linux/net/sched/sch_api.c	1296
911	Scheduler API - destroy Queue discipline entry:	linux/net/sched/sch_api.c	1297
912	Scheduler API - destroy Queue discipline entry:	linux/net/sched/sch_api.c	1298
913	Scheduler API - change Queue discipline entry:	linux/net/sched/sch_api.c	1299
914	Scheduler API - change Queue discipline entry:	linux/net/sched/sch_api.c	1300
915	Scheduler API - change Queue discipline entry:	linux/net/sched/sch_api.c	1301
916	Scheduler API - report stats via RT:	linux/net/sched/sch_api.c	1302
917	Scheduler API - NLM Message:	linux/net/sched/sch_api.c	1303
918	Scheduler API - notify RTNETLINK:	linux/net/sched/sch_api.c	1304
919	Scheduler API - dump statistics to RT:	linux/net/sched/sch_api.c	1305
920	Scheduler API - traffic class control:	linux/net/sched/sch_api.c	1306
921	Scheduler API - traffic class control:	linux/net/sched/sch_api.c	1307
922	Scheduler API - traffic class control:	linux/net/sched/sch_api.c	1308
923	Scheduler API - traffic class control:	linux/net/sched/sch_api.c	1309
924	Scheduler API - traffic class report to RTNETLINK:	linux/net/sched/sch_api.c	1310
925	Scheduler API - Packet Scheduling:	linux/net/sched/sch_api.c	1311
926	Scheduler API - Packet Scheduling Tick:	linux/net/sched/sch_api.c	1312
927	Scheduler API - Packet Scheduling Tick:	linux/net/sched/sch_api.c	1313
928	Scheduler API - Packet Scheduling Initialisation:	linux/net/sched/sch_api.c	1314
929	Scheduler API - Packet Scheduling Initialisation:	linux/net/sched/sch_api.c	1315
930	Scheduler API - Packet Scheduling Initialisation:	linux/net/sched/sch_api.c	1316
931	Packet Scheduler Interface - QDisc Walker Definition:	linux/include/net/pkt_sched.h	1317
932	Packet Scheduler Interface QDisc Class Operations:	linux/include/net/pkt_sched.h	1318
933	Packet Scheduler Interface QDisc Operations:	linux/include/net/pkt_sched.h	1319
934	Packet Scheduler Interface QDisc Structure:	linux/include/net/pkt_sched.h	1320
935	Packet Scheduler Interface QDisc Operations:	linux/include/net/pkt_sched.h	1321
936	Packet Scheduler Interface Clock Comment:	linux/include/net/pkt_sched.h	1322
937	Scheduler API ii:	linux/net/sched/sch_api.c	1323
938	Scheduler API discussion:	linux/net/sched/sch_api.c	1324
939	Scheduler API more discussion:	linux/net/sched/sch_api.c	1325
940	Scheduler API more discussion:	linux/net/sched/sch_api.c	1326
941	Scheduler API more:	linux/net/sched/sch_api.c	1327
942	A Generic Scheduler - Queue Discipline Lock Comment:	linux/net/sched/sch_generic.c	1328
943	A Generic Scheduler i:	linux/net/sched/sch_generic.c	1329
944	A Generic Scheduler i:	linux/net/sched/sch_generic.c	1330
945	A Generic Scheduler ii:	linux/net/sched/sch_generic.c	1331
946	A Generic Scheduler - Watchdog Timers:	linux/net/sched/sch_generic.c	1332
947	A Generic Scheduler - No-op Scheduler:	linux/net/sched/sch_generic.c	1333
948	A Generic Scheduler - Operations:	linux/net/sched/sch_generic.c	1334
949	A Generic Scheduler - PFIFO Operations:	linux/net/sched/sch_generic.c	1335
950	A Generic Scheduler - PFIFO Operations:	linux/net/sched/sch_generic.c	1336
951	A Generic Scheduler - PFIFO declare:	linux/net/sched/sch_generic.c	1337
952	A Generic Scheduler - create default entry:	linux/net/sched/sch_generic.c	1338
953	CBQ - Initial Comment i:	linux/net/sched/sch_cbq.c	1339
954	CBQ - Initial Comment ii:	linux/net/sched/sch_cbq.c	1340
955	CBQ - Class Structure Definition i:	linux/net/sched/sch_cbq.c	1341
956	CBQ - Class Structure Definition i:	linux/net/sched/sch_cbq.c	1342
957	CBQ - Class Structure Definition ii:	linux/net/sched/sch_cbq.c	1343
958	CBQ - Class Structure Definition iii:	linux/net/sched/sch_cbq.c	1344
959	CBQ - Class Structure Definition iii:	linux/net/sched/sch_cbq.c	1345

960	CBQ - Scheduling Data Forwarding: <code>linux/net/sched/sch_cbq.c</code>	1346
961	CBQ - Classifier - i: <code>linux/net/sched/sch_cbq.c</code>	1347
962	CBQ - Classifier - ii: <code>linux/net/sched/sch_cbq.c</code>	1348
963	CBQ - Re-Classifer: <code>linux/net/sched/sch_cbq.c</code>	1349
964	CBQ - Activate or Deactivate a Class: <code>linux/net/sched/sch_cbq.c</code>	1350
965	CBQ - Activate or Deactivate a Class: <code>linux/net/sched/sch_cbq.c</code>	1351
966	CBQ - Mark Top Level: <code>linux/net/sched/sch_cbq.c</code>	1352
967	CBQ - Enqueue: <code>linux/net/sched/sch_cbq.c</code>	1353
968	CBQ - Requeue: <code>linux/net/sched/sch_cbq.c</code>	1354
969	CBQ - overlimit: <code>linux/net/sched/sch_cbq.c</code>	1355
970	CBQ - overlimit: reclassify: <code>linux/net/sched/sch_cbq.c</code>	1356
971	CBQ - overlimit: delay: <code>linux/net/sched/sch_cbq.c</code>	1357
972	CBQ - overlimit: lower priority or even drop: <code>linux/net/sched/sch_cbq.c</code>	1358
973	CBQ - watchdog timer: <code>linux/net/sched/sch_cbq.c</code>	1359
974	CBQ - remove delay priority penalty: <code>linux/net/sched/sch_cbq.c</code>	1360
975	CBQ - remove delay penalty: <code>linux/net/sched/sch_cbq.c</code>	1361
976	CBQ - reshape: <code>linux/net/sched/sch_cbq.c</code>	1362
977	CBQ - update toplevel occupancy: <code>linux/net/sched/sch_cbq.c</code>	1363
978	CBQ - update each level's occupancy: <code>linux/net/sched/sch_cbq.c</code>	1364
979	CBQ - update each level's occupancy ii: <code>linux/net/sched/sch_cbq.c</code>	1365
980	CBQ - update each level's occupancy iii: <code>linux/net/sched/sch_cbq.c</code>	1366
981	CBQ - underlimit actions: <code>linux/net/sched/sch_cbq.c</code>	1367
982	CBQ - dequeue by priorities i: <code>linux/net/sched/sch_cbq.c</code>	1368
983	CBQ - dequeue by priorities ii: <code>linux/net/sched/sch_cbq.c</code>	1369
984	CBQ - dequeue by priorities iii: <code>linux/net/sched/sch_cbq.c</code>	1370
985	CBQ - dequeue by priorities iv: <code>linux/net/sched/sch_cbq.c</code>	1371
986	CBQ - dequeue 1: <code>linux/net/sched/sch_cbq.c</code>	1372
987	CBQ - main dequeue functon i: <code>linux/net/sched/sch_cbq.c</code>	1373
988	CBQ - main dequeue functon: <code>linux/net/sched/sch_cbq.c</code>	1374
989	CBQ - main dequeue functon: <code>linux/net/sched/sch_cbq.c</code>	1375
990	CBQ - Maintenance Functions: <code>linux/net/sched/sch_cbq.c</code>	1376
991	CBQ - Initialise CBQ: <code>linux/net/sched/sch_cbq.c</code>	1377
992	CBQ - Initialise CBQ ii: <code>linux/net/sched/sch_cbq.c</code>	1378
993	CBQ - Initialise CBQ iii: <code>linux/net/sched/sch_cbq.c</code>	1379
994	CBQ - Grafting and pruning classes: <code>linux/net/sched/sch_cbq.c</code>	1380
995	CBQ - Grafting and pruning classes ii: <code>linux/net/sched/sch_cbq.c</code>	1381
996	CBQ - Grafting and pruning classes iii: <code>linux/net/sched/sch_cbq.c</code>	1382
997	CBQ: <code>linux/net/sched/sch_cbq.c</code>	1383
998	CBQ: <code>linux/net/sched/sch_cbq.c</code>	1384
999	CBQ: <code>linux/net/sched/sch_cbq.c</code>	1385
1000	CBQ: <code>linux/net/sched/sch_cbq.c</code>	1386
1001	CBQ: <code>linux/net/sched/sch_cbq.c</code>	1387
1002	CBQ - delete CBQ from Qdisc : <code>linux/net/sched/sch_cbq.c</code>	1388
1003	Clark/Shenker/Zhang Scheduling - Initial Comment: <code>linux/net/sched/sch_csz.c</code>	1389
1004	CSZ Notation: <code>linux/net/sched/sch_csz.c</code>	1390
1005	CSZ Departure Times: <code>linux/net/sched/sch_csz.c</code>	1391
1006	CSZ - The Round: <code>linux/net/sched/sch_csz.c</code>	1392
1007	CSZ - The Round: <code>linux/net/sched/sch_csz.c</code>	1393
1008	CSZ Discussion: <code>linux/net/sched/sch_csz.c</code>	1394
1009	CSZ Discussion: <code>linux/net/sched/sch_csz.c</code>	1395

1010	FIFO Queueing Scheduler - data:	linux/net/sched/sch_fifo.c	1396
1011	FIFO Queueing Scheduler - enqueue:	linux/net/sched/sch_fifo.c	1397
1012	FIFO Queueing Scheduler - requeue:	linux/net/sched/sch_fifo.c	1398
1013	FIFO Queueing Scheduler - dequeue:	linux/net/sched/sch_fifo.c	1399
1014	FIFO Queueing Scheduler - drop:	linux/net/sched/sch_fifo.c	1400
1015	FIFO Queueing Scheduler - reset:	linux/net/sched/sch_fifo.c	1401
1016	FIFO Queueing Scheduler - enqueue:	linux/net/sched/sch_fifo.c	1402
1017	FIFO Queueing Scheduler - requeue:	linux/net/sched/sch_fifo.c	1403
1018	FIFO Queueing Scheduler - dequeue:	linux/net/sched/sch_fifo.c	1404
1019	FIFO Queueing Scheduler - init:	linux/net/sched/sch_fifo.c	1405
1020	Rate Estimator - general purpose block comment:	linux/net/sched/estimator.c	1406
1021	Rate Estimator - Data:	linux/net/sched/estimator.c	1407
1022	Rate Estimator - timer routine for estimation:	linux/net/sched/estimator.c	1408
1023	Rate Estimator - add a timer routine for estimation:	linux/net/sched/estimator.c	1409
1024	Rate Estimator - stop estimation timer routine for estimation:	linux/net/sched/estimator.c	1410
1025	Policer i:	linux/net/sched/police.c	1411
1026	Policer ii - destroy policer:	linux/net/sched/police.c	1412
1027	Policer iii - locate:	linux/net/sched/police.c	1413
1028	Policer iv - locate:	linux/net/sched/police.c	1414
1029	Policer iv - the Police!!!:	linux/net/sched/police.c	1415
1030	Policer v - the Police!!!:	linux/net/sched/police.c	1416
1031	Policer v - the Police!!!:	linux/net/sched/police.c	1417
1032	Ingress Queueing Scheduler - Data:	linux/net/sched/sch_ingress.c	1418
1033	Ingress Queueing - Tree handling:	linux/net/sched/sch_ingress.c	1419
1034	Ingress Queueing - Change Parameters:	linux/net/sched/sch_ingress.c	1420
1035	Ingress Queueing - Walk Tree:	linux/net/sched/sch_ingress.c	1421
1036	Ingress Queueing - Enqueue:	linux/net/sched/sch_ingress.c	1422
1037	Ingress Queueing - Dequeue:	linux/net/sched/sch_ingress.c	1423
1038	Ingress Queueing - Requeue:	linux/net/sched/sch_ingress.c	1424
1039	Ingress Queueing - Drop:	linux/net/sched/sch_ingress.c	1425
1040	Differentiated Services Field Marker - Data Types:	linux/net/sched/sch_dsmark.c	1426
1041	Differentiated Services Field Marker - tree functions:	linux/net/sched/sch_dsmark.c	1427
1042	Differentiated Services Field Marker - change parameters:	linux/net/sched/sch_dsmark.c	1428
1043	Marker - delete:	linux/net/sched/sch_dsmark.c	1429
1044	Marker - walk:	linux/net/sched/sch_dsmark.c	1430
1045	Marker - find:	linux/net/sched/sch_dsmark.c	1431
1046	Marker - enqueue i:	linux/net/sched/sch_dsmark.c	1432
1047	Marker - enqueue ii:	linux/net/sched/sch_dsmark.c	1433
1048	Marker - dequeue:	linux/net/sched/sch_dsmark.c	1434
1049	Marker - requeue:	linux/net/sched/sch_dsmark.c	1435
1050	Marker - drop:	linux/net/sched/sch_dsmark.c	1436
1051	Marker - init:	linux/net/sched/sch_dsmark.c	1437
1052	Marker - reset:	linux/net/sched/sch_dsmark.c	1438
1053	Marker - destroy:	linux/net/sched/sch_dsmark.c	1439
1054	Marker - dump class:	linux/net/sched/sch_dsmark.c	1440
1055	Marker - dump:	linux/net/sched/sch_dsmark.c	1441
1056	Marker - operations registration:	linux/net/sched/sch_dsmark.c	1442
1057	Random Early Detection (RED) Queue - Introductory Remarks:	linux/net/sched/sch_red.c	1443

1058	RED Queue - Behaviour:	linux/net/sched/sch_red.c	1444
1059	RED Queue - Parameters:	linux/net/sched/sch_red.c	1445
1060	RED Queue - Notes:	linux/net/sched/sch_red.c	1446
1061	RED - Scheduler Data:	linux/net/sched/sch_red.c	1447
1062	RED - ECN bit handling:	linux/net/sched/sch_red.c	1448
1063	RED - Enqueue i:	linux/net/sched/sch_red.c	1449
1064	RED - Enqueue ii:	linux/net/sched/sch_red.c	1450
1065	RED - Enqueue iii:	linux/net/sched/sch_red.c	1451
1066	RED - Enqueue i:	linux/net/sched/sch_red.c	1452
1067	RED - Requeue and Dequeue:	linux/net/sched/sch_red.c	1453
1068	RED - Drop:	linux/net/sched/sch_red.c	1454
1069	RED - Reset:	linux/net/sched/sch_red.c	1455
1070	RED - Change:	linux/net/sched/sch_red.c	1456
1071	RED - Init:	linux/net/sched/sch_red.c	1457
1072	Token Bucket Filter Queue - Initial Remarks:	linux/net/sched/sch_tbf.c	1458
1073	Token Bucket Filter Queue - Algorithm/Notes:	linux/net/sched/sch_tbf.c	1459
1074	Token Bucket Filter Queue - Data:	linux/net/sched/sch_tbf.c	1460
1075	TBF - Enqueue:	linux/net/sched/sch_tbf.c	1461
1076	TBF - Requeue and Drop:	linux/net/sched/sch_tbf.c	1462
1077	TBF - Watchdog:	linux/net/sched/sch_tbf.c	1463
1078	TBF - Dequeue:	linux/net/sched/sch_tbf.c	1464
1079	TBF - Dequeue:	linux/net/sched/sch_tbf.c	1465
1080	TBF - Reset:	linux/net/sched/sch_tbf.c	1466
1081	TBF - Change:	linux/net/sched/sch_tbf.c	1467
1082	TBF - Init:	linux/net/sched/sch_tbf.c	1468
1083	TBF - Destroy:	linux/net/sched/sch_tbf.c	1469
1084	Ingress Queueing - Ingress Hook:	linux/net/sched/sch_ingress.c	1470
1085	Ingress Queueing - Operations and Initialisation:	linux/net/sched/sch_ingress.c	1471
1086	Ingress Queueing - Destroy:	linux/net/sched/sch_ingress.c	1472
1087	Priority Queue:	linux/net/sched/sch_prio.c	1473
1088	Priority Classifier:	linux/net/sched/sch_prio.c	1474
1089	Priority Enqueue:	linux/net/sched/sch_prio.c	1475
1090	Priority Requeue and Dequeue:	linux/net/sched/sch_prio.c	1476
1091	Priority Drop:	linux/net/sched/sch_prio.c	1477
1092	Priority - Tune:	linux/net/sched/sch_prio.c	1478
1093	Stochastic Fairness Queueing Scheduler - Introductory Remarks:		
	linux/net/sched/sch_sfq.c		1479
1094	SFQ - Data Structures:	linux/net/sched/sch_sfq.c	1480
1095	Priority Queue:	linux/net/sched/sch_prio.c	1481
1096	Stochastic Fairness Queueing Scheduler - Introductory Remarks:		
	linux/net/sched/sch_sfq.c		1482
1097	SFQ - Data Structures:	linux/net/sched/sch_sfq.c	1483
1098	SFQ - Queue Hash Functions:	linux/net/sched/sch_sfq.c	1484
1099	SFQ - Queue Decrement/Increment Functions:	linux/net/sched/sch_sfq.c	1485
1100	SFQ - Queue Drop Function:	linux/net/sched/sch_sfq.c	1486
1101	SFQ - Enqueue Function:	linux/net/sched/sch_sfq.c	1487
1102	SFQ - Requeue Function:	linux/net/sched/sch_sfq.c	1488
1103	SFQ - Dequeue Function:	linux/net/sched/sch_sfq.c	1489
1104	SFQ - Reset:	linux/net/sched/sch_sfq.c	1490
1105	SFQ - Perturb Queue:	linux/net/sched/sch_sfq.c	1491

1106 Traffic Equalisation/Load Balance Queue - Initial Remarks: <code>linux/net/sched/sch_teql.c</code>	1492
1107 TE Data: <code>linux/net/sched/sch_teql.c</code>	1493
1108 TE Enqueue,Requeue,Dequeue: <code>linux/net/sched/sch_teql.c</code>	1494
1109 TE Release, Reset, Destroy: <code>linux/net/sched/sch_teql.c</code>	1495
1110 TE Initialisation: <code>linux/net/sched/sch_teql.c</code>	1496
1111 TE net device resolve: <code>linux/net/sched/sch_teql.c</code>	1497
1112 TE net device transmit: <code>linux/net/sched/sch_teql.c</code>	1498
1113 TE net device transmit ii: <code>linux/net/sched/sch_teql.c</code>	1499
1114 TE net device transmit iii: <code>linux/net/sched/sch_teql.c</code>	1500
1115 TE net device open: <code>linux/net/sched/sch_teql.c</code>	1501
1116 TE net device close and stats: <code>linux/net/sched/sch_teql.c</code>	1502
1117 TE net device mtu: <code>linux/net/sched/sch_teql.c</code>	1503
1118 TE net device master initialisation: <code>linux/net/sched/sch_teql.c</code>	1504
1119 TE net device register: <code>linux/net/sched/sch_teql.c</code>	1505
1120 Profiling - Timer: <code>linux/net/core/profile.c</code>	1506
1121 Profiling- White Hole Interface: <code>linux/net/core/profile.c</code>	1507
1122 Profiling- White Hole Interface: <code>linux/net/core/profile.c</code>	1508
1123 Profiling- White Hole- Stats and Init: <code>linux/net/core/profile.c</code>	1509
1124 Profiling- White Hole- Register/Unregister: <code>linux/net/core/profile.c</code>	1510
1125 ATM Scheduling - Author: <code>linux/net/sched/sch_atm.c</code>	1511
1126 ATM Scheduling - Introductory Remarks: <code>linux/net/sched/sch_atm.c</code>	1512
1127 ATM Scheduling - Data Structures: <code>linux/net/sched/sch_atm.c</code>	1513
1128 ATM Scheduling - Finding flows i: <code>linux/net/sched/sch_atm.c</code>	1514
1129 ATM Scheduling - Trees of Flows: <code>linux/net/sched/sch_atm.c</code>	1515
1130 ATM Scheduling - more Finding Flows: <code>linux/net/sched/sch_atm.c</code>	1516
1131 ATM Scheduling - destroy filters: <code>linux/net/sched/sch_atm.c</code>	1517
1132 ATM Scheduling - destroy all filters: <code>linux/net/sched/sch_atm.c</code>	1518
1133 ATM Scheduling - pop a VC: <code>linux/net/sched/sch_atm.c</code>	1519
1134 ATM Scheduling - Change ATM TC Parameters: <code>linux/net/sched/sch_atm.c</code>	1520
1135 ATM Scheduling - Change ATM TC Parameters ii: <code>linux/net/sched/sch_atm.c</code>	1521
1136 ATM Scheduling - Change ATM TC Parameters ii: <code>linux/net/sched/sch_atm.c</code>	1522
1137 ATM Scheduling - Delete TC: <code>linux/net/sched/sch_atm.c</code>	1523
1138 ATM Scheduling - Walk TC: <code>linux/net/sched/sch_atm.c</code>	1524
1139 ATM Scheduling - Find TCF: <code>linux/net/sched/sch_atm.c</code>	1525
1140 ATM Scheduling - Enqueue i: <code>linux/net/sched/sch_atm.c</code>	1526
1141 ATM Scheduling - Enqueue ii: <code>linux/net/sched/sch_atm.c</code>	1527
1142 ATM Scheduling - Enqueue iii: <code>linux/net/sched/sch_atm.c</code>	1528
1143 ATM Scheduling - Dequeue: <code>linux/net/sched/sch_atm.c</code>	1529
1144 ATM Scheduling - TC Dequeue: <code>linux/net/sched/sch_atm.c</code>	1530
1145 ATM Scheduling - TC Requeue: <code>linux/net/sched/sch_atm.c</code>	1531
1146 ATM Scheduling - TC Drop: <code>linux/net/sched/sch_atm.c</code>	1532
1147 ATM Scheduling - Initialise: <code>linux/net/sched/sch_atm.c</code>	1533
1148 TC - main: <code>tc/tc.c</code>	1534
1149 Masquerading Router	1535
1150 Masquerading Example	1536
1151 Port forwarding with a firewall	1537
1152 A proxy cache	1538
1153 A transparent proxy cache	1539
1154 Possible hook locations	1540

List of Tables

1	Integrated Versus Differentiated Services	1542
2	Kernel Critical Region Serialisation	1543
3	Spin Lock Outline	1544
4	Kernel spin read/write locks	1545
5	Source Code Directory Tree Organisation	1546
6	Source Code Network Sub-Directory Tree Organisation	1547
7	IPv4 Sub-Directory Tree Organisation	1548
8	Linux socket protocol families implemented	1549
9	Linux socket protocol semantics	1550
10	Other socket API calls	1551
11	Other socket API calls	1552
12	TCP sysctls	1553
13	TCP setsockopt options	1554
14	TCP ioctl options	1555
15	Terms in the Padhye TCP Equation	1556
16	multicast addressing	1557
17	Terms in the Parekh TCP Equation	1558
18	Elastic Applications	1559
19	Inelastic Applications	1560

Chapter 1

Overview of the Internet Protocols

“To some of us, preserving the Net for free speech is more important than anything in the free world.”, Ron Newman, netizen.

[Figure 1 about here.]

1.1 Roadmap

This chapter is a self contained overview of the Internet protocols. We start with an overview of the history of TCP/IP and then cover the basic concepts of IPv4, including routing, addressing, reliability and performance management. A brief introduction to IPv6, ipsec, mobile IP and multicast.

1.2 Introduction

The history of the Internet is the history of the world’s most successful software system. Unlike previous networks (postal, telephone, broadcast) which were largely built out of custom hardware, the Internet is mostly made from software components, running on relatively simple general purpose computing hardware.

Of course there is a little bit of hardware involved, as well as people using the system! The important point is that most of the hardware components are general purpose programmable computers, and as such, have all the flexibility that such devices enjoy. Of particular relevance to this book is the fact that the performance of the hardware in the form of personal computers running off the shelf operating systems such as Linux, is now sufficient to fulfill most of the important roles in the Internet.

What are the main components of the Internet? We can look at them from many viewpoints, but from 20,000 feet they might be the following:

Users Users are either people, or their agents, that access content over the Internet using networked applications to download web pages or move E-Mail or even run VOIP.

Content In the end, users want content- whether it is to access a web page, acquire a real-world good or service, or hear the voice of their friend.

Service Providers Service Providers are organisations to which user subscribe (have some contract-like relationship with) that manage connectivity by acquiring lower level services and coordinating them. Service providers may provide *content*, *applications* or *Internet* connectivity. This space is constantly expanding as more and more layers of business are refined out of the global information market! Traditionally, Internet Service Providers (“ISPs”) bought bit-transport capacity, and added value by managing a collection of links and routers. More recently, at least two more layers have been added, with Content and Application Service Providers offering “overlay” services, with collections of coordinated, managed end systems connected to one or more ISPs offering optimised performance for this set of user or that.

Hosts, or End Systems End systems are hosts (general purpose computers) that run client, server or peer-to-peer application software. The term has become someone muddied in recent years due to the emergence of the other business layers, and also due to some hacks required to make security and addressing deploy pragmatically (see chapter ten for some details on this!). Thus we have seen the evolution of “middle boxes”, which provide a variety of services ranging from proxying, caching, firewalls, address translation, and so on. Pure end systems have users (i.e. are clients, servers or peers).

Middle Boxes Middle Boxes are End Systems that don’t have explicit users (clients or servers) attached, but intermediate at the application, transport or network level in the protocol stream to provide some kind of value added service or optimisation.

Intermediate Systems, or Routers Intermediate systems that provide a router (or as it used to be commonly known, “IP gateway”) service are, from the point of view of end systems, almost completely transparent. They connect links together and construct a set of paths for packets, across a set of links. A router may be an end system for the purposes of management.

Links A link is some sort of piece of transmission medium that interconnects hosts and routers, or just routers. Links are increasingly diverse in their technologies and architectures. In fact, recently, it has become clear that the Internet uses some technologies that provide rather more than traditional serial line or local area network links as if they are merely point to point, or broadcast medium technologies. There are moves afoot to generalise the model of the Internet so that it can be used recursively below the IP layer (so called “generalised multi-protocol label switching”, for example), but there is neither time nor space to discuss this interesting topic here.

To glue this all together we have a vast infrastructure of software:

Applications Applications allow the user to copy or access files across the network, FTP or NFS, and to send and receive email and news such as SMTP and NNTP, or to browse or download content such as the family

of WWW protocols. Applications are usually instantiated via some type of client which nowadays has a graphical user interface (GUI), or even a voice or touch screen activated interface on palm devices.

Protocols Communicating applications cause flows of information. They do this by using protocols, which are programming rules for distributed applications. Most applications are only “distributed” in a limited sense of being client-to-server, or peer-to-peer, so that there is usually only a pair of systems to be coordinated from the user’s point of view. Of course, the Internet has many simultaneous users, so that some types of the Internet need to coordinate with many others. In fact, this is what distinguishes a pure end-system from a middle box, and also more specifically, from a router, since a pure end system need only concern itself explicitly with one other system, whereas all kinds of intermediaries necessarily concern themselves with multiple other systems simultaneously even in a given communications context. There is one exception to this rule which is multicast, which we’ll look at briefly in chapter eight.

Protocols include two main important types of rules in their definitions: packet formats, and rules of operation. In programming terms, packet formats are implemented as data structures with a specific machine independent encoding. Protocol rules of operation cannot be expressed in narrative form! There are far too many possible ways that a network can be used. Instead, they are typically expressed in terms of a set of states and actions. In chapter seven, we will look at the clearest example of this which is the way that the Transmission Control Protocol, TCP, is defined and implemented.

Layers: Physical, Network & Application Inevitably, the complexity of such a system of components is best understood by introducing some systematic way of categorising its modularisation. The natural way that designers tend to do this is by *layering*. In a network, it is clear that the physical transmission link and its framing rules for packets that the standard and Network Interface Card dictate is a given. The Internet Protocol *has* to be fitted “on top of” this. By this we mean that the Internet is an overlay on a service made out of lots of links of differing kinds, and in general, users of the Internet Protocol layer don’t really care what the details of the link layers are. This is an example of *information hiding*.

The idea of layering occurs in programming, as well as in design. When a design is layered, however, this does not imply that the implementation must slavishly follow the same boundaries. For example, the Application may actually care what the size of the physically largest packet a link can transmit is.

Names, Addresses, Routes These are confusing concepts - in a way we often define examples of these in the context of one layer, only to redefine them differently at different layers:

A *name* tells you what it is, e.g. “Clare Lecture Theatre”, or “Jon Crowcroft”, or <http://www.cs.ucl.ac.uk/index.html>

An *address* tells you where something is: e.g. “151 Gower Street”, or “Room 101”, or 128.16.5.31

A *route* tells you how to get there: e.g. “go in the 2nd door, up the stairs, turn left, and stop at the door marked **stop**”.

Security Security is generally concerned with defending against attacks such as theft or denial of service. It is hard for software to defend against physical attacks! However, we can provide means to reduce the chances of logical attacks. Logical attacks such as theft of content, or theft of identity, or theft of privacy cannot be completely ruled out, but there are a variety of techniques for making it expensive (i.e. computationally hard) to misbehave. The ones implemented in the Linux kernel and supporting software relevant to the actual lower protocol levels are discussed in chapter ten. However, security is part of a much larger picture and we would advise concerned users to read a good book on network security before embarking on any use of the Internet, for example on firewalls[?].

Metering, Accounting If you are providing a service, you may want to account for usage. There are quite a few tools for doing this at the microscopic level of detail - one advantage of Linux is that it is relatively easy to build on these and add layers of software to accumulate statistics and summarise them.

[Figure 2 about here.]

1.3 Protocols

Packets contain data and control.

To get information from memory to processor, the CPU uses a Bus protocol, and sends commands and addresses on some (parallel) wires, and gets data over other wires.

To get data across a network which is usually only 1 wire wide (serial), a computer uses a protocol, but everything must be in the messages - datagrams, like a telegram, carry addresses and data and other instructions to the postman

1.3.1 A Note on Programming Languages and Methodologies

Most of the Internet is programmed in the *C* language. This is especially true of lower level protocols and systems. Most operating systems running today are written in C, although many applications are programmed in higher level languages. There are several reasons for using C. Partly it is historic: previous systems had been written in assembler, but this is completely non portable and extremely hard to maintain. C reveals just enough of the underlying hardware to allow the programmer access to specific memory locations and structures, as well as interfacing to assembler, while offering enough high-level programming tools (functions, modules etc) to retain some degree of maintainability.

It is possible to write highly portable C, but it is also possible to write non portable code. It is also possible to write very obscure code (there are competitions in *obfuscated C!*), but luckily, the influence of several decades of experience has led to programming styles and conventions that make C code quite easy to comprehend.

These styles and conventions vary from community to community, however in the Linux system there are some fairly strong agreements on how code should be written, laid out, and integrated. One of the key advantages of open source approaches to systems development is that people learn by (good) examples. The Linux kernel also benefits additionally from strong central control over what is allowed into a release. A different model pertains with the systems that are packaged together by the large number of Linux *distribution* companies now in the world, where it is up to those organisations to maintain consistency.

More importantly, in general over the last few years, the practice of *Object Oriented Design and Programming* has become widespread. There was a trend towards using *C++* or even *Objective C* for a while, but these proved more complex than necessary. For application (and high level network) programming, *Java* is now in the ascendancy. However the techniques can be applied ‘semi-manually’, in C, and we will see that this is widespread throughout the Linux kernel code. In fact, there is a school of thought that it has always been the case amongst good programmers that they use patterns in common with “OO” programming!

For example, for a modular kernel, it is required to package up code with a standard set of methods (or functions and arguments and their types), and little means of modifying the internals of that module - this applies to device drivers, protocols within a layer, and even larger units of operating system functionality such as file systems and schedulers. At design time, this leads to safer code. At implementation time, it leads to extensibility. At build (“link”) time, this allows the system to be statically configured. Finally, at run time it allows a system to be configured dynamically as needed. This means that the system can be as small as needed.

It has a side effect on the design which is that usually, systems such as this are highly “regular”. That means that once a programmer has learned how to do something, it represents a general lesson.

C Style

There are quite a few different styles possible in C. Since there is a large number of examples in this book, we don’t think it is necessary to spell out the style here, but rather discuss the reasons for some rules (of thumb).

readability Having a standard layout makes it easier for folks to read your code (and find, and hopefully fix, bugs). It also allows people to share syntax directed editor functions (emacs, usually). Not all programmers have 3 21-inch displays at their disposal- its friendly to make the code fit on a standard screen/window width. In kernel code, change logging is absolutely essential. A single line at the start of the module is only enough if there are indications throughout of the changes referencing the author too!

automatic processing There are a number of programs that are used to process code (e.g. for generating call graphs, or code statistics). Some of these may rely on conventions for placing open-braces at start of line for example.

maintenance Code should contain enough comments that you can figure out its basic operation quickly, but not so many that you cannot find the actual statements. It is conventional to comment code in English. The bulk of Linux code is written by people who chose names for variables from the English vocabulary - its a ‘fait accompli’. Function and variable names should communicate use and function (and to some extent scope).

portability Be pedantic about types. Be extra pedantic about commenting the mapping between data structures and protocol wire-formats. Be clear about byte order, alignment of fields.

risk There are no reasons for deliberate obfuscation of statements- for example, if you have a complex logical expression, bracket it to make it clear rather than relying on (possibly incorrect) memory of operator precedence.

In an ideal world, all code is accompanied by separate documentation as well, written *at the same time* as the code (e.g. as dictated by the extreme programming style). Such documentation should include examples of use and examples of test output. The world is unfortunately less than 1% ideal.

C Structures, Memory and Wire Format

It is doubly useful that the fields in a compound data structure such as the C record structure, when compiled on a specific machine architecture with a specific compiler, has a known layout in memory. This allows the systems and communications programmer to be able to map external data into memory and vice versa (typically via buffers handed to input/output interfaces through some control register addressed in a device driver).

When reading a protocol definition, there are typically three places that the programmer needs to look, and in general the author finds the order that works best is:

1. Firstly, look at the packet format definitions - Internet protocols are handily (and not accidentally) designed *and* defined using fields that are typically convenient basic data types of a modern computer - byte (“octet”), and 16, 32 and 64 bit words. Most packet headers are aggregates of these (we’ll see IP and TCP and many others throughout the book). Making sure that we can access and manipulate the fields conveniently, correctly and efficiently is the first thing that we try to get right - normally, a packet header is defined as a C structure in a C header file.

The only glitches in this approach are to do with variations between C compilers and machine architectures for the fine details - e.g. “endian”ness of a byte, a short and long word. This is normally hidden in a single place through the use of functions or macros which can be used immediately pre-transmission, or post-reception, allowing the programmer to treat C fields as native data objects (e.g. for arithmetic) throughout the bulk of the code.

2. Rules of operation - e.g. a state machine, or language/grammar of the protocol. This can often directly be mapped into the core of an implementation (most TCP code for input packet handling is written directly line by line from the RFC!).

3. Interfaces to lower and higher layers, and to system functions. Here we can see the set of functions and parameters for the service interface for the protocol (called by a higher layer), the calls to send and receive (or be called with a packet that has been received), together with infrastructure calls (timers, memory allocation/freeing, critical region protection etc).

High level data structures (object oriented) is useful for large systems and for re-use - examples of this abound in Unix - e.g. APIs to filesystem, device drivers, network, and some cases quite complex emulations of C++ in C (e.g. Kuznetsov's elegant structures of operations in the queueing discipline code, which we will see in chapter nine).

The C language makes a nodding allowance for some higher level approaches to programming. Typically, we can keep code in a directory hierarchy (see later for the Linux kernel source code layout). Within a given directory, we keep the code for a given levels of abstraction. A collection of functions in a given source code file (a "module" in C terms; *not* to be confused with a "module" in Linux dynamic load terms!), perform closely related functions - almost literally, an object. To make an object more generic, C provides a simple, but extremely efficient and powerful mechanism of function pointers. A reference to a function can be passed as yet another parameter - this allows us to write code which can be quite generic, and only fill in the details at call time. It is common in Linux to wrap up these types of interface in a data structure with standard set of pointers to the set of detailed functions - since C has a function prototype, this can be made reasonably type safe (though the programmer is free to mess this up, with disastrous consequences when kernel programming - its a good way to learn some discipline!).

1.4 Protocol Stacks

Since the dawn of time, at least with regard to communications, systems have been composed out of protocols that are *layered* on each other to add more and more functionality.

[Figure 3 about here.]

Each layer implements a service (something relatively simple such as "send or receive a packet), but is internally quite complex (e.g. involves fragmentation, address resolution, etc etc). A higher layer enhances a lower layer. Common elements are kept in as low a layer as possible but only so far as they are needed by the vast majority of higher level users. Otherwise, they are implemented as another higher layer protocol. This is a version of the famous "end-to-end" design principle.

Thus it is very natural to use an object oriented style of programming with a strong separation of interface, defined by its methods and their parameters, from the internal implementation of the actual protocol. This layering leads to the idea of a protocol *stack*. The isolation of layers is enforced to some extent using information hiding (local variables, C `statics` and so on) as well as operating system techniques (separation of address space via memory management).

To counteract this, the real world imposes certain responsibilities on us, such as management, and performance, which sometimes require us to design

protocols that make use directly of internal functions in a lower layer, or report or even modify the state of a lower layer. There are many examples of this including TCP Header compression (see chapter seven), as well as most firewall applications (see chapter ten). This is particularly true in the lower levels such as in driving link devices, and in the Internet Protocol layer and transport. Nevertheless, the “OO” approach works well, and use of controlled “holes” through the interface (global variables or other techniques) is usually kept to a minimum.

Next, let's look at the specifics of some layers, in terms of the service, and the protocol.

1.4.1 The Basics

To date the Internet community has studiously avoided defining its own link protocols, and has endeavoured merrily to operate one level above this, but to port the Internet Protocol software to every new link layer within hours of the technology being available.

The IP protocol is therefore the global glue that keeps the Internet alive.

1.4.2 IP

The conventional way of looking at an internet protocol header is like this:

[Figure 4 about here.]

In Linux, and in C, this is rendered this way:

[Figure 5 about here.]

You may be asking where there is so much in an IP header? We have a version number (4 or 6), the Internet Header Length (in 32 bit words, so value is 5 for IPv4), the Type of service, which we cover in some depth in chapter nine, the total length (includes user data and upper layer protocol headers), the fragmentation information (see later in chapter six), the time to live (see chapter six) the protocol (identifies the next layer protocol up), the header checksum and finally the source and destination addresses (described in chapter three, six and eight). This seems like a lot of data for every packet, but remember that this is a datagram network, so all packets have to carry enough of a survival kit.

1.4.3 Routes

IP is a simple protocol. However, getting IP packets around in a global network is a complex business. The smart part of the job of a router is to compute routes. The easy bit is to send packets on their way. Recently the easy part got harder, because people wanted to control the relative performance that packets are forwarded with for different users or different applications. We look at the two different pieces of routing and forwarding in chapters eight and nine respectively.

For now, let's just introduce the basic idea of how routers work. The task of routing is the task of building a map, and then finding a fast way to index it when presented with a packet's destination and the question: “ok, so you're

holding the map, and here's where we are trying to get to: now do we go left, right or straight on?"

Basically, you need to know how to get to any address - so have all the routers tell each other about all the addresses on the map; then have them build a forwarding table, by searching the map for all addresses, and building an index. When a packet arrives, the destination address in the packet is an index to the map - says : Where to go next (I.e. which turn to take at this intersection).

IP is carefully designed so that destinations make quite efficient indexes. To subvert this, the Internet has evolved so that the map is extremely complex and hard to compute routes over.

IP addresses therefore currently identify points of connection in the Internet. The usual term used is an "interface". Any interface can have one (and even more than one) IP address. IP routing is the function of getting a packet from one interface, across a sequence of other interfaces to a destination interface. This model is somewhat generalised in IPv6 which we discuss in chapter six.

1.4.4 TCP

Similarly, the transport protocol header for TCP is defined in the RFC, and illustrated like this:

[Figure 6 about here.]

This, in C, is defined as follows:

[Figure 7 about here.]

The fields in TCP convey end-to-end functionality, and again, we have to emphasize that each packet must be as stand-alone as possible in an IP network environment, since the routing system may cause packets to arrive misordered, and the transmission system may cause packets to vanish or be corrupted in flight.

TCP is the Transmission Control Protocol, which provides the most common end to end protocol for most applications' reliability requirements.

TCP packets are encapsulated in IP packets, and as far as the IP routing substrate is aware, that is really almost completely transparent (with the exception of fancy forwarding treatment, or filtering as discussed in chapters nine and ten).

So long as IP can get a TCP payload from one host to another, and can get TCP payloads back too, then that's the minimum requirement TCP has from the IP layer.

What's in TCP Header? Well, the first thing that's needed is some way to identify the user or application that is running a TCP connection, so we need some way of distinguishing IP packets carrying TCP payloads. To this end, TCP headers contain 16 bit identifiers called *ports* which are used for both Source and Destination. These enable the IP layer and TCP layer software to do multiplexing, i.e. to support a large number of distinguishable sessions between any host and any other host. Well-known ports in the destination field identify the application, while dynamically allocated ports in the source field allow multiple instances of use of the same application between any pair of hosts.

To get the delivery semantics correct, packets contain sequence numbers so that the TCP code in an end system can construct and ordered delivery of bytes between the sender and the receiving application. Given the network is potentially unreliable, packets also contain acknowledgement numbers which are used to report back to a sender, what has been received correctly in order so far by a recipient TCP. This is a full duplex operation, as is flow and congestion control which is achieved by several complex Window schemes, which are discussed in some detail in chapter six. After the TCP header payload, the rest of the contents of an IP packet will be some amount (for example, on an Ethernet, up to 1500 bytes) of stuff which is data as far as TCP is concerned, but which is some application protocol as far as the upper layer protocol using TCP is concerned.

1.4.5 Packet exchange patterns

In chapter six and elsewhere we look at several packet exchange scenarios. Its worth talking a bit about this way of understanding a protocol here by way of introduction.

TCP has a very complex set of rules by which it operates - RFC793 is 85 pages long, and there are now several other RFCs that define additional functionality for TCP such as congestion control, window scaling, ECN behaviour, Protection Against Wrapped Sequence numbers and so on. Given that a pair of communicating systems are autonomous, it is entirely possible for one end of a TCP session to generate any packet or sequence of packets you can imagine within the header format illustrated above, at just about any time. Many of these will even be legal packets (with sensible sequence and acknowledgement numbers and window sizes and so on). To exhaustively list all the combinations of legal send and receive behaviours is not a sensible approach to thinking about a protocol such as TCP. Instead, we typically use illustrations of *typical* or commonly implemented behaviour, rather than trying to consider the many pathological possibilities.

One part of the way TCP works that makes life a little more complex to consider is that in practice, TCP contains several sub-protocols which operate nearly independently within the same flow of packets being exchanged within a single session. The reason for this is that the data exchange is only a part of what is being sent too and fro. There are also a number of Control Messages. Every TCP packet has all the same fields and has the capability of *piggy-backing* a control function or exchange, along with any data in the rest of the payload. Of course, sometimes, (quite often) TCP packets are pure control, for example the start and end of connections, or in one way data flows where the packets coming the other way are all pure acknowledgements. Even then, the header information is *sub-multiplexing* several functions, such as Ordering, Reliability, Flow Control and Congestion Control, through updating window fields and so on.

We can look at a simple example of how a protocol operates to achieve some degree of reliability:

```
Wait for data
Tx Data(1)
Rx data (1)
```

```

Ack data(1)
wait for ack
wait for ack
Wait for data (2)
get ack, Tx More data(2)
say the data is lost in transit -
what to do? TIMEOUT!!!
Retransmit data (2)

...
...
Rx data (2).

```

Note there are so many permutations of scenario above that there are better ways to express behaviour in a more general manner - one common one is to capture the set of desired actions for the set of feasible events in a single structure called a finite state machine or “FSM”. This is a two dimensional matrix with two indices, the current state, and the current event. The contents of each entry is a function, which carries out the desired action and sets the state variable to the next state. The total number of different paths through such a matrix for a sequence of events can be huge, but it is a compact and robust representation for the entire system, and it can be directly implemented and tested.

Here is an example of simple Flow Control in action:

```

Tx(1);wait for ack(1)
Rx(1);Ack(1)
Tx(2) Tx(3);wait for ack(3)
Rx(2);Ack(2);Rx(3);Ack(3)
TxTxTxTx;wait for ack(7)

etc, etc etc
but if a packet is lost, but if we are ‘out of steam’
we lost a packet,
it doesn’t get acked.
we TIMEOUT,
and rtx - maybe we ought to
slow down

```

In the history of communications, and the Internet specifically, there has been a long-running argument about the fact that the datagram network is connectionless and usually “stateless”. This argument is based on a naive view of layering (some would say a religious view) which holds that “you cannot look outside the packet header of the protocol at *this* layer when processing this layer”. In fact, this is nonsense and as we will see in later chapters, in practice it is very very common to do this for lots of good reasons (TCP and RTP

header compression, header prediction, differentiated services, masquerading, and many, many others). For now, suffice it to say that the common notion that has become accepted is that of a “flow”.

A flow roughly corresponds to a user session. The simplest form this can take is a single web page download, or an ssh session. There are more complex examples of flows that include aggregations of traffic between two machines, or between two sites, or between one machine and a number of sites. All of these are identifiable by some combination of packet header information gleaned from the IP and transport headers - the simplest example is the IPv4 integrated services (or differentiated services) 5-tuple, consisting of the IP protocol number, source and destination IP addresses (host interfaces), and transport source and destination port numbers. More complex examples can be built by either wildcarding some fields, and pattern matching, or by looking at even higher level information.

Flows are a very useful concept, and we will revisit them in later chapters (especially in chapter eight). For now, we can think of them as the unit of resource control (whether for allocating resource, or blocking for security reasons). Linux has powerful mechanisms for identifying flows efficiently.

There is some debate in the Internet measurement community about how easy it is in practice to use flow information (statistics about the pattern of end points, and distribution of duration for flows). The debate is important for large network operators in provisioning their networks, and is a topic for several other books!

1.4.6 Performance trends

Networks get faster and bigger, as well as smaller and slower, year by year.

The Host count on Internet increases by around 100% every eight months, and any number we write down (e.g. as I type this there are now around 50M public, and probably 200M private networked computers) will be out of date before you finish reading this sentence!

A typical link speed was 56kbps in 1976, but now 622Mbps is not uncommon wide area, and people working on designing terabit/second routers.

[Figure 8 about here.]

Specific examples of national institutional network access in the UK National University network, SuperJANET, were 64kbps in 1980, 2Mbps in 1990, 155Mbps in 1995 1Gbps in 2001, and currently deploying 2-10Gbps, to be rolled out by 2002. However, note that Moore’s Law has computer speeds tracking this performance quite nicely. Not only that, but commodity PCs are also reducing in price, so that there is a win-win scenario with using them as a networking end system, router or middle box.

1.5 Security

Security concerns discussed above briefly can be expanded on a little, although, as we said, it really is important to be well educated about cyber-crime-prevention!

Typical questions to ask yourself before hooking up a machine to the net include:

- Who can use your computer?
- Who can use your network?
- Whose computer can you use?
- Whose network can you use?
- Access control, privacy, depend on protocols:
- Authentication protocols plus encryption techniques...

This of course still leaves denial of service and traffic pattern analysis. The best defence against these may be to invoke a different model of costs, such as performance management through pricing. We'll look at this next.

1.6 Performance

As the network becomes a more important component in your enterprise, you may have concerns about controlling the absolute, or relative performance that different users and applications can obtain. There is now a plethora of techniques for doing this, and Linux implements most of them very well in its most recent versions, and is a favourite amongst engineers looking at this area.

1.6.1 Integrated Services

The most stringent performance requirements can be controlled through meeting the IETF's Integrated Services standards. These specify three QoS service-levels. Current the service is best-effort, but this definition adds the controlled-load service (RFC2211), and the guaranteed service (RFC2212) and other services possible (RFC2215, RFC2216). There is a complex signalling protocol called RSVP ("Resource Reservation Protocol) (RFC2205, RFC2210).

The standards describe the service semantics which specify how packets with a given service should be treated by network elements along the path. This is achieved by setting values for a general set of parameters `|service_name|`, `|parameter_name|`, both in the range [1, 254] for the traffic specification, and the service specification known as TSpec (the allowed traffic pattern) and RSpec (the service request specification).

Performance is in terms of a token bucket (rate, bucket-size) and is controlled through a token bucket filter which measures the total data sent. To make sure we do not add traffic to a network that is already full, the routers (and/or sources and receivers can do this too) running this scheme carry out a calculation called "Admission Control".

While running this scheme, traffic is also subject to policing. This means that conforming routers check that the TSpec is adhered to and packet handling may change if the TSpec is violated (e.g. degrade service-level, drop, mark, etc.).

The idea of the controlled load service is to provide a virtual network which looks very much like a path for Best-effort, but in under unloaded conditions.

The idea of the guaranteed service is to offer a specified assured data rate with bounded-delay. This is a deterministic guarantee, but offers no guarantees on jitter.

1.6.2 Differentiated Services

Integrated service is expensive to offer in router state (processing and memory) terms, so a less complex model is defined called Differentiated Services.

Again, an architecture demands a service model (RFC2475), and this entails tiered service-levels, indicated now through simple packet markings (RFC2474), rather than via out-of-band (RSVP) signalling. Packets are intended to be marked by network, not by the application, so differentiated services would support legacy applications. It is intended to be simpler to implement than INTSERV, and can be introduced onto current networks.

It should scale better since state is no longer kept (necessarily) per-flow, and the aggregate treatment of packets from a source is defined.

Service classes that have been defined so far include:

- Premium (low delay) - EF (RFC2598)
- Assured (high data rate, low loss) - AF (RFC2597)

Ongoing research is defining mechanisms to specify the Service level agreement (SLA) via a service level specification (SLS). This will include a statement of the policy between a user and a provider, with policing at ingress, and a service provided by network (end-system unaware).

Packet marking is achieved through use of the IPv4 ToS byte or IPv6 traffic-class byte, now known as the DS byte. Traffic classifiers that work to mark packets according to the SLA, act on either a multi-field (MF) (as in integrated services) DS byte + other header fields, or a behaviour aggregate (BA), recursively re-using the DS field only. DS codepoints define values for the DS byte, and SLAs are implemented by understanding the Aggregate per-hop behaviour (PHB) together with the aggregate treatment within network.

So typically, we would specify the rate and delay in the SLS. Expedited Forwarding (EF) (RFC2598) defines a virtual leased line (VLL) service, with data rate specified in SLS and more loosely defined performance for the other parameters, such as low delay, low jitter, low loss.

Assured Forwarding (AF) (RFC2597) defines 4 classes (1-4) with 3 levels of drop precedence per class (1-3), where AF11 is best and AF43 is worst.

Traffic conditioners act at the ingress to the network to meter traffic and apply a marker and if appropriate, a shaper/dropper. Metering of traffic is the function of deciding whether a flow (aggregate) is in-profile or out-of profile. Re-marking can be used as the flow(s) traverse multiple providers with different implementations of the PHBs, to a new DS codepoint, and to re-shape or drop packets.

[Figure 9 about here.]

When is allocation made to SLA? This can be at subscription, per user/user-group/site/customer, and could be multi-field, policy-based, or it could be within organisation, per application/user/user-group.

We could apply it using ad hoc tools or network management system, and the behaviour aggregate or multi-field possible, or dynamically, using RSVP.

Since the same DS codepoints could be used for different services by different providers, different providers using the same PHBs may have different behaviour we have to define edge-to-edge semantics. There is also the problem currently

of the lack of symmetry: protocols such as TCP (ideally) require symmetric QoS. Finally, differentiated services for multicast, to support for multi-party, symmetric communication, are as yet undefined.

[Table 1 about here.]

1.7 Summary

In this chapter, we have sketched an overview of the Internet protocol suite. The interesting thing about the design philosophy embodied in the split of functions between the unreliable IP layer, and the reliable TCP layer¹ is that it fits the evolution of technology so well. Link and physical layer systems have continued to evolve that are affordable with tremendous performance and low cost (both for Local Area networks and for long haul links and switches) so long as we don't expect them to mask faults at the bit or packet time scale. Instead, end-to-end protocols such as TCP have been devised and implemented in all common operating systems for the end user to be offered the required communications semantics. The split has also led to twenty years of optimisation techniques familiar to software programmers being applied to the IP routing and forwarding layer, and to the end-to-end protocols, so that now on a commodity price computer, we can have near to gigabit per second performance without even using significant about of CPU or memory resources.

In the next chapter, we take a tour around the Linux operating system looking at its general features, before coming back in the rest of the book to look at TCP and IP and all their friends and family.

¹if we can characterise what may also have been an astronomical quantity of blind luck, as “design”!

Chapter 2

Introduction to the Linux Operating System

echo '16i[q]sa[ln0=aln100%Pln100/snlbx]sbA0D4D465452snlqxq'|dc

Submitted by Simon Chapman to the Irish Linux community web site.

2.1 Roadmap

In this chapter, we review the purpose and functions of an operating system and put the networking mechanisms in this context. We look at the overall features of Linux. Then we study a day in the life of a process, a file, the operating system as a whole (from boot to shutdown), a device driver a kernel module, and a system call.

We explain the difference between user space programming, and the kernel view of the world and take a detailed look at support for concurrency, and the consequence for control for access to shared resources, with the idea of critical regions, and mechanisms for preventing inconsistency such as interrupt disabling, semaphores and spinlocks.

We then follow the story of a day in the life of a file, covering the variety of file types and file access which make up the rich world of Linux input and output, and the unification of process, device and module management.

Following that, we take a look at the day in the life of an operating system, from boot til shutdown.

Similarly, we follow the day in the life of a device driver, looking at interfaces (the view from above - unifying all devices) and down to hardware and interrupts and DMA.

Because of its special status in network code, we dissect a day in the life of a kernel module, and show how this gives Linux the properties of a micro-kernels but without the overhead of server or helper processes with their associated context and IPC overheads.

Getting down to more detail, we examine the day in the life of a system call, and getting from one space to another.

After that, we take a look at the operating systems idea of time and events.

We wrap up the chapter with a look at multi-processor support, and in particular, the effect of Symmetric Multi-Processing (SMP) design on other parts of the operating system and networking implementation.

2.2 What is an Operating System?

An operating system (OS) provides *services*. Typical services that a user wants such as management of storage through a file system and backup system; the ability to run application programs; easy access to the network; management of a Graphical User Interface.

There are several different approaches to designing and building such software. At its simplest, much of an operating system is a collection of libraries of commonly used functions. An OS usually also incorporates a number of useful utility programs (operations and management applications). However, what makes an operating system different from other collections of code is that much of the OS code interfaces between the hardware and applications.

Services are provided through a set of layers of software which unify common elements and hide the idiosyncratic details of the underlying hardware. Access to hardware requires use of specific instructions and memory addresses that the system may wish to protect the normal user from accidentally disturbing. Hardware may autonomously wakeup and attempt to access memory or interrupt the processor at any moment - this is especially true of user interface hardware (mice, keyboards) and networking interfaces. While user programs (applications) are increasingly being implemented by programmers who are familiar with event-driven programming paradigm, it is still hard to design systems that manage large numbers of competing streams of events - gone are the days when co-operative multi-tasking was a viable way for the every day program to manage resources on a system. Now this role is largely taken by the *kernel* of the operating system. Applications are largely insulated from this level of complexity.

Different operating systems have been designed with different types of user in mind. This means that the three general purpose operating systems in most commonly in use today in the world, Unix, Windows and MACOS, have undergone very different pressures during their development. This has led to quite different emphasis in how services are provided.

The principle differences lie in whether the system was originally designed to support a single or multiple users; whether the user is a technical expert or someone with no interest in developing computing system further; whether the user has intense multimedia requirements, or mainly text and graphics. MAC OS and Windows historically both targeted single user systems, while the Apple offering emphasised multimedia. Unix systems have always targeted the technical *systems* developer, engineer or scientist. As such, Unix has traditionally ran on large computing systems as well as small desktop ones. Because there were many different vendors providing different processors and systems for scientific and engineering based computing, Unix systems have always been designed to enjoy the ability to run on a variety of hardware platforms.

An analogy for the different services is to think about a car; in a car, you have an operating system that the normal user thinks about - navigation tools, the accelerator, the brake, the steering wheel and so on; you have the operating

system that the maintenance engineer thinks about: the tire pressure, the engine management system, the fuel supply and so on; there is the operating system that the mechanic is concerned with: the steering, the suspension, the braking system and so on. Each has their place. In some system designs, there is more effort on the user side, while in others (off road vehicles!) there is more on the engineering. To some extent, recent work Linux is trying to cross this divide and cover both territories, by extensive work on application and GUI support. Fundamentally, the kernel of an operating system is rooted in the efficient exploitation of the hardware to meet one of these goals.

To make efficient use of an expensive resource such as an old mainframe, mini-computer, or supercomputer, the Unix designers adopted a number of the techniques from the commercial large operating systems of the 1970s. Over the last decade, there has been a clear trend to adopt these same ideas in desktop systems, but at the same time, Unix like systems such as Linux have been designed to work in just the same type of constrained environments. In particular, Unix-like systems are the industry standard for network servers, systems that run application-level services accessible over the network, which is one of the reasons for this book!

Networks are another example of a potentially expensive or at least shared resource. As described in the introduction, they are typically shared between many users. This means that an operating system support for network access, and access from the network to operating system resources, must follow the rules for sharing the network.

Linux provides a general set of means for computing resources to be shared between multiple users at the same, and at different times. Linux unifies different hardware under a resource management umbrella, to allow safe and efficient use of the processor, memory, and input and output devices.

2.2.1 Multi-user

Since the 1970s, there has been a saying, that “Everything in Unix is a file”. What is meant by this is that the standard abstraction for any object on the system is provided by the overloading of the concept of a file.

Of course, normally, files are objects stored on disk. Files are organised in file systems. Any file system (and Linux supports quite a few different ones) consists of a way of organising storage devices (“file-stores”), and a way of organising files within a naming hierarchy (directories, files within directories and so on), and a way of formatting data within files (Unix has traditionally kept this as simple as can be). Unix *unifies* the view of a number of objects in the system by using the file name space to name, and therefore provide access to) devices and (in modern Unix systems) to processes as well as to simple stored data. This unification makes the system programmers’ tasks easier - there are fewer system interfaces than in other families of operating systems. We will see throughout the book how this makes access to the communications facilities quite straightforward. We will also see that in some cases, this unification comes at a cost - sometimes the analogy between a file and some other type of object stretches the use of the system call interface in some rather odd ways, and this shows in the use of some of the parameters!

An important aspect of the file systems is the identification of ownership of files. In the data structures associated with any file, whether it is a normal

plain file, or a directory, or a handle on a device or even a process, there are fields that show the ownership of the file, and the permissions associated with different users (user, group, others). This data is stored with the files on disk, and is kept in memory when a file is opened by a program - the data structure that stores this all is called an *inode*. At the root of all things, all files have an “inode number”.

Traditional interaction between the user and the system was via command line interfaces (the “shell”). Of course, on modern systems, we expect to use a Graphical User Interface (GUI). The underlying system for this on unix is X-Windows, which for 20 years now has provided networked graphical interfaces to programs running on remote systems. GUI’s come into their own when coupled with multi-tasking systems - it is common for many classes of user to combine and overlap a number of loosely related tasks (e.g. editing and previewing a letter, or programming, debugging and viewing documentation, or browsing the net and taking notes). A windowing system combined with a tasking system is essential to the modern user. Lets look a bit further at tasks.

2.2.2 Multi-tasking

High street shops now sell computers that execute nearly 1000 million instructions per second. Yet much of the time, the user is sitting in front of their computer (or standing holding their palm-top) thinking deep thoughts about what to type next. In the meantime, the computer might as well get on with something useful. In the early days of computing, people would hardwire in to the operating system some useful task in the “idle loop” such as computing the next few million decimal places in

$$\pi$$

. Nowadays, we do the same thing more flexibly - for example, screen savers may download some data from the radio-telescopes around the world and try to find some meaningful signal in the noise (“Search for Extra-Terrestrial Intelligence”-SETI). Multi-tasking is the ability of the operating system to provide the illusion of serving many masters at once. The illusion is easiest to see as a user when you fire up some background task such as printing. The two tasks, a foreground one (e.g. an editor) and a background one (e.g. printing) are carried out by the system by alternating between the programs, running part of one, then part of the other, and so on. Programs that have started and not finished are known as “processes”. All processes have a process id.

A system that has enough cycles per second to fool a user in this way, and enough memory to hold the code and data for multiple programs is relatively simple. The key idea is that a process can be described by taking a snapshot of the state of the processor and the code and data at any instant (between one instruction and the next) - this snapshot is known as the “context”. The operating system can save the context of a running program into some special location in memory associated with the process, and ‘switch’ to another process by restoring its context - really, this is just a case of loading all the registers with the last values that this other process had, and setting the CPU’s program counter to the next address to start getting code from! The part of the operating system that carries out this “context switching” is called the scheduler. When there are several processes to choose from, the scheduler may make use of all

sorts of book keeping data to base the decision on. Linux supports several different schedulers nowadays. This can be very useful for specialist applications such as real-time systems (e.g. in physical process control systems in factories where time is of the essence).

Multi-tasking is really essential for providing a clean programming abstraction for an open platform where users can safely run different applications in a windowing environment. Multi-tasking is also non-optional when building network servers. Nowadays, a server (e.g. a Web Server) may receive many requests nearly simultaneously, and will need to schedule dealing with servers in many different ways. There are several different ways that this can be done, but a multi-tasking system is really one of the simplest. We will see later on in this book that there are reasons why general multi-tasking is sometimes a little too slow for really high-throughput web serving.

General purpose servers are very common in the networking business -these may provide a number of networked services such as network time (via Network Time Protocol) to synchronise the clocks on a set of remote computers, network printing, E-mail, logging, file serving and so on. The general term used for programs that provide useful services on a Unix system is “Daemon”.

Processes are associated with users, just as files are. In Unix, processes are created in an ancestor-descendant way from a primordial process (init, a process that Unix starts when it finishes booting - see later!). child processes inherit their parents user identity. To change identity, a process may set its user id, although only to reduce the access rights that a child may obtain. This sounds draconian, but is in fact, quite general.

Linux also keeps track of time. This is fairly essential for a number of networking protocols, as well as being important for book-keeping (e.g. file creation, access and modification times, process startup, and system usage statistics).

There is a lot more context associated with a task that Linux manages in order to keep life simpler for the application programmer.

For example, the file system provides an hierarchical organisation of directories. This is useful for users and applications as a navigation aid. To help keep things brief, the **current working directory** is part of the state associated with a process. In keeping with the simple (but effective) approach to security, Unix allows a process to change its view of where the root of the file system is. This allows parents to restrict children’s’ view of things!

Processes start life with three standard channels to the outside world:- input, output and errors. These are associated with the *controlling terminal* or *tty*. Parents pass these through to their children (of course a process is at liberty to “detach” from a controlling terminal so that it can run “in background”). A group of processes controlled by the same terminal are managed together. The idea of groups of processes is very useful in networking, since one can manage the state of the group together, particularly through the use of *signals*.

Signals provide a way for the operating system to notify a process of an event. This idea originates from the requirement for raising exceptions (software interrupts) in a task: for an extreme example, many unix systems can send a signal to all processes when the power is about to die! Nowadays, signals are used for many asynchronous communications, along side more normal file, pipe and socket communication tools, of which we will see more next.

The kernel and user processes may use signals to communicate (albeit abruptly) with another process. If a process wishes to communicate in a more civilised

manner, with another process on the same machine, there are several choices. If the programmer wants a record of the communication, the simplest technique is just to use files. Unix doesn't restrict the reading and writing of files by default to a specific process, but there are tools to control access to files and provide synchronisation. A more common technique is the use of a pair of file like channels, called a *pipe*. Finally, the programmer might use full-fledged communications channel such as a *socket*, which can be used for purely local, or inter-machine inter-process communication.

To summarise, we can take view from below as the operating system context, or the view from above, the application/user context. The viewpoints are like a dual personality or the sub-conscious and conscious.¹

2.2.3 Multi-device

Modern computers support a massive range of devices. As well as traditional keyboards, screens and storage devices, we have a range of network interface types, including fixed and wireless, and a number of multimedia devices (audio, video capture, 3D, etc).

At the lowest level, the operating system needs to manage input and output from and to these devices. There is a surprising amount of commonality in the functions required to match the OS to the different hardware.

A large percentage of the code associated with any operating system is so called "device driver" code. Device drivers hide the specifics of a piece of hardware to control an i/o device, and provide the standard systems and application programming interfaces to the device, translating between the generic model, and any idiosyncrasies that a specific device may possess. Of course, a particular hardware configuration only needs drivers for the devices it possesses. Linux supports the concept of *modules*, which allows on demand dynamic loading of code for operating system sub-components such as device drivers, which makes it even more economic on memory.

All unix systems identify the kind of device driver by a unique number called the *major number* and the specific device by its *minor number*. Major numbers are allocated by an authority, although free ones can be used by configuration or as needed, while minor numbers are allocated as needed by the system or by configuration (e.g. through *mknod*).

Linux divides devices into three basic classes:

character these devices are typically serial lines (traditionally for keyboard type input) but also parallel interfaces, mice, sound cards, frame buffers, and so on. The main emphasis here is on interaction with humans and external machines.

block these are for storage type devices (floppy, hard and optical disks)). Most importantly, these support file systems.

network this a special case where Linux differs from other Unix-like systems in its treatment of these as a distinct device type, and as networking is the main topic of this book, this is discussed in detail later. The critical

¹A more fanciful model of system calls might be the visual metaphor used in Jean Cocteau's film of the Orpheus and Eurydice legend when the poet follows his true love into hell but going through mirrors.

different between character and block devices, and network devices is that the network device usually has another intelligent computer at the far end, rather than a human user, or just dumb storage.

What makes devices similar? The ability to read and write data to a device is the main common strand. What makes devices different? A number of things ranging from performance (data rate, latency etc), through to intelligence (dumb keyboards, smart audio and network cards etc), and most importantly management and control complexity.

As discussed before, operating system services are layered. Applications typically access files, and Linux supports various file systems which reside within kernel space, but make use of block devices to access storage devices. Applications may carry out inter-process communication, via networking protocols, which (largely) reside in the operating system, but make use of network device drivers to access local and wide area networks. The exception to this 3-layered pattern is that applications wishing to carry out interaction with users via keyboard and screen generally do so directly. The case of the graphical application using a frame buffer for output and mouse for input is slightly exceptional here, as this is generally done via the X-Window system, which resides in user space, but requires a special process to manage the frame buffer display and pointer devices, rather than using a kernel resident task to do this.

We've already mentioned that the file system naming is also used to name devices. The naming convention for network devices is somewhat different and is discussed later. Now we introduce the idea of *binding* of names. "Binding" is a word full of resonance: from the binding power of certain kinds of magic in the Lord of the Rings through to the ancient art of book binding. The use of the word "binding" in computer science is no less important.

In the context of I/O, binding is concerned with associating one or both ends of a channel, which is identified by some short descriptor, with some *named* entity. In the file system name space, we need to make a storage device that contains a file system visible in the name hierarchy. This is done through "mounting" the device, which attaches the root of the file system of the device into the overall hierarchy. In the driver names space, a driver is associated with a specific device through the major plus minor device numbers. In the process names space, processes are associated with a process identifier through the `/proc` mechanism (more later!). In the communications world, we associate a channel with a pair of end points through some Internet naming and addressing system, of which more later! In the windowing system, an external server associated regions of the screen with window identifiers.

At the lower level, device drivers may make use of a number of different programming techniques for access between the device to move data to and from the processor or memory. The simplest technique is called "polling", and depends on continual access to a device to see if it has data ready (or it is ready for data). Few devices work like this in real systems. More commonly, the device has a modicum of intelligence and can make use of *interrupts* to notify the processor that data is ready. Different processor families have different hardware techniques for providing interrupt handling facilities. Typically, most systems provide some number of channels, on the PC known as Interrupt ReQuests or IRQs. There may be more devices than IRQs in which case the devices have to share interrupts, and a processor has to be programmed to distinguish which

device interrupted. Each IRQ is associated with an interrupt service routine (ISR). Typically, the operating system installs code at a set of addresses which is where the interrupts are “vectored” to. Think of an interrupt as an “unsolicited” procedure call, with a few side effects. These side effects are important (they involve changing processor mode from user to kernel for example). We will see later how they work, and how we synchronise between processes, kernel tasks and interrupt handlers. Many devices also have enough intelligence to move data to and fro from memory to device controller buffers and vice versa - this is Direct Memory Access (DMA) and is typically setup by device drivers to take more work away from the CPU, and to get some measure of parallelism between normal work and I/O work. Most faster (block mode and network) devices work this way.

Linux has one more trick up its sleeve with regard to interrupt handling. Interrupt service routines often have a critical time within which they must respond to a device, at least initially. However, there may be a lot of housekeeping work which can then be done with more slack. Linux ISRs schedule the later work to be dealt with by what is known as a “bottom half”. The bottom half routines often involve moving data up to application space and rescheduling processes and so on. A list of bottom half routines is scanned, and any ready routines are called explicitly during the last stages of system calls. In Symmetric Multi-Processor Linux, bottom halves have been re-engineered using a more general mechanism which we’ll look at briefly later on.

The fact that an interrupt changes processor mode (typically from user to kernel) and therefore allows code in an ISR to access privileged resources, makes it the idea way to implement System Calls. System calls have also been known as traps for this reason, that they are like a trapdoor down from user space into the cellar that is the kernel. Most unix implementations vector all the system calls through one IRQ, and then use a jump table to access the actual system call code. System calls rigorously check the validity of parameters (e.g. data buffer addresses, and lengths thereof, for read and write must reside within the user’s process’ address space!). System calls may de-schedule a user process if necessary.

2.2.4 From input/output to Interprocess Communication

“Everything is a file” is an oft-repeated Unix mantra. In fact, there are several activities supported by operating systems that can be made to look like files, but hide a multitude of sins of commission and omission behind this simple faced.

For example, there are at least four radically different things that can be accessed by the same system calls, but are not all files.

File A File is an array of bytes on a storage device. In the Unix view of the world, we can open, close, read and write it. There’s a huge amount of code in Linux dedicated to supporting files on a range of different storage devices with a variety of different goals, including foreign operating system support and performance goals.

Pipe A pipe is a means of communicating between two related processes. It looks like a file, but we can’t do some things t a pipe that we could do to a file (e.g. seek).

Socket One view of the concept of a socket is the extension of a pipe between two processes to include two more steps: firstly, communication between unrelated processes; secondly, communication between processes on different CPUs or different computer systems. Sockets are supported by a large amount of communications protocol code, in the analogous way to the support for files on many different file systems.

Device A device is a piece of hardware. Some things a device can do are support input and output, but don't actually perform it directly - for example, a device might tell you its temperature. Devices may be for storage, for communication, or for something completely different (robots, car engine management systems, etc).

The things in common between these are simple. The aspects that differ are complex. In common is that all unix communication is via file system interface code, and starts with getting a handle on a channel via a system call such as `open()`, using a name for the end point (file, device, computer/process). Subsequent communication might use `read()` and `write()`, or other calls, but there are then other things that the programmer might wish to do, including polling, or controlling some facet of the communications channel. We'll see later on that in one case (communications routing protocols) this is so complex that Linux ends up supporting a control channel as another file i/o channel (the routing `rtnetlink` interface) rather than *overloading* yet further one of the overburdened system calls such as `setsockopt()` or `ioctl()`.

2.2.5 Processor Independence

The core components of most modern operating systems are reasonably independent of processor architecture. This is due simply to modern software engineering practice of using high level languages and of separation of concerns for services and hardware.

However, Linux appears to enjoy a degree of portability across processor architectures beyond many other systems (apart from its Unix cousins), due perhaps partly to the efforts of its implementors in retaining a clear goal of including-in any other processor architecture work on the operating system. Another factor in this has historically been the relatively small size of the Linux kernel source, which means that identifying architecture dependence is a smaller task than with some commercial systems. This is becoming less true as more variations (e.g. multiprocessor support) mean that there is more complexity in the kernel which may depend on hardware architectural differences.

However, the vast bulk of the hardware specific code is in the device support, which is where things are necessarily processor, bus and I/O architecture dependent in any case, so this is not of any concern.

There are some parts of the low level memory management, time management, and scheduler which have some assembler and a few C code routines which need attention when moving to another hardware platform.

Types

Linux defines a set of C types for use in the kernel (and elsewhere if you like) to aid in remembering what is and what is not portable. These are built on top

of the POSIX standard definitions, and architecture specific types.

[Figure 10 about here.]

[Figure 11 about here.]

[Figure 12 about here.]

[Figure 13 about here.]

[Figure 14 about here.]

2.2.6 Concurrency Control

A very important part of an operating system is that it has to manage a set of co-operating tasks. In a modern system these include I/O processors (at the very least, DMA capable devices, but often complex disk controllers with internal schedulers), but possibly even multiple processors running different parts of the operating system concurrently. In any case, it is conventional to provide the illusion of concurrency to the application programmer, so that different application processes can run apparently sequentially, using kernel services in a way that appears to be independent, but is in fact interleaved. This makes many types of application programming much simpler, while making efficient use of computing resources at the same time. For example, a word processor is simply a program that appears to control all the input and output from the user typing at a keyboard and using a mouse, and to the screen, and all the input and output from and to files, in a sequential way. In fact, the computer may be carrying out a whole host of other tasks simultaneously, both on behalf of this specific user (e.g. printing, backing up etc) and for other users. Indeed, breaking down the details of device and file system access into smaller steps reveals that not all of these are sequential either. What happens on a single processor is that the illusion of a single *virtual machine* is provided by a scheduler which can switch context between different user processes (and within the kernel, a dispatcher which can switch between different interrupt contexts). This is known as *interleaving*. When a computer has multiple processors (even a single processor, but with smart independent I/O such as DMA, which we look at more later), then the concurrency can become actual rather than illusory, without modifying all the applications or large pieces of kernel code, thus leading to potential (at best, near linear in CPU count) improvements in performance for the overall set of tasks (though not necessarily for a single process) as you add more processors to a system.

When execution is interleaved in an arbitrary way for strands of code that access the same data, there is a need to control access to the data (to “serialise”) the code at certain points to maintain data consistency. It is common to hear kernel hackers refer to *race conditions* which are situations when concurrent pieces of code leave data in an inconsistent state, in some situations (but not in all). It is hard to debug such errors since they occur in only some combinations of interleaved execution (some out of very many possible ones) and heavily depend on timing. Even the introduction of simple tracing or debugging code may make them vanish. Sections of potentially concurrent code that access the same data are known as *critical regions* and require separation. As well as this,

we also need low level communication between processes via global variables to be specifically synchronised.

To this end, Linux provides a wide range of primitives. These depend on whether one is controlling concurrency for a user process or for the kernel.

At the highest level of the operating system (synchronising with user processes), the User Process Scheduling Level needs a `sleep_on` (interruptible) and a wakeup function.

User processes (when a CPU is in user mode) can be interrupted at any time (this is expected) and one device interrupting can interrupt itself or another device's interrupt service routine. In kernel mode, interrupts do not occur inside a routine at the same priority or lower. At the lowest level, the basic requirement to support this is for atomic operations (e.g. a single "test and set" instruction that is uninterruptable, at least on a single processor architecture). Different hardware architectures provide slightly different atomic instructions, so Linux provides a set of macros to hide this.

For kernel critical regions, we can divide the functions into two approaches - the first simply disable interrupts, rendering code serial; the second provide synchronisation.

Kernel level locks work on the interrupt level, and can be separate for reading, and writing functions. The "spin" versions work across multiple processors in an SMP environment, and employs simple busy waiting. To maximise concurrency, separate read and write locks are provided. `cli` and `sti` are "clear" and "set" interrupt respectively:

[Table 2 about here.]

In addition to this, the kernel has wait queues, which we can then put a thread on (add wait queue/remove wait queue) and kernel level semaphores to provide ordered access to a region: The classic `down` (`down_interruptable`) and `up` functions operate on a semaphore.

Linux has a mechanism for deferred interrupts. These allow hardware interrupt service routines to be as short as possible (which can maximise the rate at which they can be handled) and puts off handling the bulk of the work (usually to do with higher level functionality) to later, using what it calls a *bottom half* level:

- `start_bh_atomic()` = `atomic_inc (&global bh_lock)`
- `end_bh_atomic()` = `atomic_dec (&global bh_lock)`
- `bh_mask` is set and cleared by `enable_bh(n)`, `disable_bh(n)` for nth bh
- `do_bottom_half()` checks that there is no lock (no global `bh_lock`) and no interrupt handlers (global irq count null), and that interrupts are enabled (irq-lock is free)

In SMP, (in Linux 2.4) we have *software interrupts* which are interrupts are SMP able (i.e. non serialized). On top of these, have a serialized task mechanism called a *tasklet* - these are how bottom halves are implemented, as serialized software interrupts. The old deferred interrupt code such as bottom halves, is run by one of the new tasklets. Spin locks actually do busy waiting between CPUs, but then this is no longer a problem (doesn't lead to priority

inversion for example). A nice way to think of a spin lock is to say “what I want to access is locked now, but to wait will require a context switch, which is slow and expensive. So I’ll spin around for a while in the hope that the lock unlocks soon, rather than context switch immediately.” This might add say 25% of the unlock not occurring, but this might only happen 1 in 10 times, so the benefit is obvious

[Table 3 about here.]

[Table 4 about here.]

Note Bene - cause for confusion (term also used for *signal* and for *system call trap*).

So we have a whole plethora of techniques at our disposal which have different levels of applicability.

2.2.7 Memory Management

The Linux kernel model of memory is (typically) as simple as can be: Linux uses *linear* addresses, which are a flat 32 bit address space of 4Gbytes, rather than making segmentation visible. However, this is the model. In practice we cannot allocate a real 4G to the kernel and to each process, so we need to use paging to manage mapping from linear addresses to real address space. Also, some hardware architectures (e.g. Intel) oblige one to use at least a limited form of segmented addresses.

Segmented addresses are *logical* addresses made out of two distinct components: the *base* and *offset*. When the CPU (or some devices) access memory, addresses are translated from those used in the code into physical addresses by one or two stages. If we just use paging, then we just need to use the linear address to access the page table. If we are also using segmentation, then the segment descriptor needs to be accessed: this is done on the Intel platform by implicit access via a non-programmable register which contains the segment descriptor. Typically, on such hardware, segments are used merely to separate four segments: kernel and user code and data. The rest of memory management (and all of it on other platforms) is done via page table hardware. Depending on the exact nature of the page table, some number of the high (i.e. most significant) bits of a linear address (the one the code references) are used to index a page table. Typically, since most page tables are not fully occupied, we use a combination of caches and translation look-aside buffers to get space/performance tradeoffs. A complete discussion of segmentation and paging can be found in most good operating systems books (see [7] for an excellent discussion).

Linux processes (as with most operating systems) are built out of statically allocated memory at load time, for the text (code), data and stack. However they can allocate memory dynamically as well.

The kernel is also built out of a statically allocated set of memory (at boot/startup time) for the kernel code and data, and dynamic memory. However, the kernel has to manage memory more dynamically, for two reasons. Firstly we have processes being loaded and started and stopped all the time. Secondly, the requirement for process memory may exceed the actual physical memory in a machine.

To this end, Linux implements a virtual memory system (like many operating systems) which extends the limits of physical memory by using disk space. To organise the memory systems into a coherent framework, the unit of allocation and mapping is the page. At the lowest level, this is typically managed by memory management, mapping, and paging hardware, which pre-determines a limited set of page sizes (for efficiency reasons). The most common size is 4K bytes. The full set of descriptors of possible pages are in memory themselves and are represented by the page table, which is an array of descriptors with an associated set of flags indicating how the page is being used. Efficiency considerations are to do with the tradeoff between the page table size, and the cost of *paging* from disk to memory and back, and the accuracy of matching the unit of allocation to the unit of request for memory. Of course, actual usage doesn't map simply onto convenient 4K page chunks of memory.

Some page use puts constraints on the memory addresses (e.g. devices that cannot go via the MMU and/or can only access low parts of the address space) and this is specified in the flags on request.

At the next level of abstraction, many kernel functions require variable length pieces of memory allocated and freed dynamically which are often much shorter than a whole page. Linux implements a system called a *slab* allocation algorithm which is based on the idea that there are a common set of objects of a common set of sizes that are frequently allocated and freed (e.g. process descriptors, file table entries etc etc). The slab allocator groups requests into caches of similar sizes. Another important feature of memory allocated from slabs is that it can be aligned to a multiple of bytes or words or even an entire page size, so that accesses can be faster and even avoid the address mapping process.

Memory management is quite a complex process in general, and has to be tuned often to the mix of jobs a system typically runs. In practice, it is the pattern of memory freeing that can dominate the performance.

In the communications code, most memory allocation is done by a special purpose function, and uses its own set of data structures to overlay the memory with some patterns that are highly optimized to network packet header processing. This is the *socket buffer* structure, and the `alloc_skb()` and related functions. These in turn call `kmalloc()` (and `kfree()`) to allocate general purpose dynamic memory for general purpose use. Typically, these calls `kmem_cache_alloc()` to allocate memory from the appropriate cache block size. The use of these functions for networking is discussed in more detail in chapter five.

The exception to this is the use of lower level access to memory allocation typically for i/o buffers in device drivers when we need to restrict the possible locations that memory is allocated. This is often because devices do not access memory via the page table or the segment register or have restricted numbers of bits in the device's register that is used to specific buffers' locations.

2.3 What more could you ask for?

As well as normal multi-user, multi-tasking Unix (as you would expect), Linux 2 supports:

Platform Independence several processor architectures (e.g. Intel, arm, alpha, Motorola, and lots of exotics)

Multi-processor Symmetric Multiprocessor operation

Sophisticated Scheduling multiple user process schedulers (two classes of real-time scheduler and some smarter than standard time slicing and also run to completion as well as normal unix semantics)

Efficient Execution Startup copy on write demand load executables (i.e. fork uses no memory, exec uses no memory (mmaps the file, but doesn't demand page until a new data page is needed)

Efficient Application Memory Utilisation shared libraries (so processes are smaller as well as faster)

Efficient System Memory Utilisation modules (run time loadable kernel code - e.g. most drivers and most alternate schedulers, and file system code is not present until needed)

Sophisticated File Systems several alternate file systems (e.g. Unix, NFS, EFT, DOS, NTFS and SAMBA)

Binary Compatibility Multiple execution formats

Industry Desktop Compatibility WINE (Windows emulator)

Internationalisation National Language support and other normal stuff...

Applications include some fairly convincing Microsoft clones, plus several desk tops that make X less clunky (and more MAC/Windows like - including shortcuts including URL/Web like ones).

Reliability, Efficiency, Safety, Security

Another factor in Linux success is the robustness of the system. In general, networks and operating systems are robust because of two main factors: simplicity and feedback. The end-to-end model[9] describes a system design principle for networks (and OSs) that prescribes inclusion of functions at the “lower” layers, only if they are *generally* required in the layers above. In a network protocol architecture, this is as clear as it is in an operating system. Devices do not concern themselves with applications, just as link layer protocols do not contain features that serve merely one application or transport or network layer protocol.

An emergent feature of the end-to-end model that is common across operating systems and network systems is that considerations of statefulness (maintaining history across a sequence of invocations of a service) is not a useful feature in a low level function. This is true in Linux of device drivers and of file systems, and network protocol stacks at the lower layers too.

This leads to remarkable lifetime and safety, as well as reasonable transparency when there are failures.²

It means that application writers (and this applies beautifully to network server implementation patterns) are free to choose the performance/reliability tradeoffs they desire.

²When I first built and installed a Linux 2.4.0 kernel under a Red Hat 7.1 distribution, it ran 24*7 until I wanted to do an upgrade - for 2 months.

Because of the *generic* design of its layers, Linux will also run in 4M of memory (but wont run very far or fast). This is done simply by leaving out what you don't want. With modules, this happens automatically, for free, although you can do this statically at kernel configure time if you prefer.. On a 486, with 16M it will run X (KDE, Gnome etc) and be usable. On a pentium 2 with 56M of memory it is really quite impressive. This means that it is evidentially a scalable system, which is very interesting if you want to run a range of systems such as desktop, small server, router, and large cluster.

Linux is extremely good at resource sharing. In networking, we tend to refer to this as multiplexing. Multiplexing is a simple idea - it requires the system to keep track of resource shares. This is a large part of what network and transport protocol state machines are for. It is also a large part of an operating system functionality. User and process and I/O management are all about keeping track of ownership and utilisation of resources. Schedulers and security subsystems carry out this work.

Of course, there are some differences between intra-system resource management and inter-system problems. For example, systems separated by a network are autonomous. This means that there are potentially multiple administrative domains: Each operating system has a set of users and superusers and so on. Each system owns a set of resources and can separately and unilaterally allocate them. In network terms, this translates into policies which can be local or global, link wide, network wide, or worldwide. To make these work in a scalable fashion is a large problem, so typically, we only very loosely couple the local and global management if at all.

2.4 The Source Code Organisation

The reference site for the Linux kernel is <http://www.kernel.org/> . For the purpose of stability for this book, (which is absolutely *not* for building up to date kernels!) we have mirror of the 2.4.0 code at

<ftp://cs.ucl.ac.uk/darpa/linux/> .

The Linux source code is the subject of the Gnu Public License copyright statement. This is part of a philosophy of Open Source. Now is not the time or place here to discuss the details of this. The GPL is in the file COPYING.

The directory structure of the source tree is important to understand. It is quite simple! Assuming we are at the top of the source code tree (wherever you put it in your file systems), it then looks like this:

[Table 5 about here.]

Of these, this book is mostly about the code in the one main `net/` sub-directory, and the network specific sub-directories in the `include/` and `driver/` sub-directories.

[Table 6 about here.]

Going one stage further down the tree, lets look at the IPv4 directory for example:

[Table 7 about here.]

2.5 A day in the life of a Process

A Process starts life as a compiled program sitting on a disk and a user somewhere with a desire to run the program. By interacting with the system (via some other program, e.g. a shell, or perhaps via cron), the user informs the system that they want this program to run.

A process lives in a memory address space of its own, made out of its own set of pages and runs with the CPU in “user mode”. A process can extend its memory (using the `sbrk` system call via `malloc()`) and can share memory (see chapter four on IPC). It can also memory map files.

Processes live in a family tree, and new processes are spawned through `fork()` and various forms of the `exec()` system calls copying the process table entry, creating a new process id, and then overlaying or allocating new memory for a new executable, and loading it from disk. In some cases we clone many copies of a process (e.g. for web serving in very old CERN HTTP daemons!). Nowadays, this is more commonly done through a threads system.

Processes make use of kernel services to share resources (as discussed before) such as cpu, memory, disk, files, networks etc.

In their extremely useful book[5], Beck et al., describe the two different ways that we can look at the relationship between a process and the kernel. To paraphrase what they say:

Process Viewpoint From “above”, the kernel provides services to processes.

Processes access these services by making “System Calls”, which appear to be calls to library functions, that carry out some work and return. Processes feel like they are on their own - they appear to have a system dedicated to them. When a user program enters a system call, the kernel takes over and has more privileges to access more of the system - because of this, parameters to system calls are extremely constrained, well defined and carefully checked.

Kernel Viewpoint From “below”, the kernel runs the system. The kernel can access all memory, all devices, all additional hardware (kernel registers, MMU etc). The kernel is structured as a set of cooperating tasks, but there are no protection domains between these tasks and the switching between them is entirely cooperative. Programming mistakes are disastrous (hence why systems programmers are paid more than application programmers!). When the kernel runs a user’s code, it runs it in a less privileged mode to make sure that user programming errors do not “infect” it! We can see this separation of concerns illustrated in diagram 15.

[Figure 15 about here.]

[Figure 16 about here.]

[Figure 17 about here.]

[Figure 18 about here.]

[Figure 19 about here.]

[Figure 20 about here.]

[Figure 21 about here.]

[Figure 22 about here.]

[Figure 23 about here.]

[Figure 24 about here.]

[Figure 25 about here.]

[Figure 26 about here.]

XXX need example of bits of actual entry and return with re-schedule or not

At the end of the day, when a process is done, it exits. Typically, there may be various pieces of tidying up to do so the well behaved parent process will wait for the kernel to finish closing all files and so on before continuing. An example of a network task that needs finishing up, is a TCP connection needs shutting down properly even if a process rudely exits (e.g. due to crashing because of a bug). This is a good example of why transport protocols such as TCP (see chapter six) are in the kernel.

2.6 A day in the life of a File

Files in Linux come in a variety of types. A discussion of the virtual file-system is beyond the scope of this book (see [5] again, or [7], for an excellent overview). Suffice it to say that we have already discussed the use of the name space to manage a set of different operating system resources including devices as well as disk storage. On disk storage, so-called *regular* files are accessed via the standard system calls, and basically consist of a 1D array of bytes. Other structures can be built above this as applications need it. Standard techniques are used to improve read and write access via caching and read-ahead (and write behind on some systems).

There is also the possibility of memory mapping a file, which means that subsequent random access will be far more efficient (unless thrashing in the paging system is incurred!).

Files reside in a file-system, which is created when a disk is formatted and partitioned. Each partition is laid out with a doubly linked list of *superblocks* and ordinary blocks which can be used for file information or file contents. The superblock contains information about the file system type (which indicates the allowed operations on a file system). Typically, operations read, update or delete blocks or *inodes*. The `inode` structure is the core data structure which holds all the file attributes such as ownership, access times and block location information. An in-core copy is held whenever a file is open, as well as several blocks of the actual file, and also contains information about any memory mapping of the file.

Files are created by allocating an inode and some blocks.

In addition to regular file systems, there are also networked file systems (e.g. accessed via NFS or Samba) and virtual file-systems. Of the latter, the most important is the `/proc` file system. The kernel maintains a bunch of information

about active processes and uses the access procedures in the `/proc` file system to allow applications to manage this data. This is generally useful for access to quite arbitrary kernel data without having to allow direct access across the user/kernel memory address spaces.

2.7 A day in the life of an operating system

It has become commonplace for users to leave their computers running all the time. However, one does have to start them up when you bring them home from the shop, and from time to time, one has to shut them down (even if only to throw away), though usually for upgrading.

2.7.1 The start of time

When you power up your machine, the hardware generates a special signal which *should* propagate to all devices, memory and processor to cause a *power up reset*. This sets everything into a *known state*, so that when software starts, it can make certain assumptions about how things are.

From this point on, the details depend on the processor architecture, but in practice, all systems arrange for the processor (or at least one of them) to start its instruction fetch-and-execute cycle at some *well-known* address.

Normally, we organise the memory hardware so that this address (a physical address since the memory management unit is not active yet) actually is a piece of ROM (read only memory) which holds a standard well known program, which allows some very basic operations to be done including some rudimentary input and output. On a PC this is the BIOS (Basic I/O System). Normally, this will run some self-tests on the hardware, then initiate a “real” bootstrap process.

This consists of looking on some storage device (such as a floppy disk, or hard drive) for a bootable operating system - usually stored in a well known place (the *boot sector* of the disk).

In fact, as systems have become larger over the ages, what is normally stored in the boot sector is a second stage bootstrap, or *Boot Loader* program. If you want to boot multiple systems, and manage this via Linux, then this program is the Linux Loader program, *LILO*.

The loader needs to be able to load arbitrary code onto arbitrary addresses, so normally it copies itself to high memory, and then sets up the addressing modes so that it can load code all over memory. It then starts to read in the configured (or selected) operating system from the (selected or configured) boot device.

Finally, the boot loader changes some of the systems settings and jumps into the newly loaded code, which goes through the operating systems specific initialization:

Here we go...

[Figure 27 about here.]

[Figure 28 about here.]

[Figure 29 about here.]

2.7.2 As time goes by

Once the system is up, it runs the `idle()` process and awaits some work to do.

2.7.3 The end of time

At the end of the day, when you want to shut down the system, the appropriate signal is sent to all the running processes, and the system synchronizes the file systems, and halts.

2.8 A day in the life of a device driver

A device is a generic object with a set of standard routines. It is initialised, then there are a set of standard functions for input and output, as well as interrupt handlers.

Here we look at the three main types of devices, and how they make themselves known to the operating system when the drivers come on-stream.

[Figure 30 about here.]

[Figure 31 about here.]

[Figure 32 about here.]

[Figure 33 about here.]

[Figure 34 about here.]

[Figure 35 about here.]

[Figure 36 about here.]

[Figure 37 about here.]

[Figure 38 about here.]

[Figure 39 about here.]

2.9 A day in the life of a kernel module

One of the critically acclaimed features of Linux today is that it achieves the flexibility of a micro-kernel without the need for server processes with the inevitable overhead of domain-crossing inter-process communication. This is done through modules. This is illustrated in diagram 40.

[Figure 40 about here.]

Modules are simply run-time loadable code that is linked into the kernel. This is achieved through a relatively simple interface which works by defining a very generic set of routines to initialize and cleanup a module, and to export an interface with a set of symbols (variables, functions etc) and module cross-dependencies. Dealing with cross dependencies is not rocket science - it is what

link loaders do every day when you compile and link an application which uses several functions from several different libraries. What happens is that kernel code is annotated via explicit listing of the symbols that are exported, then some utilities are used to build an index of the symbols (see `/proc/ksyms`, and the module management tools use these to resolve what is and isn't needed.

There are some applications for managing modules which live in `/sbin`, such as `insmod`, `rmmmod`, and `modprobe`. The first two can be used manually (or via scripts) to add or delete a module. The latter is actually invoked by the kernel (so the kernel that uses modules requires a file system with at least `/sbin` so that it can run this external program!), to manage the module dependencies on demand.

An example of the moist adventurous use of modules is the MIT Click Project [26] which implements a component based router completely as a collection of modules, configured through a programming language, and then installed via the kernel module mechanism.

2.10 SMP

Linux supports Symmetric Multi-Processing. What this means is that if you have a multi-processor system, the Linux kernel will be executed on behalf of processes on more than one processor. This is not rocket science either, but it does make the inter-locks between critical regions potentially a little more complex. It also makes performance analysis of a system (particularly a large file, web or router server type system) much more complex. This is a topic for an entire other book!

A large part of Linux 2.4 changes to the networking code from previous releases is associated with what is known as *softnet*. This is the change to remove the bottleneck associated with a lot of networking code being run from bottom halves in the deferred interrupt model, and instead to run it from software interrupts. There are two soft interrupts allocated to networking, the `NET_RX_SOFTIRQ` and `NET_TX_SOFTIRQ`. These trigger the appropriate handler. Since this is a different interrupt from the actual hardware interrupts, it can be concurrent with hardware interrupts (and with other kernel and user processing) on a multiprocessor platform. The second part of reducing the bottleneck due to serialisation was to change the various kernel locks to use spins. Finally, instead of scheduling a network bottom half handler, we schedule a network thread by calling `netif_wake_queue()`

This is defined in `include/linux/netdevice.h`. Essentially, `netif_start_queue()`, and `netif_stop_queue()` manage a given interface by tagging it as active or not, and waking up the queue schedules an interrupt, as shown in the code extract below 41.

[Figure 41 about here.]

2.10.1 Application Support

This book is not about applications per se. Suffice it to say that support for applications extends beyond the actual operating system into application level libraries, as well as threads and inter-process communication, and a number of toolkits and GUIs.

2.11 A Day in the Life of Time

A notion of time, in the sense of ordering events, but also in terms of an absolute clock and the concepts of *duration* and *interval* are very important in operating systems, and especially with regard to communications and distributed systems. Most operating systems in use today on the desktop, and for network server based computers, do not provide the user with a direct *real time* support. However, within the lower levels of the kernel, and right down to the hardware level, there is usually some support at least for measuring intervals, if not for actually *scheduling* events on exact boundaries. There is also usually some support for scheduling events at approximately some later time (after *at least* some duration).

The way this works in detail at the lowest level depends rather on the exact hardware support in a given processor and system architecture.

However, at the next level up, Linux has consistent support for managing dynamic timers in the kernel, and offers a standard Unix *interval timer* mechanism to the application programmer.

RTC Real Time Clock Starting at the lowest level then, lets assume that the hardware provides a real-time clock, which ticks at some relatively low frequency (compared to the processor clock for fetching and executing instructions at least!), for example a few tens or hundreds of Hertz (cycles per second). More importantly, this is battery backed up, so that it moves forward even when the system is deprived of power, which means that it is a stable store of state and is useful for long term protocols (see later in TCP Initial Sequence Number selection for example) across reboots.

TSC Time Stamp Counter Secondly there is some type of clock input to the processor which is used to cycle through the fetch and execute cycle - this is on the order of hundreds of MHz, or even several GHz nowadays. This typically increments a particular register or location (e.g the Time Stamp Counter register in the 386 family), which can be read from software in the system.

PIT Programmable Interval Timer Thirdly, there is usually some provision for a programmable device that can interrupt after some interval has elapsed, on the order of milliseconds accuracy, up to seconds away - usually, this is programmed to provide a periodic “clock interrupt” every HZ where this is defined in the kernel (e.g. 100 times per second). The term *jiffies* refers to the number of ticks since the system booted, and counts the number of clock interrupts.

The clock interrupt handler deals with a hardware exception, and thus must not take too much time to do its work (or else there’s not enough time to use the CPU for any real computing - worse still, if we take too long we might even miss a clock interrupt, although we could tell later on by reading the TSC, but by then it might be too late to take any actual actions that had been intended). Instead, clock interrupt actions are divided, as with other interrupts, into two types: *urgent*, and *deferred*.

The main urgent task is simply to record the system uptime. A number of other tasks can be handed by one of two bottom half (TIMER_BH and TQUEUE_BH)

(or, in 2.4 by a tasklet associated with these) and include: making a record of how long the current process(es) have been running and other resource utilisation levels, checking if any interval timers for any tasks have expired, and calling the appropriate call-back functions if so.

The main relevant kernel service then to other kernel functions is that one can install, delete and alter timer functions, which are essentially lists of a simple data structure, defined in `include/linux/timer.h`. These define after how many jiffies a given function is to be called. `kernel/timer.c` defines some simple functions to create a list entry, initialise it, and load and set the function pointer and expiry time for the timer.

[Figure 42 about here.]

[Figure 43 about here.]

[Figure 44 about here.]

[Figure 45 about here.]

From user space, interval timers are setup with `setitimer()` and `alarm()` system calls, and result in a `SIGIO` event.

2.12 Summary

In this chapter, we have presented an overview of the Linux operating system, pointing out various salient features of process, user and I/O management, but not paying a lot of regard to network services yet.

In the next chapter, we take a vertical cut through the whole protocol stack to see how it all fits together, and in the form of a rapid tour through the topics of the rest of the book.

Chapter 3

The Brief Life of a Packet

“It is better to give than to receive.” - traditional saying.

“Be conservative in what you send, and liberal in what you receive.” - Jon Postel.

3.1 Roadmap

In this chapter we take a lightweight structured walk-through of the process of constructing and sending a packet in Linux, and then receiving and decoding the packet at the far side. We do this by way of three different examples, each seen from three different viewpoints: firstly we look at the transfer of some data over TCP connection (e.g. part of an HTTP down-load); then we look at a UDP packet exchange (e.g. part of a DNS lookup); finally, we look at a one way RTP flow. The viewpoints are taken from the source code; from the execution; and from the “wire”. In the last case, we also consider a router as part of the wire, and go into a bit more detail about its operation: looking at an extended case which entails capturing some audio, putting it into an RTP packet, calling the socket API to send a packet, going through the socket kernel code (skbuffs etc), adding the UDP header, firewall filter checking, and multicast. Then the packet traverses a link (SLIP or PPP say) to a router which is also Linux, running diffserv - so we get a look at a packet being scheduled in packet queueing code, by a WRR or an other scheduler, then forwarded, say, over an Ethernet, then finally received by driver, IP layer, demultiplexed and passed up through UDP through to the process which is scheduled, reads the socket and outputs the audio.

This is a combination of the way to explain Internet communications ideas from many RFCs and related Internet documents¹, where the view of single protocol is expanded through an operational example *and* a top down view of the entire protocol stack in one go.

Thus we also provide a rapid introduction to the rest of the book at a fast-track technical level.

¹see for example, TCP/IP Illustrated Volume 1.

3.1.1 User, Application and Protocol Viewpoints

There are at least three very different views of communication that one can take:

User from a user perspective, in modern systems, communication is made as transparent as possible. when clicking on an anchor in a web page, there is no indication whether the download is local, or very remote. Many FTP implementations include GUIs that make copying files the same whether remote or local - usually the only visible differences are in the areas of naming (specifying a remote system name as part of a URL or FTP site name, although this can often be hidden in a global extension to a normal object or file naming system; and security - access to a remote system usually (though not within an organisation) entails presenting more credentials than when accessing local resources.

Application An application is programmed to talk to an Application Programming Interface to a lower level system. Normally, such an API hides detail and, especially in the area of data communication, this detail may be of a very complex and rich structure - if the API is to a lower level protocol, then it will hide the concurrency and state machine present in the lower level and attempt to present a more sequential, simpler style of programming for everyday implementors to understand and use.

Protocol The protocol perspective is the most complex. here we need to understand the way that information is exchanged. Here we need to consider the fact that each end point in a communication session is autonomous, and has its own quirks and failures and successes. The channels between end points are imperfect, introducing errors and independent failures and performance degradation.

Each of these perspectives can be illustrated in different ways. In the user perspective, one is interested in the resource naming, and visible performance parameters. In the application perspective, one is interested in the APIs and interface programming. From the protocol perspective, one really wants to understand the range of sequences of possible events and the way that state is changed by events.

From a user perspective, we can show information models and organisation of how resources are mapped into cyberspace. From an API perspective, we can document the interfaces - the function prototypes, pre-requisites and results of functions and how they relate in families to each other are usually part of good systems documentation in any case. However, as we move through the layers of abstraction to lower levels, often these APIs become more complex and the interactions more subtle. From a protocol perspective, we need to understand the behaviour of senders and receivers and the structure of information exchanged.

This latter view is often best illustrated through looking at the operational examples of a protocol, showing the actual data exchanged on the wire for correct, and common error cases. in this sense, understanding a protocol stack implementation is often best done by effectively *debugging* it.

In the Linux community, the use of high level tools for debugging the kernel is somewhat frowned upon. To some extent, this is justifiable since most high level

debugging tools (e.g. GDB) introduce changes to the function call mechanisms and therefore actually alter timing which in low level code (especially interrupt handling, but also anything that involves any fancy concurrency and scheduling) will alter the behaviour of the system under examination, often critically to the point where the problem being chased is no longer evident!

However, one can conduct the same type of activity ‘by hand’. Thus in this chapter we start doing what the rest of this book does in detail, which is to take a structured walk through o the Linux kernel communications stack. In doing so we occasionally take a look at a trace of packets on the wire, and occasionally take a look at a trace of function calls, but largely, especially given space and time considerations, we concentrate on the code itself. This is because there is really one main example piece of code, while there are an infinite variety of examples of execution traces of the code.

Many tracing, debugging and profiling tools do abound, and I would encourage you to use them to help find problems (especially non intrusive tracing for performance problems) then all well and good. For now, lets look at some of the code!

Sending

In the diagram below⁴⁶, we try to illustrate the overall structure of the code, showing the way the control flows from the application down to the driver, and thence out onto the transmission medium.

[Figure 46 about here.]

So here we can see the flow of information from the application, which uses system calls from the file system API and Socket API such as `send`, `sendto`, `sendmsg`, `writetv`, `write` to interface to the lower level protocols in the kernel. The parameters to these system calls are passed through what is normally a synchronous, blocking, procedure call to the lower level, and actually makes things like reading and writing to a remote application look extremely similar to reading and writing from a local file or application (in chapter four, we look in detail at Inter-process communication APIs. The parameters are typically data and addressing information, and these are passed through to the transport protocol in the correct family, for example UDP, TCP, etc

The kernel then triggers the appropriate state machine for the protocol and then passes information through to the IP level which then entails doing the necessary route work, and then adds any appropriate IP header information, finally queueing the data for driver output. The device is then usually woken and actual transmission occurs. In fact, several of these stages may be deferred to allow more efficient use of CPU, memory, and network resources, so that a network scheduler is then part of the picture. In fact there are several complex levels of scheduling which are not always explicit. Sometimes, it is simply a matter of using a flow control and on-off scheduler mechanism, for instance when looking at device output for busy links (or device input for full queues) we may rely simply on semaphores or other mechanisms to handshake between the layers, and wake up device driver, which does link and physical layer work to send or receive a packet.

Receiving

In the diagram here 47 we try to show the converse process - from unsolicited arrival of a frame from the wire, up to delivery to an application that had, in fact, been earlier primed with the idea that it might want to receive some data sometime soon!

[Figure 47 about here.]

Physical reception causes hardware to wakeup, and typically interrupt. Most modern network devices include both interrupt and DMA hardware. The interrupt service routine pulls packet off device and puts in memory and flags the "network bottom half" that packet(s) ready. Bottom halves and other Linux mechanism (software interrupts, and tasklets and other SMP capabilities) are discussed in chapter two and later on in the book.

The general system scheduler (after system call) runs network bottom half (and can dispatch tasklets or software interrupts) on completion of any kernel work on behalf of any user process that called a system call. The bottom halves basically complete work that the hardware interrupts started- for example, they can dequeues any packets in the interface queue for IP input , and sends any more packets waiting to send.

IP input checks packets, carries out any re-assembly work needed, get a route if the packet is to be forwarded rather than being destined for a local application process. If it is for local delivery, IP then demultiplexes the packets according to the IP protocol number to the appropriate transport handler receive function (UDP, TCP etc).

TCP (and UDP) input then do further packet header checking, update state machinery, and then demultiplex the packet further to the correct socket (i.e to the queue of data pending for any user process that has an outstanding read on that socket, which is "connected" to the appropriate port numbers.

Once the transport protocol demultiplexing routine has put data on the appropriate socket receive queue, it calls wakeup on any process waiting for data, thus making any such process ready for general running by the system scheduler,

Sometimes a process that had earlier called a `recv`, `recvfrom`, `recvmsg` `read`/`readv` and was asleep - data is copied from receive queue to user space buffer; process is now woken up, and continues with data in buffers.

Recent Linux work on avoiding the receive packet copies (copy buffer only on write) has improved this performance somewhat.

Forwarding

The final case of a packet handling pattern we need to worry about is that of forwarding. Forwarding is (almost) entirely kernel bound - with the exception of the process of managing the tables used to make forwarding decisions (see chapter eight) and performance (see chapter nine) the forwarding task for IP entails back to back, reception and transmission.

3.1.2 Source Code, Execution and Wire Viewpoints

In the rest of this chapter then we look at the source code and trace through some simple vertical cuts across the stack for some examples, with some very

slimmed down view of what occurs on the wire during these executions.

3.1.3 State, Memory etc

Throughout the code, protocols need to keep track of where they are. Rather than just using a set of randomly allocated messy global memory, most protocols gather together all the appropriate information on a given flow (at their level - e.g. an IP route, a TCP connection, a UDP session) in one data structure - in BSD Unix this is generally referred to as the Protocol Control Block. In Linux, the general all purpose data structure of `skbuffs` is also used to hold this data. It contains information to fit a packet into a sequence, to multiplex packets downwards (locally route them) and to de-multiplex them when receiving and passing them upwards (e.g. link to the user socket structure to queue data and so on) as well as very complex state machinery for parts of the TCP machine.

3.2 TCP Example

First we look at the example of the Transmission Control Protocol and follow through the code applied to a single data packet as it is transmitted, then subsequently as one is received.

3.2.1 Socket Level

In this simple example, we assume that an application process (netscape, apache, etc) has called `write()` on socket which maps into a system call (see chapter two), which subsequently calls `tcp_sendmsg()` and and deschedules the process (see also chapter two).

3.2.2 TCP Output

Here we at the socket glue, then the tcp work - in the source distribution this (as is most of the rest of the code discussed in this chapter) resides in the `linux/net/ipv4/` directory in the (version 2.4) kernel distribution tree.

An extract from the code referenced below is displayed in figure 48.

```
inet_sendmsg()
sk->prot->sendmsg() ...
tcp_ipv4.c:
struct proto tcp_prot {}
tcp.c:
tcp_sendmsg()
tcp_output.c:
tcp_send\_skb()
tcp_transmit_skb()
err = tp->af_specific->queue_xmit(skb);
which maps to ip_queue_xmit()
```

Then in `tcp_transmit_skb()`, the state/header changes are managed. This is discussed in detail in chapter seven.

[Figure 48 about here.]

3.2.3 IP Output Work

TCP calls IP functions to carry out output. Some of this work is structured as follows, and a relevant extract from the code is illustrated below in figure 49.

```
ip_output.c:
ip_queue_xmit()
ip_route_output()
NF_HOOK()
```

`ip_route_output()` is an inline function defined in `linux/include/net/route.h` and interfaces to `ip_route_output_key()`, which is explained in chapter eight. `NF_HOOK` is part of the network filter support in Linux which is machinery to do the low level work for firewalls and other functions and is discussed in more detail in chapter ten.

```
netfilter.h:
```

```
NF_HOOK(pf, hook, skb, indev, outdev, okfn)
NF_HOOK_slow((pf), (hook), (skb), (indev), (outdev), (okfn))
ip_queue_xmit2
skb->dst->output(skb)
```

This queues, or calls `okfn()`, which maps to `op_queue_xmit2()`. The `skb->dst` structure was filled in by local route lookup and is link level output routine.

In `ip_queue_xmit`, header changes are as follows:

[Figure 49 about here.]

This is covered in detail in chapter six.

3.2.4 Link Level Output

Output at the link level can be complex (see chapter eight). It includes specific technical details of driving devices, but also the important part is that this is where Linux provides different treatment for different traffic types by providing fancy queuing and scheduling management.

In general, we call the device specific function to start output on the device. This does the book work, then informs the device that there is work to do, which then typically is done by DMA - this is covered in some detail in chapter five.

3.2.5 Link Level Input

Link level input requires dealing with hardware and software interrupts and DMA devices - this is covered also in chapter five in detail. Some relevant extracts from the code are displayed in figures 50, 52 and 53

[Figure 50 about here.]

[Figure 51 about here.]

Network device interrupts end up calling the function `netif_rx()` via the `net_devices` structure, in `linux/net/core/dev.c`.

[Figure 52 about here.]

[Figure 53 about here.]

Then the software interrupt schedules the rest. which eventually results in a call to

Next, `net_rx_action()` which dequeues the packet, and calls a packet handler based on the packet type (via a table) - if IP, this calls `ip_rcv()` in `ip_input.c` - which checks various things.

3.2.6 IP Input Work

IP input has to decipher the packet, so the first thing to do is figure out if it is really “for me”., e.g. is it IP version 4!

In `ip_input` `ip_rcv()` checks, then calls `NH_HOOK` (to run filters, which may call, or queues directly `ip_rcv_finish()`, which calls `ip_route_input()` in `route.c` which fills in the “internal” route.

This includes a function (forward or local deliver), and calls `ip_local_deliver()` in `ip_input` again, and this then calls `ipprot->handler`.

This pointer was previously filled in (via the `inet_protocol` structure) in `protocol.c`, which matches the protocol to the protocol handler function - in this case, TCP - `tcp_v4_rcv()`.

This is investigated in more detail in chapter six.

3.2.7 TCP Input Work

In the IPv4 case, the relevant TCP input code is kicked off from `tcp_ipv4.c`, which looks up if we have a connection `_tcp_v4_lookup`, then tries to queue in process context, or else, calls `tcp_v4_do_rcv()` then `tcp_rcv_established()` which actually processes the TCP header, (header prediction etc) and calls `tcp_event_data_rcv()`, does any acknowledgement needed and calls `sk->data_ready`. The last function queues data to the socket.

3.2.8 Socket Input Work

In `core/sock.c` we see that the main effect is triggered because

```
sk->data_ready = sock_def_readable;
```

[Figure 54 about here.]

3.2.9 On the wire

```
16:09:07.462590 brahms.cs.ucl.ac.uk.ssh > ovavu.cs.ucl.ac.uk.1023: P
2466:2574(108) ack 971 win 17376 <nop,nop,timestamp 7115105 382435932>
(DF)
```

3.3 DNS/UDP Example

Here we look at the difference if the packet was a UDP packet, for example part of a DNS exchange. To a large extent, these are restricted to the actual UDP protocol function itself.

3.3.1 UDP Output

In `udp.c`, the main function of interest is `udp_sendmsg()`, for output.

[Figure 55 about here.]

[Figure 56 about here.]

[Figure 57 about here.]

3.3.2 UDP Input

UDP is small enough that it is all in one C file, `udp.c`. For receive, the function of interest is `udp_recvmmsg()`

[Figure 58 about here.]

[Figure 59 about here.]

3.3.3 On the wire

```
16:11:35.321380 brahms.cs.ucl.ac.uk.1904 > bells.cs.ucl.ac.uk.domain:
60803+ (43)
16:11:35.323169 bells.cs.ucl.ac.uk.domain > brahms.cs.ucl.ac.uk.1904:
60803* 1/2/2 (172)
```

```
14:19:51.488399 0:20:af:ab:e1:6e 8:0:20:7d:a5:36 ip 85:
brahms.cs.ucl.ac.uk.2167 > bells.cs.ucl.ac.uk.domain: 6517+ (43)
14:19:51.490923 8:0:20:7d:a5:36 0:20:af:ab:e1:6e ip 214:
bells.cs.ucl.ac.uk.domain > brahms.cs.ucl.ac.uk.2167: 6517* 1/2/2
(172)
```

```
14:22:05.123790 0:20:af:ab:e1:6e Broadcast arp 42: arp who-has
merci.cs.ucl.ac.uk tell brahms.cs.ucl.ac.uk
```

3.4 RTP/UDP (multicast) Example

For multicast, most higher level protocols are built on top of UDP, so again we need to look in `udp.c`.

Sending is the same as for unicast, but the receive case has to handle the chance that there is more than one process waiting for copies of a multicast packet, so receiving is a tad different. as we can see in `udp_v4_mcast_deliver()`

[Figure 60 about here.]

3.4.1 On the wire

```
16:25:49.747290 d230-17.uoregon.edu.1025 > 224.2.163.188.23824: udp 332
16:25:49.790001 hocus.cs.ucl.ac.uk > 224.1.127.255: igmp nreport 224.1.127.255 [ttl 1]
16:25:49.829479 d230-17.uoregon.edu.1025 > 224.2.163.188.23824: udp 332
16:25:49.853864 d230-17.uoregon.edu.1025 > 224.2.163.188.23824: udp 332
16:25:49.898744 d230-17.uoregon.edu.1025 > 224.2.163.188.23824: udp 332
16:25:49.992876 d230-17.uoregon.edu.1025 > 224.2.163.188.23824: udp 332
16:25:50.104227 d230-17.uoregon.edu.1025 > 224.2.163.188.23824: udp 332
16:25:50.135776 d230-17.uoregon.edu.1025 > 224.2.163.188.23824: udp 332
16:25:50.212101 d230-17.uoregon.edu.1025 > 224.2.163.188.23824: udp 332
16:25:50.272045 d230-17.uoregon.edu.1025 > 224.2.163.188.23824: udp 332
16:25:50.276215 Ez.Stanford.EDU.33235 > 224.2.163.188.23825: udp 88
```

3.4.2 ...but with a router in the path...

We know where we are going, but not how to get there - some kind strangers along the way will help - these are routers. See chapters eight and nine for more details Linux as a router.

Linux works well as a router. Basically, you need to consider input, then output - the decision above, where a packet was discovered to be destined “for me”, in the call in `route.c` validates the source address for the input device, then decides if its for me, multicast, loopback or for someone else and adds it to a hash based cache table of most recent routes

Then if we are forwarding, we do `fib_lookup()`. This will in the routing structure, including a reference to the input handler - if its a non local delivery, this will be set to `ip_forward()` in `ip_forward.c`:

`ip_forward` does various checks (router alerts, route options etc) `ttl` processing, `mtu` work, fragmenting, generating ICMP errors if necessary, NATs, if needed, then calls `NF_HOOK` (for net filter work, again see chapter 10). and then this will schedule a call to `ip_forward_finish`, which checks the route cache for a fast route decision, and then does `ip_send(skb)`.

Then `ip_send()` does fragmentation or not, and calls `ip_finish_output()` does yet more NF work, then calls `dst->neighbour->output(skb)`; and/or `hh->hh_output(skb)`; (hardware header cache stuff)

3.5 Three Views of the Traces of Ping

As an illustration of the actual total API, including another protocol case, that of a `raw` socket, used to do ICMP access, lets look at the trace of the ping program, usage: `ping bells`, as shown in figure 61:

[Figure 61 about here.]

On the wire, a ping is an exchange of ICMP echo messages, and might look as shown in figure 62:

[Figure 62 about here.]

Finally, from a systems viewpoint, we can see the set of calls made by the application by running *strace* on the ping command, and getting a listing of all the system calls it makes, as shown in the excerpt in figure 63. In this trace, you can also see the stage of looking up the IP address of the destination since the ping command (as above) had been given a DNS name - on the system under consideration, it is possible to configure this as a local NIS lookup, a lookup in `/etc/hosts`, or a DNS lookup, with a sequence of stages of what to do next if one type of lookup fails.

[Figure 63 about here.]

Chapter 4

Inter-Process Communication

“Socket Scientist” - Job description seen at an IETF meeting.

4.1 Roadmap

In this chapter we look at the programming interfaces provided for Inter-process communication (IPC). As luck would have it, the de-facto standard for application programmers is a truly cross platform capability, generically referred to as *sockets*, and even extends to Microsoft’s Windows systems, although it originates with the BBN and Berkeley Unix family, dating from the early 1980s.

There are other IPC mechanisms that operate within a single system, which we do not cover in depth as they do not rely on the TCP/IP protocols. These are generally simpler to use, and also may (in some cases) be more efficient provided that the programmer only wants to achieve *local* communication between processes that are known to only reside on the same computer (for all time!).

These other mechanisms include:

Pipes Pipes are the oldest mechanism in the Unix family of operating systems, but are restricted to processes that are directly related in the creation tree. Pipes are analogous to a temporary file, that has no actual representation in the mounted filestore (neither disk space nor in the file name space). Linux pipes are *pairs* of file descriptors, one of which is used for reading, the other for writing, in each direction. As with any Linux files, a pipe is a 1D array of bytes. However, note that you can’t *seek* on a pipe. This would be tantamount to time travel (like reading a file before it is written)! A pipe is implemented in the kernel as an inode which has a virtual file system type (see chapter two) that allows `read()` and `write()` calls and internally keeps a simple buffer, with appropriate locks, to implement the channel between the two (or possibly more) processes.

Named Pipes or FIFOs FIFOs get around the limitation of pipes that a pipe has no *explicit* name, and therefore cannot be accessed except by processes related in the ancestor/descendant tree of the process that created the pipe, and therefore have a copy of the file descriptor(s) for the pipe.

Instead, FIFOs have a representation in the file system *name space*, although they do not occupy any actual file store disk blocks. Thus the VFS functions for opening and creating a FIFO are different than for a standard pipe, and have to include a reference to the filename. In fact, a named pipe or FIFO is created the same way that a device file is created, via a `mknod()` call.

Shared Memory Linux includes the so-called *System V IPC* mechanisms, which derive from the AT&T Unix System development tree. Shared memory segments are one of three different IPC resources which can be allocated by a process and then used to communicate with other arbitrary processes. The resources also include the semaphore and message types as below. Shared memory is very efficient in terms of preventing the data copying that is sometimes associated with crossing process-process memory protection domain boundaries, although Linux's copy-on-write semantics probably means that sockets are nearly as efficient. One benefit of the System V mechanisms is that they cleanly separate synchronisation (achieved via semaphores, messages and bulk memory usage. The shared memory mechanisms also include access right mechanisms modeled directly on the file system access right and process ownership structure of {owner, group, other}, and {read, write} permissions.

Semaphores Semaphores are used to place a set of processes in an ordered execution sequence so that consistency of shared memory segments can be maintained. They provide user processes using shared memory with the same functionality as kernel semaphores do for kernel threads. Obviously, executing a semaphore operation typically suspends the calling process until the relevant semaphore is clear.

Messages The System V messaging facility allows for sending, and receiving, fifo queued messages.

In the rest of this chapter, we concentrate on an overview of the Socket API and its implementation, and cover Management and Control and the life of a socket.

4.2 Socket API

We discussed in chapter two in general why the interface for application programmers to kernel services is not implemented simply as a subroutine library. This is especially true of the API to the transport level which isolates the networked application programmer from network layer errors, and from the complexity of dealing with error recovery and timing matters, as well as enforcing the correct handling of independent failure modes at each end of an end-to-end session.

The kernel needs to control access to shared (a.k.a. *multiplexed*) resources such as network devices. The transport and network and link level protocols need access all of memory (mmu/dma etc) The kernel decides on scheduling for a share of resources. Thus, we restrict the less sophisticated programmer from access to these complex responsibilities via system calls. System calls are traps or exceptions or s/w interrupts use a special instruction to change processor

mode. Once in kernel space, we can decide what to do (or not, e.g. simple syscall like `getpid()`) There is not necessarily a one-to-one mapping between the set of function calls that implement the socket API library, and system calls. In different Unix versions, it has depended on history whether this is the case, or whether the calls map through a single call, or a smaller set of actual system calls. This is also affected by the ability of different hardware architectures to support vectored software interrupts or similar mechanisms (221 calls on a i386 family in 2.4 kernel).

4.3 The actual socket API

The actual Socket API looks something like this.

[Figure 64 about here.]

The `socket()` system call created an end point for interprocess communication. It returns the socket descriptor, which is used in later calls to setup the state for protocols that require that, and to send and receive data to a far end. The API is supposed to be quite general purpose, and not tied specifically to the TCP/IP protocol suite, and to prove this, there are in fact quite a few *Protocol Families* implemented below this API. Which family is intended is indicated by the *domain* parameter. Linux currently supports:

[Table 8 about here.]

Unix domain sockets are analogous to the named pipes discussed earlier. The `netlink` socket is used to provide a control channel between user space and kernel protocol control functions, and is discussed later. The other socket protocol families supported come from a variety of protocol worlds, but the ones we are most concerned with in this book are the `PF_INET` and `PF_INET6` ones.

For a given protocol family, one then also defines a socket *type*, which essentially selects a protocol *semantic*, although the semantics can be refined later with calls to `setsockopt()`. Semantics currently defined include:

[Table 9 about here.]

The final parameter in the `socket()` call indicated the specific protocol required. In general, for a given family, for a given semantic, there is usually only one protocol supplied in a protocol suite. However, if there is more than one, this allows one to select the appropriate one wished for.

Once a socket descriptor is returned, there are a set of further calls to carry out protocol state setup, and data exchange. It is also possible to change the behaviour of these calls in terms of whether they are *blocking* and whether they are *synchronous*.

The `fcntl()` call allows one to setup a socket so that calls to input routines that would normally block (i.e. not return from the kernel) until the operation is complete, return immediately and indicate whether they worked or need to be re-called. Subsequently, the use of `poll()`, or `select()` calls allows one to check if there is input (for up to some specified amount of time). `fcntl()` can also be used to setup a process group to receive *signals* for input and output events,

instead of having to explicitly call input read routines. We show examples of both of these mechanisms later in this chapter.

The other routines in the socket API include:

[Table 10 about here.]

The following table illustrates the socket options that can be set or retrieved by `setsockopt()` or `getsockopt()` and what they are intended for:

[Table 11 about here.]

When a socket is setup and the protocol is specified, there are further, *protocol specific* parameters which may be configured. Generally, these are setup via the `sysctl()` interface. Examples of these are mainly TCP specific parameters - here are some of these. The internal implementation of these is discussed in chapter seven. These work using the `/proc` interface (see `/proc/sys/net/ipv4/` for details).

[Table 12 about here.]

Also, there are socket level parameters:

[Table 13 about here.]

There are even a couple of TCP specific `ioctl()`'s too:

[Table 14 about here.]

4.4 Using the Socket API

Communication between separate processes involves moving data between one address space and another - in former times, this was known as crossing protection *domain* boundaries. BEcause of this, the various communicaitons facilities are named under this confusing use of the word domain (*nothing* to do with the Domain Name System at all!).

The example C code below illustrates the use of an *INET* domain socket, with stream semantics. This basically makes use of TCP code in the kernel - in the example here, communciaiton is between a parent and child process, so the socket is simply being used to replace a pipe.

[Figure 65 about here.]

The `fork()` call sets up the second process (child) so that now we have two processes, one of which is a client, the other a server. They then setup up a connection oriented scket to communicate with each other.

[Figure 66 about here.]

The client side here is the pro-active end which takes the iniative and opens the connection. Then the server side is (hopegfully) already sitting there waiting ("listening") for an inbound connction, which it then accepts.

[Figure 67 about here.]

Normally, of course, the client and server would be unrelated processes on unrelated computers!

Polling a socket

It is often useful to be able to check if data is available on an interface without actually waiting until it is. This is known as polling. The way it is done from the unix file system and socket API is via the `select()` system call. This takes a mask parameter which allows one to poll several file descriptors simultaneously. This is very useful for programs which wish to provide the appearance of concurrent user interface interaction and network I/O, for example, network management programs, or networked graphics programs.

[Figure 68 about here.]

As well as `select()`, many systems implement the `poll()` call.

[Figure 69 about here.]

`poll()` is a variation on the theme of `select`. It specifies an array of `nfds` structures of type shown below:

[Figure 70 about here.]

The timeout in milliseconds, rather than the more complex structure in `select()`.

Non Blocking I/O

In some cases, we are not interested in waiting to see if output has finished yet or not, but we do wish to start it. In other cases, the programmer would like to read as much data as there is available immediately if it is ready, but not to wait if there is no data yet. For this purpose, *Non Blocking I/O* is provided, and is triggered by setting a flag in the process data associated with the file descriptor.

[Figure 71 about here.]

Finally, we may wish to carry out some set of tasks, but be interrupted with the arrival of new data - this can be done in user space with the use of `signals`. For I/O, this is setup as follows.

Asynchronous I/O

Figure 72 illustrates asynchronous input. Here the application continues with its current work after registering an interest in call-backs for given events.

[Figure 72 about here.]

An example of an application using this is shown below in figure 73.

[Figure 73 about here.]

Alternative programming models

Other languages such as Java, have built in mechanisms for concurrency such as *threads*. It can be very elegant for event handling programs to program using these.

4.4.1 Those System Include Files

There are several include files in the examples above which are quite generic to all socket programming, and provide the definitions of types to use the socket system layer. These are installed in `/usr/include/netinet` from the source tree.

[Figure 74 about here.]

The IPv6 version of this is as follows (and is, not surprisingly, more complex):

[Figure 75 about here.]

Here also, and in related include files, there are definitions of the structures used to communicate through the socket layer calls such as `setsockopt()`.

4.5 Summary

In this chapter, we have taken a brief look at the application programmers interface to kernel communications services, with a survey of some of the different models for programming.

In the next chapter, we will look at the infrastructure *within* the kernel for implementing link, network and transport layer protocols.

Chapter 5

Protocol Implementation Framework

“The penguins had the most powerful army in the world. So had the porpoises.” - Anatole France

5.1 Roadmap

This chapter covers the overall background to kernel protocol implementation. It is bottom up in that we start with hardware and interrupts, go through drivers, then look at buffering, and multiplexing and finally examine the socket glue.

To some extent, we have given an overview of this in the preceding chapters, but here we go into more detail about the protocol implementation framework and the architecture within which the functions for multiplexing and layering fit. We look at the split between kernel versus user space again, and pay particular attention to the Linux structures defined for network protocol buffers which encourage efficient memory management as data and control are passed up and down the stack.

We look at interrupt handling, and DMA for different I/O devices, and how to deal with expected versus unexpected events.

We look at the support for network devices and show the way that this encourages (at least to some extent) modularity, abstraction and hiding

We look at a day in the life of a network interrupt. Then we look at handling users' calls to the socket layer. We then take a walk through a day in the life of a buffer. Following that, we wrap up with a look at the day in the life of a protocol control block.

5.2 A Day in the Life of a software interrupt

5.2.1 Concurrency in the Linux Communications Stack

As discussed in chapter two, the design of kernel functionality, including communications services, is concerned with both reliability, and performance. The

reliability is largely a feature of correctness of the protocol specifications and implementations, but performance is much more of an art. It is measured by several metrics, including code size, execution time, and memory use. Code size in Linux is typically on the low side. Execution time must also include the degree to which concurrency (both interleaving, and true multi-processor concurrency) are exploited. Memory use is constrained by good discipline in the allocation of buffers and other dynamic structures (e.g. protocol management structures) and timeliness in freeing up resources when no longer needed, or needed elsewhere. Another consideration is fairness, which is a topic for a whole book on its own.

Within OS, protocol stack is arranged as a pair of loosely coupled producer consumer chains for output and input. This avoids well known deadlock problems and gives natural model for buffer use disciplines, *and* for critical region protection! State management (through the Protocol Control Block for each session) provides a connection between one system call and the next, and between threads that handle interrupts and threads handling system calls. Queues and counters provide data for flow control. Sleep/Wakeup semaphore/queues are used to synchronise sequence of process/threads.

Critical Regions are then kept to a minimum. User process can only get to kernel through system calls, and once there, the kernel has control, except for timer and device driver interrupts. Thus we can have three levels of code accessing *global* data in the protocol stack state (PCB and Socket structures).

- Upper layer “call”
- Timer event
- Lower Layer “call”

Use interrupt level setting or spin locks as explained in chapter two makes sure that code accessing or modifying the global state in the protocol stack from one of these three “directions”, is not pre-empted inappropriately.

5.2.2 Producer/Consumer Chain

Ideally, a protocol stack consists of two chains of tasks, one for output, one for input. These are loosely coupled (e.g. via resource management, or via the protocol itself, for example when a data packet requires acknowledgement, and acknowledgements are piggy-acked on data). This is illustrated in figure 76.

[Figure 76 about here.]

In principle, there is a variety of ways to implement these chains of tasks. In practice, the TCP/IP protocols make life relatively regular for the modern operating systems programmer, since the IP level is so straightforward (at least superficially - see chapter six for the glaring reality!). This means that the interference between input and output chains is relatively simple, and restricted to a few places where we need to have communication via the protocol state machine, typically in TCP, or through resource management in the buffer allocation in the output flow or input flow.

Then protocols in the chains or flows have to look after a number of communication specific tasks, including:

Multiplexing and De-Multiplexing State is held at the socket, transport protocol, network and driver level to indicate how to handle incoming packets and pass them on to the appropriate handler at the next level up. Socket state gives session to process mapping. Transport protocol state to some extent repeats this data. Network protocol state is longer lived - essentially, in the IP world, it represents the binding of the IP protocol identifier to a transport protocol. Then at the driver level, there are indications in the frame level which network layer to demultiplex a packet to, and when sending a packet, there is a binding between an IP address and the link layer addressing, maintained by address resolution tables (or neighbour tables).

Reliability and Resource Control Reliability and resource management both require timers. The problem that they have in common is that when things *stop* working, that is exactly the time that the system doesn't have resources to tell you. For example, packet loss may happen because of congestion, which is when there is not enough bandwidth to send packets (e.g. to tell you there is congestion). Instead we use autonomous timers (watchdogs) that fire after the interval when we would *expect* there to be an event. Each timer are associated with a layer in the producer-consumer chains described above, and therefore can be associated with the appropriate global state for that protocol session.

Buffering, encapsulation and de-Encapsulation Autonomous communicating systems (layers, or different computers) have different logical and physical resources for buffers and packets so choose to allocate different sizes. This means that as one traverses layers in the producer consumer chain, on output, one may need to break large packets into smaller pieces, or, going the other way on input, aggregate small pieces together into more convenient (or logically more complete) units. This data is also appropriately stored in protocol control blocks:- for IP, there is fragmentation on output, and re-assembly, with a timer for lost fragments on input; for TCP, there is segmentation and re-assembly, which operates directly to the socket layer.

As discussed in chapter three, the way that network input and output work is driven from software interrupts. For output, the producer is the application process. For input, the network interrupt.

Concurrency Problems - what if we had no discipline!

Problems with concurrency:- why these Design Patterns are useful discipline:-

There are a variety of problems with interleaving code from different places - three well known problems are

Deadlock - a has x and needs y, b has y and needs x to proceed, as shown in the diagram, 78.

Livelock - a sequence of trades of x and y without either party ending up with the necessary full set of resources, as illustrated in figure 78.

Fairness - this is discussed further in chapter nine.

[Figure 77 about here.]

[Figure 78 about here.]

If one process, thread, task or driver gets more turns at the CPU (or memory, or bus or whatever resource is congested) we have what is called *unfairness*

Fairness is complex, but typically, a round robin scheduler can control fairness. In general, protocol fairness is even more complex as it includes fairness of more than one system (far end, links in between and so forth!).

5.2.3 What about real Multi-processor Systems?

Of course, the model above of a simple producer-consumer chain potentially breaks down (or is at least a lot more complex) on a true multi-processor systems.

Linux is now pretty much fully Symmetric Multi-processor capable. Atomic access to shared data must be protected across processors, since any layer (e.g. driver interrupt, software interrupt, system call handler) may run on any processor.

In chapter two, we discussed the mutual exclusion support that this requires, the technique called a spinlock. For input and output, this is a relatively simple (localized) change to all the communications code that has happened in the last few years of evolution of the Linux kernel so that now we have greatly enhanced performance in multi-processor hardware systems. The remaining area for performance enhancement is in SMP forwarding, but this is a topic for future research at the moment, as mentioned before.

5.3 Bottom Up: Interrupts, DMA etc

Now network interrupts are dealt with under software interrupts, and passed up the “foo chain” by such. The software interrupts are separated into receive and transmit side, and essentially schedule a set of tasklets, which replace Linux’s older mechanism of *bottom halves* or deferred interrupts, which were not easy to make work under SMP!

Instead, linux now has a structure for tasks that happen in an interrupt context, and thus can be per processor in an SMP machine.

[Figure 79 about here.]

In figure 80 we see the definition of the core data structure to describe the tasklet queue structure:

[Figure 80 about here.]

Now we have to support legacy deferred interrupts, since a great deal of code is still written in terms of this older Linux construct.

[Figure 81 about here.]

It is common to define some inline functions in the header (same place as relevant structures are defined):

[Figure 82 about here.]

Now, lets look at the soft IRQ mechanisms that use this.

[Figure 83 about here.]

In the next extract, the comment makes it clear that SMP interaction with soft IRQs is not yet a well understood performance area:

[Figure 84 about here.]

Here is the code that actually acts on the soft IRQ, and fields it to the appropriate handler.

[Figure 85 about here.]

This part then makes sure that we don't nest hard and soft IRQs arbitrarily which would lead to bad things happening!

[Figure 86 about here.]

This section kicks off the soft IRQ vectors.

[Figure 87 about here.]

Tasklets

As we mentioned before in chapter two, Linux recently replaced the whole bottom half mechanism for deferred interrupts with two levels - the lowest level was soft interrupts, shown above, and is used especially for networking. The next level is the general tasklet framework which enables processors in an SMP system to pick a deferred interrupt task from a queue.

[Figure 88 about here.]

In the code above, we see the way that tasklets are dequeued when the general handler is called and then the `t`

-`func()` is applied to the `t`

-`data`.

Next, the exact same structure is seen (just the function declaration) to handle the high priority version of the tasklet queue - maybe, eventually in some later Linux release, tasklets will be in a multi-level queue, although its not clear that many levels of deferred interrupt priority makes a lot of sense when it gives you finer grain concurrency, but coarse grain multi-processing is cheap!

[Figure 89 about here.]

The whole tasklet framework has to be kicked into action, and thats what we see here:

[Figure 90 about here.]

Bottom Halves

In the past, we had bottom halves. We still have a lot of legacy code written in terms of these, so the code must be called from the tasklet framework. It just follows the rules about serialization outlined in chapter two:

[Figure 91 about here.]

This installs the `bh_` in the tasklet queue.

[Figure 92 about here.]

This sets up the whole soft IRQ and tasklet system and starts it going.

[Figure 93 about here.]

5.4 Device Level, bottom up, one level up

Moving up one level from generic interrupts, we take a look at network device support.

Since many years ago, operating systems have provided a standard abstraction/interface for classes of device, so that higher level operating system (and application) software can be common across all such devices. Unix traditionally divides devices into 2 classes, Character (low rate, interactive, serial line typically) and Block (Disk, Display, etc). Its possible to squeeze network devices into the block mode paradigm, but its messy:- Linux adds a 3rd type of device - network. This is discussed in some detail in chapter two.

Typically, device has name to place it in the file namespace, but also has identifier: Unix has major/minor numbers. A driver is a structure (class) with a set of entry points (functions/methods). At boot (or module load) time, the device is initialised by calling its `init()` function - this resets the device, and installs any relevant interrupt handlers and so on. It then registers with the OS. The rest of the time, we manage I/o with device with `open`,`close`, `queue_xmit`, and interrupts/notifications

Device driver details manages specifics like:

- Bus interface/memory/I/o address of device registers
- DMA and timer chip use
- Irqs, etc

Notice the assymetry of input and output - output is requested, whereas input arrives unexpectedly (I.e. at this level we could never sensibly implement a device driver by polling!) Input results in packets being queued, and `netif_rx()` called to find out which higher level protocol function to dispatch

The traditional catchall interface is via the `ioctl()` call. This allows one to set and get anumber of network specific device control parameters - e.g.

- Link up/down
- Broadcast address

- Arp -see later
- Mtu
- Statistics. etc

We can see these by saying `ifconfig <dev>`. The command `netstat -i` will tell you what network devices you have. We look at this further in chapter six.

Address resolution is done either algorithmically, or through a table, or through a helper task which runs the Address Resolution Protocol (ARP). Arp is invoked from output when the IP address is known but the destination link layer (or strictly, Media Access Control) address is unknown. Arp is demuxed in `netif_rx()` for input frames with ARP type - it makes sure that it enters table updates and maintains these with a lifetime so that they expire when not used in a timely fashion. It can be overridden by configuration management

In general, output is multiplexed by indicating frametype (IP, ARP, or other). Input demuxes, by looking at frametype, and using to *upcall* the right higher level (from inside `netif_rx()`, called from the input interrupt)

As discussed before, buffer disciplines are asymmetric: for output, the higher level allocates buffer, calls stack of send routines, and the lowest level frees, while for input, the low level has a pool, hands up to upper level, which, when done with a buffer, frees.

What if packet is *not for us*? Basically, will either bridge, route, or discard. Bridging is intensive (requires the ethernet interface to be in promiscuous mode, which is most expensive in packet discard!). We don't discuss that further.

Routing is a builtin part of linux and requires a forwarding table, and is discussed in much more detail in chapters eight and nine. Discard is often the most common case and requires efficient handling. There has been lots of good work on efficient filtering mainly from the implementation of network monitoring and firewall mechanisms, which are discussed in chapter ten.

Most hardware (driver level) batch packet handling to reduce the interrupt count to much less than one per packet. This is good for performance, although it results potentially in bursts of IP packets arriving to the higher levels (or for forwarding). This doesn't cause problems too often in the common cases, and can be controlled using network schedulers and shapers, as discussed in chapter nine..

Now we take a detailed look at a specific network device by walking through the driver code.

5.4.1 Network Device Driver API

We covered the overall view of drivers in chapter two. In this section, we take a look at the specifics of a network device in more detail, and then use a classic example of an ethernet driver for a very well known device to provide some specifics.

The Linux network device driver interface information is basically all held in one general data structure - it is fairly horrible, as it combines various protocol layers in one place!!!

Here is the main network device data structure.

[Figure 94 about here.]

Common across all network devices is initialisation.

[Figure 95 about here.]

Then we have the generic network device data and statistics, including MAC addresses - for a LAN card we have to support broadcast address(es) as well as unicast, and various modes such as *promiscuous*, where the card accepts frames to all addresses, not just those in its address filter table.

[Figure 96 about here.]

In chapter nine, we'll see how output queueing disciplines are attached to a device - here we see that the generic network driver level is where the hooks for this are:

[Figure 97 about here.]

[Figure 98 about here.]

5.4.2 Network Reception

This is a generic receive handle/demultiplex function called from receive interrupts from all network device drivers.

[Figure 99 about here.]

5.4.3 A specific device driver - 3c501

This is a 3Com Ethernet card driver from Linux 2.4. This is a pretty old device, but it has enough intelligence to show some of the features of Linux. In fact, some of its limitations also mean that a driver for it must also exhibit some classic workarounds for well known device problems - some device problems never go away even in state-of-the-art hardware!

The `net_device` structure illustrates what data is associated with the hardware specifics of network device, including bus interface and memory and interrupt priority as well as external card specifics.

Basically, we are usually given an I/O address that a card is memory mapped to, and then a given card has a number of registers for control and status - in this case there are also addresses we can read internal configuration back from the card (e.g. Manufacturer assigned Ethernet address for example).

[Figure 100 about here.]

The transmission and reception are (typically) separated out, and there are status bits that indicate specific conditions associated with the type of link (Ethernets have lots of ways that frames or the media can present problems, and device manufacturers like to enable the device driver writer to be able to get at all of these - in most cases, the actions in the driver are simple (do nothing except count statistics) and it's up to network managers to look at the statistics and notice problems!

[Figure 101 about here.]

3c501 Initialisation

Here we see initialisation of the device. There are some specifics to do with interfacing to the operating system so that the data is available and the driver for the card can be started. We see the retrieval of the *station* or MAC address from the card here.

[Figure 102 about here.]

Here we probe to find the IRQ that the device is on.

[Figure 103 about here.]

If all is well so far, we try to allocate space for the net device structure.

[Figure 104 about here.]

This section of the startup fills in the device structure with the function pointers for this specific driver. If all is well, it then does the generic Ethernet initialisation (since there is a lot in common across all Ethernet devices).

[Figure 105 about here.]

3C501 Start transmission

DMA network devices have quite a lot of autonomy (i.e they can get on with quite a lot of work in parallel with the processor(s)). Sometimes, though, they can be quite slow about some phases so it is common practice to separate out the work, and make as few (and as short) critical regions as possible.

[Figure 106 about here.]

The next few code extracts show the continuation of the transmission *startup* code. Essentially (as the comments indicate) we are telling the card that there is data ready to send, and where it is, then we tell it to “go away and send it”, and interrupt us when done.

[Figure 107 about here.]

[Figure 108 about here.]

[Figure 109 about here.]

3c501 interrupt handling

At the top level, we have to handle a generic device interrupt from this device and figure out then what actually happened that it is trying to tell us about (i.e. another example of de-multiplexing!).

[Figure 110 about here.]

So we read the status from the card to figure out which event triggered this.

[Figure 111 about here.]

In many cases it may have been an error condition (such is life on a CS-MA/CD network!). In a few cases, we are done with something, and we want to get on with the next piece of work so we call `netif_wake_queue` to find what more there might be to send.

[Figure 112 about here.]

[Figure 113 about here.]

Of course, there is also the case that we have received a packet. Then we want to deal with that to.

[Figure 114 about here.]

Many problems can still be present even though the hardware has delivered a packet and an interrupt to say so. We have to check for these (remember the Postel quote at the start of chapter three).

[Figure 115 about here.]

Here we start the work of receiving a packet in earnest. We check for simply illegal packet lengths for example.

[Figure 116 about here.]

Finally, if all is well, we call `netif_rx` on the socket buffer that has the packet data.

[Figure 117 about here.]

3c501 device specific statistics

[Figure 118 about here.]

5.4.4 link driver level config

The widely used `netstat -i` command, shows data such as this:

[Figure 119 about here.]

Meanwhile, the `ifconfig` command reveals:

[Figure 120 about here.]

3c501 multicast support

Here is how the driver supports multicast:

[Figure 121 about here.]

3c501 module support

Finally, here is the interface to the kernel so that the device driver can be installed as a module:

[Figure 122 about here.]

5.5 Top Down: Socket Glue Level

In the previous part of this chapter, we've looked very much bottom up from the generic to the specifics of driver (link) level support for networking.

In the previous chapter, we looked at the API used to program networked applications. Now, working further down from the top level, we can look at how the user/system interface level maps onto kernel code, and the kernel code looks as follows.

First, as this level was complex, there are a lot of programmers to give due credit to.

[Figure 123 about here.]

There is a copy of the GNU General Public License here too:

[Figure 124 about here.]

Moving on, we can see the set of C prototypes for the set of exported functions from this API.

[Figure 125 about here.]

As explained in chapter two, the socket code is yet another variant of the virtual file system API, so it must provide the mappings for the generic functions therein. These are wrapped up in the `file_operations` structure:

[Figure 126 about here.]

Slightly more "bottom up", again, we also have to have linkage to the set of protocols that the socket layer supports. This is done through an array of `net_proto_family` structures, each one holding pointers to the relevant input and output functions (and others) for that protocol family. These are discussed briefly in chapter three, and also in chapter seven on transport protocols.

[Figure 127 about here.]

Locking Sockets

As discussed in chapter three and four, this is the layer in which a lot of concurrency can occur (between user processes and kernel transport protocols) thus there needs to be support for mutual exclusion and synchronisation. Here is some of it.

[Figure 128 about here.]

Each CPU needs to keep copies of this data.

[Figure 129 about here.]

When moving buffers between a user process and the socket layer (actually, any virtual file system instance), we need to check that the buffer *is actually* owned by the user, so that we don't make any terrible privilege violations. Checks need to happen from user to kernel;

[Figure 130 about here.]

We also check buffers when copying data from the kernel to the user.

[Figure 131 about here.]

5.5.1 Virtual File System Socket instance functions

Now we can look at the actual instances of the virtual *file system* functions that implement the socket (i.e. protocol interface) functions.

This function returns file system status - for a socket, thats not really terribly meaningful, but we try.

[Figure 132 about here.]

Read a block from a “file”.

[Figure 133 about here.]

The following operations really make very little sense at all on sockets (of any address family!).

[Figure 134 about here.]

“Seeking” is another thing that we do on storage device type file systems, but doesn’t make a whole lot of sense on a socket!!!

[Figure 135 about here.]

Memory-mapping a socket doesn’t really make sense!

[Figure 136 about here.]

Mapping socket and file descriptors

Now we need to map descriptors so that we can use file system management code on socket descriptors as if the are file descriptors.

The first function finds a spare file descriptor:

[Figure 137 about here.]

We then need to map inodes to socket descriptors and vice versa.

[Figure 138 about here.]

The next part fills in the file function points and other administrative parts of the structure.

[Figure 139 about here.]

We need to map the other way too:

[Figure 140 about here.]

[Figure 141 about here.]

We need to allow free passage between the inode and socket descriptor space. Here is where we allocate and associate the relevant numbers:

[Figure 142 about here.]

Of course, we also need to be able to return these identifiers to the pool of free ones:

[Figure 143 about here.]

This was all for the convenience of the operating system so that we have a single unified interface between the system call code and various lower level services, both file system, and inter-process communication, including a unified set of identifiers.

Back to the VFS socket code instances

Now we return to looking at more instances of socket functions that implement virtual file system functions - here is the code for accessing the `open()` call:

[Figure 144 about here.]

Then we need the inverse of this to deal with tearing things down (either explicitly or after process exits).

[Figure 145 about here.]

Finally, some real work is involved when we want to actually send or receive messages on a socket:

[Figure 146 about here.]

We also need to support the `read()` call:

[Figure 147 about here.]

Similarly, there's also the `write()` system call:

[Figure 148 about here.]

Non contiguous I/O buffer parameters

The following routines manage the calls that users make with non-contiguous buffers (`iovec[]` structures).

[Figure 149 about here.]

The function above is a service function for the separate handlers here:

[Figure 150 about here.]

When a user process calls the `sendto()` socket API call, it maps into this:

[Figure 151 about here.]

More of this `sendto()` (probably a datagram) follows.

[Figure 152 about here.]

When a user process calls `recvfrom()`, it maps to this code. This means we have a user who wants to receive a packet (typically a datagram).

[Figure 153 about here.]

[Figure 154 about here.]

[Figure 155 about here.]

Socket layer VFS function for ioctl

The `ioctl()` call really is an all purpose, swiss-army-knife of a system call!

[Figure 156 about here.]

Polling, as discussed in chapter four, is like `select()`, only simpler.

[Figure 157 about here.]

Closing a socket frees up the appropriate resources of course.

[Figure 158 about here.]

Asynchronous events on sockets

The use of asynchronous input was discussed in chapter four. Here we see how it is mapped through the virtual file system again onto the socket code. Asynchronous input events are of course more likely on a network interface than a disk!

The socket layer associates a list of functions to call with the socket via a *callback* field which are then invoked by the lower layers as necessary. Notice careful locking of the socket to avoid inconsistency in call backs during changes to the list!

[Figure 159 about here.]

Here's some more of the function.

[Figure 160 about here.]

And the next part:

[Figure 161 about here.]

Here's where we handle removing a callback from the list.

[Figure 162 about here.]

The next piece of code actually fields the asynchronous calls.

[Figure 163 about here.]

Creating a socket

[Figure 164 about here.]

Start the module up if it isn't already going!

[Figure 165 about here.]

Associate the module with the right protocol:

[Figure 166 about here.]

Creating a socket pair

Socket pairs are a useful way to emulate a pipe.

[Figure 167 about here.]

[Figure 168 about here.]

[Figure 169 about here.]

Bind, Listen, Accept

All three of these functions are associated with the process of binding one or other end of a connection to an identifier (effectively an IP address) or an active protocol association (i.e. transport level connection).

`bind()` takes the socket and associates the local end of it with an address (typically, an IPv4 or IPv6 address):

[Figure 170 about here.]

tt `listen()` deals with awaiting an inbound connection request:

[Figure 171 about here.]

`accept()` is called when an inbound connection request informs the application by a return value from `listen()`:

[Figure 172 about here.]

And there's some more work here on accepting a connection.

[Figure 173 about here.]

If all is well, we go ahead.

[Figure 174 about here.]

At the other end of things, the active (client) end of a protocol calls this to cause the protocol to try to setup a connection to a given remote address.

[Figure 175 about here.]

Book keeping of connection identifiers

We can ask the socket layer for the addresses that the local, or far end of a “connection” are bound to.

[Figure 176 about here.]

In this case, it is like asking “Where did a connection come from”?

[Figure 177 about here.]

Setting various socket layer parameters (see previous chapter for the use and meaning of these!).

[Figure 178 about here.]

Getting various socket parameters (see previous chapter!).

[Figure 179 about here.]

BSDisms - these calls are derived specifically from BSD Unix

As well as reading, writing, sendto, recvfrom and the iovec versions, we also have the interface for message like structures that derives from the BSD Unix family.

[Figure 180 about here.]

[Figure 181 about here.]

[Figure 182 about here.]

[Figure 183 about here.]

Other VFS calls

[Figure 184 about here.]

[Figure 185 about here.]

[Figure 186 about here.]

[Figure 187 about here.]

Associating a protocol with the socket layer

We can dynamically install protocol families at run time.

[Figure 188 about here.]

We may also want to remove socket protocols, so they need to be de-registered.

[Figure 189 about here.]

[Figure 190 about here.]

Then the socket VFS module itself may be installed or removed at runtime to, as needed.

[Figure 191 about here.]

And we also need to keep statistics (per CPU) on the socket module layer.

[Figure 192 about here.]

5.6 Sideways: A day in the life of a socket buffer

Linux has a fairly complex sophisticated network buffer type, which is somewhat different than other opensource operating systems that have been documented.

This picture 193 statically tries to show the regions of the `sk_buff` structure - these change dynamically as a reference to a buffer is moved up and down the stack. Essentially, buffers are allocated at the producer end of a producer-consumer chain, passed through the protocol layers, and freed back to the pool at the consumer end. Thus for input, drivers allocate buffers, and pass them up through the demultiplexing layers up to the socket layer, where data is copied into user space, and the buffer freed. For output, the data is copied from user space into buffers allocated by the transport protocol and passed down through the layers until it is transmitted.

Notice that if this discipline is followed systematically, there is at most one copy (ignoring the DMA from a device into memory) and that even this can be eliminated (and has been in some research Linux systems) by carrying out the kernel/user space mapping via memory management and marking the buffer as copy-on-write.

[Figure 193 about here.]

The `sk_buff` structure is critical to understanding the networking code. It is used to hold a plethora of different types of network data, although principally, this is packet buffers (including their nested headers and so forth).

In the next two listings in figures 194 and 195, you can see the way that these structures are part of a linked list, and can in fact be on several lists. They can be owned by a socket that is sending or about to receive them (high level interface), and have a link to a device that they either arrived or will be sent on (low level interface). There is a timestamp associated with when they were last timestamped (depends on the use or last protocol that touched the field and how it wishes to communicate with the next - this can be particularly useful in scheduling packet output, as discussed in chapter nine).

[Figure 194 about here.]

The socket buffer structure *overlays* a number of possible interpretations of the actual contents of the memory it points to - the headers have to include transport, network and link layer information for fast access and multiplexing/demultiplexing, but for each protocol layer the `union` of all types of different packet header for the different available protocols at that layer is used for programming convenience.

[Figure 195 about here.]

Manipulating sk_buffs

There are a number of functions for manipulating socket buffers. Here are the definitions of the publically available ones.

[Figure 196 about here.]

5.6.1 Allocating a Socket Buffer

[Figure 197 about here.]

[Figure 198 about here.]

In TCP, there is a wrapper function for this, called `tcp_alloc_skb()`

[Figure 199 about here.]

Here we see the definition of the standard functions for accessing the `skbuff` structure.

[Figure 200 about here.]

5.6.2 Allocating and freeing a socket buffer from the global pool

[Figure 201 about here.]

The following extract from the `skbuff` code describes the allocation of a socket buffer in some detail:

[Figure 202 about here.]

The code to allocate a socket buffer calls lower level kernel functions to actually allocate the right type of memory.

[Figure 203 about here.]

[Figure 204 about here.]

[Figure 205 about here.]

5.6.3 Manipulating Socket Buffers

[Figure 206 about here.]

The following function copies form, but not content, bumps reference count!

[Figure 207 about here.]

Here's the global data part of the code to manipulate socket buffers.

[Figure 208 about here.]

[Figure 209 about here.]

Finally, we get to the code which sets up the actual pool of buffers.

[Figure 210 about here.]

5.7 A day in the life of a Protocol Control Block

Here is the relevant data for the protocol control block vis-a-vis the state of a stream type socket in the INET protocol family!

[Figure 211 about here.]

[Figure 212 about here.]

[Figure 213 about here.]

[Figure 214 about here.]

There is a lot more to the TCP state machine than this, but we'll look at that in the next but one chapter.

5.7.1 Socket and Route related transport state

For historical reasons, some state associated with the transport protocol is kept in the socket structure (it is, in any case, per connection per process for connection oriented unicast protocols, so this is not a terrible error, although in future for reliable multicast transport protocols with multiple senders and receiving processes potentially on the same machine, this could be a problem).

The route related state is discussed in chapter eight. In any case, one of the most formidable data structures in the Linux communications protocol stack implementation is contained in the innocently named file `sock.h`:

Along this is the `tcp_opt` struct, which is discussed in detail in chapter seven.

[Figure 215 about here.]

There is also IP level information (e.g. `struct ipv6_pinfo` and `struct inet_opt`, and then socket level data itself:

[Figure 216 about here.]

[Figure 217 about here.]

[Figure 218 about here.]

[Figure 219 about here.]

[Figure 220 about here.]

[Figure 221 about here.]

[Figure 222 about here.]

[Figure 223 about here.]

[Figure 224 about here.]

5.7.2 Future of the socket structure

There is a comment to the effect that its desired by the code authors to tidy this all up, which describes a much more elegant solution:

[Figure 225 about here.]

5.7.3 Reference COunting

Further on, there are two more important facets of the socket state maintenance code. Firstly we see the reference counting:

[Figure 226 about here.]

[Figure 227 about here.]

[Figure 228 about here.]

[Figure 229 about here.]

5.7.4 Sockets and Routes

Next we see the code to associate a socket with a route

[Figure 230 about here.]

[Figure 231 about here.]

[Figure 232 about here.]

Penultimately, there is a set of functions to queue data safely to and from a socket (remembering that a socket is shared between user process and kernel threads):

[Figure 233 about here.]

[Figure 234 about here.]

Finally, we have some macros to control synchronisation between user and kernel via socket wait queues:

[Figure 235 about here.]

5.7.5 Socket Credentials Manager Interface

Its necessary to control use of resources in the socket interface - this is done through keeping a list of credentials for access to socket buffers (basically, typical unix user and group identifier information).

An example function is illustrated in figure 236 which maps between given credentials and actual process rights.

[Figure 236 about here.]

The interface to this code is like this:

[Figure 237 about here.]

5.7.6 A brief with comparison BSD Style Stack Infrastructure

By way of contrast, lets look at the way that the BSD Unix kernel might achieve the synchronisation:

[Figure 238 about here.]

[Figure 239 about here.]

Here is a brief outline of what the pseudo-functions above do:

[Figure 240 about here.]

5.8 Summary

In this chapter, we have looked at the general support for protocols in the kernel, from network device driver and software interrupts, through to the socket glue, taking in the socket buffer data structures and the protocol multiplexing and demultiplexing layers along the way.

In the next chapter, we look at the network layer protocol IP, itself, and its associated management protocols.

Chapter 6

Infrastructure Protocols

“Three things are inevitable in life: death, taxes and middleware.” - anonymous.

6.1 Roadmap

In this chapter, we cover the protocols that provide an internetworking infrastructure, that is to say IP (versions 4 and 6), and the associated management protocols such as ARP, DHCP and ICMP and IGMP.

We introduce the basic concepts of the MAC Address, IPv4 Address and look at proposed IPv6 Addressing, and Multicast Addressing. We then look at autoconfiguration including ARP, and IPv6 stateless autoconfiguration.

Then we look at IP host packet processing for sending and receiving and, briefly forwarding.

6.2 IP

6.2.1 Internet Protocol Output

[Figure 241 about here.]

[Figure 242 about here.]

[Figure 243 about here.]

[Figure 244 about here.]

[Figure 245 about here.]

[Figure 246 about here.]

[Figure 247 about here.]

[Figure 248 about here.]

[Figure 249 about here.]

[Figure 250 about here.]

[Figure 251 about here.]

[Figure 252 about here.]

[Figure 253 about here.]

[Figure 254 about here.]

[Figure 255 about here.]

[Figure 256 about here.]

6.2.2 Fragmentation

Here is where IP output loops over a buffer that is bigger than the MTU size and generates fragments to send:

[Figure 257 about here.]

[Figure 258 about here.]

[Figure 259 about here.]

[Figure 260 about here.]

[Figure 261 about here.]

Non fragmented transmission

It is common that the transport layer has used MTU discovery to find the maximum packet size tolerable on a path so that fragmentation then subsequently is not necessary. In this case, we want super-fast code to handle the situation.

[Figure 262 about here.]

[Figure 263 about here.]

[Figure 264 about here.]

Here is the actual tight loop over the transport packet buffer, outputting fragments:

[Figure 265 about here.]

[Figure 266 about here.]

[Figure 267 about here.]

[Figure 268 about here.]

[Figure 269 about here.]

Non contiguous buffers

As in chapter five, we need to handle non contiguous buffers. It would be excellent if this could be done across the entire stack without generating packet copies or new `sk_buff` allocations!

[Figure 270 about here.]

The next case is dealing with a common case, where we have a prototype packet but received from somewhere (e.g. ICMP request) and we need to generate the packet going the other way.

[Figure 271 about here.]

Initialisation

[Figure 272 about here.]

6.2.3 Internet Protocol Input

Receiving IP packets - first of all, there is a brief discussion of IP reassembly hacking.

[Figure 273 about here.]

[Figure 274 about here.]

[Figure 275 about here.]

[Figure 276 about here.]

[Figure 277 about here.]

[Figure 278 about here.]

[Figure 279 about here.]

[Figure 280 about here.]

[Figure 281 about here.]

[Figure 282 about here.]

6.2.4 Internet Protocol Forwarding

[Figure 283 about here.]

[Figure 284 about here.]

[Figure 285 about here.]

6.2.5 Internet Protocol Fragmentation Support Functions

[Figure 286 about here.]

[Figure 287 about here.]

Memory allocation and freeing for fragments

[Figure 288 about here.]

[Figure 289 about here.]

[Figure 290 about here.]

[Figure 291 about here.]

[Figure 292 about here.]

[Figure 293 about here.]

[Figure 294 about here.]

[Figure 295 about here.]

[Figure 296 about here.]

[Figure 297 about here.]

[Figure 298 about here.]

[Figure 299 about here.]

Re-assembly

The inverse of fragmentation is re-assembly. This is not strictly an inverse, since fragmentation can occur at any, and many hops along a route wherever the MTU on an output link is smaller than the MTU on the input link. However, reassembly is only done at the receiving end system. The other part of reassembly that is complex is that it requires a timer (the only timer in the IP layer) since one needs to deal with lost or out of order delivery of fragments (and to be able to discard fragments for packets which will never be complete).

[Figure 300 about here.]

[Figure 301 about here.]

[Figure 302 about here.]

[Figure 303 about here.]

6.2.6 Internet Protocol Options

IP options in IPv4 are curious things. They have a high potential value, but since (for over 2 decades) they have remained relatively un-used, most routers process them in the *slow path*. The slow path is the uncommon case code, and contains many more checks than so-called it fast path processing. In fact, when router vendors throw hardware solutions at a problem, they are committing designs to silicon which is costly, so typically they reduce the number of options to zero.

Nevertheless, we need to handle them because of hosts requirements. The loose and strict source route and record options would be very useful if they were widely implemented, although they do represent a nasty security hole in the capability for an ISP to do “traceback”.

The “router alert” is becoming an interesting exception to this, since it is typically used only occasionally in the stream of packets that rely on its effect.

IPv6 makes a different design decision to separate out hop-by-hop options from end-to-end options and give a handle to routers to find the next-header easily. We’ll look at this later in this chapter.

[Figure 304 about here.]

[Figure 305 about here.]

[Figure 306 about here.]

[Figure 307 about here.]

[Figure 308 about here.]

[Figure 309 about here.]

Option Clearing

This is the Preamble to processing options.

[Figure 310 about here.]

Next, we deal with No-op and option end case.

[Figure 311 about here.]

Process Loose and Strict Source Route Options.

[Figure 312 about here.]

Route record

[Figure 313 about here.]

Timestamp option processing.

[Figure 314 about here.]

[Figure 315 about here.]

Router Alert option handling.

[Figure 316 about here.]

[Figure 317 about here.]

[Figure 318 about here.]

[Figure 319 about here.]

[Figure 320 about here.]

[Figure 321 about here.]

6.2.7 Internet Protocol Socket Glue

[Figure 322 about here.]

[Figure 323 about here.]

[Figure 324 about here.]

[Figure 325 about here.]

[Figure 326 about here.]

[Figure 327 about here.]

[Figure 328 about here.]

[Figure 329 about here.]

[Figure 330 about here.]

[Figure 331 about here.]

[Figure 332 about here.]

Setting Socket Options

In the previous chapter we saw the generic socket layer for dealing with socket options. Here we have the IP protocol specific part of that function. Other protocol families would have different instances of this code.

[Figure 333 about here.]

[Figure 334 about here.]

[Figure 335 about here.]

[Figure 336 about here.]

Multicast socket options

[Figure 337 about here.]

[Figure 338 about here.]

[Figure 339 about here.]

[Figure 340 about here.]

Getting Socket Options

Getting socket options is just the inverse of setting them.

[Figure 341 about here.]

[Figure 342 about here.]

[Figure 343 about here.]

[Figure 344 about here.]

[Figure 345 about here.]

6.3 Addressing

Link and network addresses have specific finite lifetimes and scopes. In fact, so of course do names (as in Domain Names), and this means that we need to consider binding names to addresses from one level to the next.

address data structures We have the IP and Link layer data structures.

address formats We have the specified formats for addresses as discussed earlier.

configuration/management We can override binding and scope rules through management.

ARP Management

[Figure 346 about here.]

[Figure 347 about here.]

[Figure 348 about here.]

[Figure 349 about here.]

[Figure 350 about here.]

[Figure 351 about here.]

[Figure 352 about here.]

[Figure 353 about here.]

[Figure 354 about here.]

ARP solitication

[Figure 355 about here.]

[Figure 356 about here.]

[Figure 357 about here.]

[Figure 358 about here.]

Send an ARP

[Figure 359 about here.]

[Figure 360 about here.]

[Figure 361 about here.]

ARP Reception

[Figure 362 about here.]

[Figure 363 about here.]

[Figure 364 about here.]

[Figure 365 about here.]

[Figure 366 about here.]

[Figure 367 about here.]

[Figure 368 about here.]

ARP User Interface

[Figure 369 about here.]

[Figure 370 about here.]

[Figure 371 about here.]

[Figure 372 about here.]

[Figure 373 about here.]

[Figure 374 about here.]

[Figure 375 about here.]

[Figure 376 about here.]

[Figure 377 about here.]

[Figure 378 about here.]

[Figure 379 about here.]

[Figure 380 about here.]

ARP Initialisation and Usage Example

[Figure 381 about here.]

[Figure 382 about here.]

6.4 Network Level Debugging - ICMP

[Figure 383 about here.]

[Figure 384 about here.]

[Figure 385 about here.]

[Figure 386 about here.]

[Figure 387 about here.]

[Figure 388 about here.]

Now we look at the data associated with running ICMP.

[Figure 389 about here.]

[Figure 390 about here.]

[Figure 391 about here.]

[Figure 392 about here.]

[Figure 393 about here.]

[Figure 394 about here.]

[Figure 395 about here.]

[Figure 396 about here.]

[Figure 397 about here.]

Avoid recursive ICMP errors

[Figure 398 about here.]

[Figure 399 about here.]

[Figure 400 about here.]

Handle unreachables correctly

[Figure 401 about here.]

[Figure 402 about here.]

[Figure 403 about here.]

[Figure 404 about here.]

[Figure 405 about here.]

[Figure 406 about here.]

[Figure 407 about here.]

[Figure 408 about here.]

[Figure 409 about here.]

[Figure 410 about here.]

[Figure 411 about here.]

[Figure 412 about here.]

[Figure 413 about here.]

[Figure 414 about here.]

6.5 Group Management - IGMP

A reasonable fraction of the Internet supports multicast, particularly in tier one ISPs and in academic and research networks where the access networks are LANs. The Host Group membership system is managed through the host-router communications protocol, Internet Group Management Protocol. The Linux implementation is covered here.

6.5.1 IGMP

In this section we take a tour around IP multicast support in Linux. Multicast is a tricky concept and is intimately bound up between host and router functions, so rather than defer the main discussion to the chapter on routing, we look through most of the code here. The rationale is also that UP multicast makes use of encapsulation (tunnels) which are also discussed here as they are part host, part router functionality!

[Figure 415 about here.]

[Figure 416 about here.]

IGMP timers to manage state

[Figure 417 about here.]

[Figure 418 about here.]

IGMP Transmission

[Figure 419 about here.]

[Figure 420 about here.]

IGMP Reception

[Figure 421 about here.]

[Figure 422 about here.]

[Figure 423 about here.]

6.5.2 Managing the group/interface state

[Figure 424 about here.]

[Figure 425 about here.]

User/Group Management layer

[Figure 426 about here.]

[Figure 427 about here.]

[Figure 428 about here.]

[Figure 429 about here.]

[Figure 430 about here.]

[Figure 431 about here.]

Device Events that influence Group State

[Figure 432 about here.]

[Figure 433 about here.]

[Figure 434 about here.]

[Figure 435 about here.]

[Figure 436 about here.]

[Figure 437 about here.]

[Figure 438 about here.]

6.5.3 Multicast Forwarding

[Figure 439 about here.]

[Figure 440 about here.]

Virtual Interface support for Multicast

[Figure 441 about here.]

[Figure 442 about here.]

[Figure 443 about here.]

[Figure 444 about here.]

[Figure 445 about here.]

[Figure 446 about here.]

[Figure 447 about here.]

Multicast Cache

[Figure 448 about here.]

[Figure 449 about here.]

[Figure 450 about here.]

[Figure 451 about here.]

[Figure 452 about here.]

[Figure 453 about here.]

[Figure 454 about here.]

Multicast forwarding and Lookup

[Figure 455 about here.]

[Figure 456 about here.]

[Figure 457 about here.]

User management of multicast forwarding cache

[Figure 458 about here.]

[Figure 459 about here.]

[Figure 460 about here.]

[Figure 461 about here.]

Multicast socket options

[Figure 462 about here.]

[Figure 463 about here.]

[Figure 464 about here.]

[Figure 465 about here.]

6.5.4 Multicast Transmission

[Figure 466 about here.]

[Figure 467 about here.]

[Figure 468 about here.]

[Figure 469 about here.]

[Figure 470 about here.]

[Figure 471 about here.]

[Figure 472 about here.]

[Figure 473 about here.]

[Figure 474 about here.]

[Figure 475 about here.]

[Figure 476 about here.]

6.5.5 PIM Specifics

[Figure 477 about here.]

[Figure 478 about here.]

[Figure 479 about here.]

[Figure 480 about here.]

Router User to Kernel Mangement

[Figure 481 about here.]

[Figure 482 about here.]

[Figure 483 about here.]

[Figure 484 about here.]

[Figure 485 about here.]

6.5.6 Starting Multicast

[Figure 486 about here.]

6.6 NATs, Tunnels and Other Hacks

These three (NAT, IP in IP and GRE) are very similar - note also that the multicast tunnel support is quite homomorphic too! Network Address Translation is covered further in chapter ten in some more detail.

6.7 Network Address Translation

Network Address Translation is popular wherever one has a shortage of addresses, and is frequently also used as part of simple site security, especially in conjunction with firewalls.

[Figure 487 about here.]

[Figure 488 about here.]

[Figure 489 about here.]

[Figure 490 about here.]

6.8 Tunnels

6.8.1 IP in IP Tunnels

[Figure 491 about here.]

[Figure 492 about here.]

[Figure 493 about here.]

[Figure 494 about here.]

[Figure 495 about here.]

[Figure 496 about here.]

[Figure 497 about here.]

[Figure 498 about here.]

[Figure 499 about here.]

ICMP in IP in IP tunnels

[Figure 500 about here.]

[Figure 501 about here.]

[Figure 502 about here.]

[Figure 503 about here.]

[Figure 504 about here.]

[Figure 505 about here.]

[Figure 506 about here.]

Data in IP in IP Tunnels

[Figure 507 about here.]

[Figure 508 about here.]

[Figure 509 about here.]

[Figure 510 about here.]

[Figure 511 about here.]

[Figure 512 about here.]

Management of IP in IP Tunnel

[Figure 513 about here.]

[Figure 514 about here.]

[Figure 515 about here.]

[Figure 516 about here.]

[Figure 517 about here.]

[Figure 518 about here.]

[Figure 519 about here.]

[Figure 520 about here.]

6.8.2 GRE Tunnels

[Figure 521 about here.]

[Figure 522 about here.]

[Figure 523 about here.]

[Figure 524 about here.]

[Figure 525 about here.]

[Figure 526 about here.]

[Figure 527 about here.]

[Figure 528 about here.]

Receive Data in GRE Tunnels

[Figure 529 about here.]

[Figure 530 about here.]

[Figure 531 about here.]

[Figure 532 about here.]

Transmit Data in GRE Tunnels

[Figure 533 about here.]

[Figure 534 about here.]

[Figure 535 about here.]

[Figure 536 about here.]

[Figure 537 about here.]

[Figure 538 about here.]

ICMP in GRE Tunnels

Care needs to be taken in mapping the causes of ICMP messages when the emanate within or outside a tunnel.

[Figure 539 about here.]

[Figure 540 about here.]

[Figure 541 about here.]

[Figure 542 about here.]

[Figure 543 about here.]

[Figure 544 about here.]

[Figure 545 about here.]

[Figure 546 about here.]

GRE ioctl

[Figure 547 about here.]

[Figure 548 about here.]

[Figure 549 about here.]

[Figure 550 about here.]

GRE Broadcast

[Figure 551 about here.]

[Figure 552 about here.]

[Figure 553 about here.]

GRE Management and Initialisation

[Figure 554 about here.]

[Figure 555 about here.]

[Figure 556 about here.]

[Figure 557 about here.]

[Figure 558 about here.]

[Figure 559 about here.]

6.9 IP version 6

To a large extent, the IPv6 code is modelled on the IPv4 code, so we need only look at the salient places that it is different. IPv6 is both more complex, and simpler. For example, its stateless autoconfiguration means that a lot of associated protocols are not required. On the other hand, this means that this built in functionality is more complex.

6.9.1 IPv6 Header Definitions

[Figure 560 about here.]

IPv6 interface header definitions for INET address family.

[Figure 561 about here.]

[Figure 562 about here.]

[Figure 563 about here.]

[Figure 564 about here.]

[Figure 565 about here.]

[Figure 566 about here.]

[Figure 567 about here.]

[Figure 568 about here.]

6.9.2 IPv6 API

[Figure 569 about here.]

[Figure 570 about here.]

Flow labels can be set by application programmers - using this is described in the IProute2 documentation.

[Figure 571 about here.]

6.9.3 IPv6 Address (Auto)-Configuration

Not surprisingly, the most complex part of IPv6 is the addressing, since that is where it has both more bits, and more ways of allocating them than its ancestor, IPv4.

An IPv6 address can have different scope, such as loopback, link local, site local, and so on.

[Figure 572 about here.]

It can be unicast, or multicast or anycast. It can also be IPv4 compatible.

[Figure 573 about here.]

Addresses need to be allocated when devices come up, and freed up when devices go offline.

When sending a packet, we also need to choose an address carefully!

[Figure 574 about here.]

[Figure 575 about here.]

[Figure 576 about here.]

[Figure 577 about here.]

Deriving an IPv6 address

Multicast address management is not so different from IPv4. There are also addresses for tunnels (for SIT, see later).

Typically we build an address from a LAN interface's MAC address and a route prefix.

[Figure 578 about here.]

The unicast address can start with the LAN device address. It can then have a route based prefix added.

[Figure 579 about here.]

[Figure 580 about here.]

[Figure 581 about here.]

[Figure 582 about here.]

[Figure 583 about here.]

[Figure 584 about here.]

[Figure 585 about here.]

IPv6 Address Management

Addresses are managed through timers, which limit their lifetime.

Addresses can also be managed manually. There are a bunch of parameters that can be controlled by the system:

[Figure 586 about here.]

[Figure 587 about here.]

6.9.4 IPv6 Neighbour Discovery

IPv6 is trying to be as free of manual configuration as possible. Thus it tries to discover from its environment any configuration necessary. One thing that needs finding is neighbours.

[Figure 588 about here.]

Essentially, if a network supports broadcast, then this is used to propagate messages to find out “who is out there”. An example of such a message is the neighbour advertisement. It is like ARP, but uses the ICMPv6 message formats. We’ll just look at this one - others are all similarly constructed.

[Figure 589 about here.]

[Figure 590 about here.]

Neighbour Discovery - Router Advertisement

Inbound messages may include router advertisements, for example:

[Figure 591 about here.]

[Figure 592 about here.]

[Figure 593 about here.]

[Figure 594 about here.]

[Figure 595 about here.]

Other messages include redirects and errors.

6.9.5 IPv6 Input

As with IPv4, IPv6 input is called on demultiplexing a packet (see discussion in chapters three and four). This ends up here:

[Figure 596 about here.]

[Figure 597 about here.]

This ends up calling `ip6_rcv_finish()` via the Network Filter Hook again, which then routes the packet to the appropriate demultiplexing (e.g. forwarding, or local reception to a transport protocol) via the normal pattern:

[Figure 598 about here.]

6.9.6 IPv6 Output

Output is pretty straightforward. As usual we have to go via the Network Filter Hook.

[Figure 599 about here.]

Then the hard work is done in the actual transmit function:

[Figure 600 about here.]

[Figure 601 about here.]

The packet building is a little more complex due to the IPv6 option structures:

[Figure 602 about here.]

[Figure 603 about here.]

[Figure 604 about here.]

[Figure 605 about here.]

[Figure 606 about here.]

[Figure 607 about here.]

6.9.7 IPv6 Forward Function

[Figure 608 about here.]

[Figure 609 about here.]

[Figure 610 about here.]

6.9.8 IPv6 Socket Glue

The socket glue code is slightly different than for IPv4 because the socket options are slightly different as you would expect! Here is an excerpt:

[Figure 611 about here.]

Addressing.

[Figure 612 about here.]

[Figure 613 about here.]

Packet Options.

[Figure 614 about here.]

Multicast parameters

[Figure 615 about here.]

Multicast membership

[Figure 616 about here.]

Path parameters.

[Figure 617 about here.]

6.9.9 IPv6 Flow Label

Flow labels are a unique feature of IPv6 that can be used as fast routing table indexes. They may refer to a session at a higher level (hence the name *flow*), and thus also index flow state such as differentiated or integrated services as discussed in chapter nine.

Here is an excerpt of the flow label management code.

[Figure 618 about here.]

[Figure 619 about here.]

[Figure 620 about here.]

6.9.10 Extension Headers

IPv6 has a number of extension headers - to illustrate what may be present, lets look at the function used to skip them for ICMP, which doesnt want to consider them at all:

[Figure 621 about here.]

[Figure 622 about here.]

6.9.11 Simple Internet Transition

SIT is like a GRE tunnel:

[Figure 623 about here.]

[Figure 624 about here.]

6.9.12 Effect on transport of IPv6 of note

In general, IPv6 has as little impact on transport and above as possible. The TCP and UDP pseudoheaders used for checksums have to be modified since they include parts of the IP header (including addresses). The modules `linux/net/ipv6/tcp_ipv6.c` and `linux/net/ipv6/udp.c` incorporate the relevant changes. These are relatively minor.

6.10 Summary

In this chapter, we looked at the implementation of the Internet infrastructure. The network is built out of links that carry IP packets. IP packets have to be framed for transmission and reception, and so we looked at the Ethernet Local Area Network as an example of this, and covered a specific driver for this. IP addresses need to be translated into link layer addresses, and so we looked at the Address Resolution Protocol. IP packets cause problems, and these must be reported from time to time, and so we looked at the ICMP protocol. Ultimately, there are some things that require additional levels of encapsulation, and so we looked at tunnels. Finally, IP is running out of address space, and so a replacement, IPv6 has been designed and implemented, and so we looked at the Linux implementation of that.

In the next chapter, we look at end-to-end protocols such as TCP and UDP and RTP, which run over this infrastructure.

Chapter 7

Transport

7.1 Roadmap

This chapter talks about end-to-end transfer of data across IP networks. The Internet provides a number of different transport mechanisms, the most common ones are TCP (Transmission Control Protocol) and UDP (User Datagram Protocol)¹. TCP provides a connection-oriented, reliable, stream-based transport and UDP an unreliable connectionless datagram-based one. RTP is a newer protocol designed for the transport of Real Time Streams, such as video and audio. In this chapter we will examine the workings of UDP, TCP and RTP, and their implementations in Linux.

The services transport protocols have to provide cover a wide range, including:

Multiplexing

Reliability

Flow Control

Congestion Control

Framing

Synchronisation

This chapter covers the UDP protocol, by analysis of its use in the anatomy of a DNS lookup, and the TCP protocol, including the anatomy of flow providing service to HTTP, for a single web page download; We also take a brief look at RTP.

7.2 Introduction

Transport protocols handle the end-to-end transfer of data between two IP-connected hosts. Each packet created by the transport layer is addressed to the

¹As with most acronyms on the Internet the abbreviation is more important and well-known than the expansion

destination host, and transported by the network layer. Higher layer protocols and services will make use of transport layer to transport their protocols. Recall from chapter XXX that the network layer (and below) devices, such as switches and routers, store no state about the connections established or flows traversing through them. So any state information needed to maintain communication must be stored by the end-system hosts.

7.3 UDP—User Datagram Protocol

UDP is much more simple than TCP so we will discuss its operation first.

7.3.1 Introduction

UDP is a connectionless, unreliable datagram service. This means that there is no connection information maintained and therefore each packet carries its own addressing information. By unreliable we mean:

- that there is no guarantee that the data will be received by the destination process,
- that if it is received that it may contain errors.
- that packets may arrive out-of-sequence, i.e. in a different order to the one with which they were transmitted.

UDP provides a multiplexing service over the basic IP functionality, and a simple form of error detection.

7.3.2 Multiplexing

Although most people will refer to hosts communicating over a network (such as the Internet), in reality it is software processes that communicate. Devices in the Network layer (such as routers) will use the IP address information to route the packets to hosts. The host must then process the packet and eventually route the data carried to the appropriate software process. In the same way that a sailing ship needs to know both the country of its destination and the port of disembarkation, a port is used to identify the process that the data is destined. Ports are transport layer information. In the TCP/IP suite port numbers are 16 bit quantities.

Many hosts provide a set of server software processes such as web servers. These servers listen for potential data packets or connections on so-called *well-known* ports. A mapping between service names and port numbers is in the `/etc/services` file. A sample is shown in figure 625. Each line consists of a service name such as `ftp`, followed by a port number and protocol

[Figure 625 about here.]

7.3.3 UDP packet header anatomy

The UDP datagram contains an 8-byte header and is carried within an IP datagram. When carrying UDP data the IP protocol field is set to 17. The UDP header is shown in figure ??.

[Figure 626 about here.]

The only transport layer facilities that UDP provides over and above that of the IP datagram layer is that of multiplexing and a checksum for rudimentary error checking.

Not only does UDP offer the disadvantage of no reliability in transmission, but also it cannot adapt to the network conditions. If the network is too highly loaded for the communication to take place the UDP connection will be poor, and also have an adverse effect on other traffic on the network. The TCP transport described in the later in this chapter provides this extra functionality.

7.3.4 Transport Layer linkage

This section describes the functions that link the transport layer to the socket layer above and the network layer below.

Socket to Transport Layer

A socket's life is one of create, operate on and close. This section describes how the user space calls are passed through to kernel functions. We will use the example of socket creation to show this.

Socket state information is maintained in a number of structures:

- `struct socket`
 , which is a protocol independent structure.
- `struct sock`
 , which contains transport layer information. In most of the functions, the current sockets `sock` structure is called `sk`

The main socket code is in `net/socket.c`

. Functions in this file are coded to call functions specific to the type of socket created.

When a socket is created (`socket` call) there is a call to `sys_socket`

. This calls `sock_create`

and then maps the socket to a file descriptor. This descriptor is passed back to the user process for future operations on the socket. `sock_create`

allocates the socket and then calls the create function for the family of sockets being created (in the case of an IP network socket `PF_INET`

. Here reference is made to the `net_families`

array, so we now need to travel to the initialisation code for the linux networking in order to see how this and other structures are initialised.

The `net_families`

array is initialised when the kernel is booted in the `inet_init` function in `net/ipv4/af_inet.c`

. This function is part of the code run at boot up as can be seen from the printed message at the start of the function. One of the first operations in `inet_init`

is to register the inet family of sockets with the socket functionality. Then the protocols are added to a hash table of protocols. These functions are in `protocol.c`

.
The inet family is registered with the `sock_register` call, which is a function in `socket.c`, which stores the family operations structure. This structure contains the family name and a create function for the socket family—`inet_create`

The create function for PF_INET

sockets is `inet_create`

, which is the function called as `net_families[family]->create` in `sock_create`

. `inet_create`

allocates and initialises the sock structure. An important part of the initialisation is the assignment of the socket operations structure to the `ops`

field. The two main types of IP socket (`SOCK_STREAM`

and `SOCK_DGRAM`

) each have a set of functions associated with their operations. These are

`inet_stream_ops`

and `inet_dgram_ops`

, which are statically initialised in `net/ipv4/af_inet.c`

.

A further important structure to be initialised is the `prot`

structure which contains specific functions for each protocol type. `udp_prot`

and `tcp_prot`

are static structures in `net/ipv4/udp.c`

and `net/ipv4/tcp_ipv4.c`

. This use of sets of function pointers allows for a great use of generic code in the the socket and inet functions. When the socket is created there is a call to `prot->init`

to initialise the socket.

To summarise the above on startup the protocol family (`PF_INET`

) is first registered. Each socket created has a set of socket operations (either

`inet_stream_ops`

or `inet_dgram_ops`

). Each protocol, tcp or udp, has a set of protocol functions (`tcp_prot`

or `udp_prot`

). Throughout the code calls are made to these functions.

The first operation on most sockets after creation will be a connect call. This appears in the kernel as `sys_connect`

in `net/ipv4/socket.c`

. Here the socket is found (by looking up its file descriptor) and the memory block containing the address this socket is to be connected to is moved to the kernel. Then the the `connect`

function out of `sock->ops`

is called. For udp this will be `inet_dgram_connect`

(in `af_inet.c`

), which itself calls `connect`

in the `sk->prot`

structure, which for `udp` will be `udp_connect`. Other socket operations are passed through to the relevant function in this way.

[Figure 627 about here.]

[Figure 628 about here.]

IP Layer Linkage

The `af_specific` structure contains functions which pass data to and from the network layer. This is set to `ipv4_specific` when the socket is initialised. See figure 10.6.

[Figure 629 about here.]

7.3.5 The `sock` structure

This structure holds information relevant to a particular socket. This information includes values such as the port numbers, destination address etc. It is declared in `linux/include/net/sock.h`

, see figure 628 for the part of the structure that stores the source and destination addresses and ports. Whenever a socket option is changed or queried then the kernel stores the value of the option in this structure. The `tp_pinfo` union contains a `struct tcp_opt` which itself contains important tcp information, which we will cover in the TCP section below.

Socket structures are allocated by the `sk_alloc` call. Each socket maintains a reference count. When this reaches zero the socket can be freed from memory (`sk_free`). The `sock_hold` and `sock_put` increment and decrement the reference counts. These are implemented in `linux/net/sock.c` and `linux/net/sock.h`.

[Figure 630 about here.]

7.3.6 DNS look-up

The Domain Name Service is responsible for converting names to addresses on the Internet. The actual service is not described in this chapter, but we will use a DNS look-up as an example of an application that uses a UDP transport. In C the function `gethostbyname`

is the programming interface to this protocol. If we take a simple program that performs a look-up and simply outputs it to the screen we observe the strace output in figure 629. This output has been edited to only show the information appropriate to the DNS request and response. Here a call to `socket`

creates a datagram (i.e. UDP) socket and connects it to the DNS server. As UDP is a datagram based protocol, this connect simply updates structures in the operating system rather than generating network traffic. An datagrams sent to this socket that are unaddressed will be forwarded using this connection information. A DNS packet is then formed and sent. The program must now wait for a reply and also be ready to timeout if a reply is not forthcoming. Eventually a message is available and the `recvfrom` call allows it to be read. The socket is then closed.

[Figure 631 about here.]

We can now trace the path of this request through the Linux kernel to the IP layer and then the response from the IP layer back to the application.

The socket is first created and then connected (`udp_connect`, see 630), which as this is a datagram socket doesn't result in any traffic across the network, but a modification of the kernel structures. Specifically a routing table entry is created for this destination address and this entry added to the socket structure.

The application library now builds a DNS request packet and passes this to the kernel with the `sendmsg` call. This is passed on by `socket.c`

```
to udp_send
in net/ipv4/udp.c
```

, figure. One of the operations of `sendmsg`

is the calculation of the UDP checksum. This checksum provides for some rudimentary error checking at the receiver and is calculated over the entire udp packet and also some fields from the IP layer (a fake header) see figure ???. The checksum value is the one's complement of the sum of all the 16 bit words to be checked. This is inserted into the checksum field. At the receiver, the stack can simply add the values together and an all ones indicates a correct packet. The packet is passed to the datagram (IP) layer via the `ip_build_xmit()`

```
call. Note that the call to ip_build_xmit()
```

call's second parameter is a function pointer. This pointer can take one of two values depending on whether the packet's checksum is to be calculated or not. In either case the purpose of this function is to copy the data from userspace to the IP layer directly and if necessary calculate the checksum as the data are copied. A further optimisation here would be to copy the data from user space directly to the interface adaptor card. Figure 632 shows the building of the udp header and the eventual transmission of the packet.

The request traverses the Internet to one or many servers and eventually a response is received. This is processed by the infrastructure protocols and buffered. The packet arrives at the udp layer via a call to `udp_rcv`

```
, this function is the handler function stored in the inet_protocol
structure, see chapter ??. udp_rcv
```

```
first validates the packet (udp_checksum_init
```

). If this check is passed and the packet is a multicast or broadcast packet then it is delivered using `udp_v4_mcast_deliver`

```
. Otherwise the packet's socket is searched for (udp_v4_lookup
```

```
). This function is shown in figure 633
```

If the socket is found then the data is queued. If a socket is not found then this could be due either to an errored packet, or an attempted communication to a port without a bound process. If the former case the checksum failure

will cause the packet to be silently dropped and in the latter an ICMP port unreachable message is returned to the sender, figure 634

UDP (and TCP) checksums are performed over the entire segment and an additional pseudo-header, which includes some IP layer fields (addresses, protocol type, length etc). This provides some extra integrity if the IP checksum has failed to catch errored packets. For reasons of performance the checksums are calculated as the packets are copied from user space to the kernel, see `linux/net/core/iov.c`

When the application is ready to receive a message it calls either `recv`, `recvfrom` or `recvmsg` which arrive in `udp.c` as a call `udp_recvmsg()`

. Here if a datagram is available it is taken from the datagram layer (`skb_recv_datagram`

) and copied to user space. As with `transmit` a checksum calculation is done as the data are copied and if this fails then an error returned. After this the source address of the the datagram is placed into a `sockaddr`

structure for copying to the parameter of a `recvfrom` or `recvmsg` call.

When the application has finished using the socket `close` will be called, this passes to the kernel as `sock_release`

, which then calls the release operation for the protocol—`inet_release`

. This in turn calls the close operation—`udp_close`

, which in turn calls `inet_sock_release`

. Here the resources used by the socket begin to be released back to the system. A point to note is that although this machine has finished with the socket and its communication, there may be packets in the network still arriving. The system is left in a state to handle these.

[Figure 632 about here.]

[Figure 633 about here.]

[Figure 634 about here.]

[Figure 635 about here.]

[Figure 636 about here.]

7.4 TCP—Transmission Control Protocol

In addition to multiplexing TCP provides other transport layer services to the application layer. These are reliability, flow control and congestion control. These are described here and followed by a description of the TCP packet header and the stack implementation in Linux.

7.4.1 TCP Transport Functionality

Reliability

The data are sent and received by the application arrive in exactly the same order, none are lost, none are repeated and none are subject to errors in transmission. These aims are achieved in a number of ways. Firstly, each segment is

checksummed in the same way as the UDP checksum described above. If a packet fails the checksum test then it is dropped and treated by the receiving stack as a loss. Secondly, each segment transmitted is subject to an acknowledgement. If the acknowledgement does not arrive after a predetermined time interval then the segment may be retransmitted. The calculation of the optimum timeout interval is important for efficiency and the estimator functionality is described later in this chapter. Thirdly, each packet is given a sequence number. If two packets arrive with the same value then a duplicate has been detected and the second may be discarded. Sequence numbering also provides for detection of out-of-order arrivals and allows the receiving stack to reorder these saving on retransmissions.

Connection Oriented

As the IP layer is a simple datagram only layer TCP adds the concept of a connection at the transport layer. There is a handshake between the sender and the receiver at the start of a communication in order for each end to agree on whether they wish to communicate and also some specific communication parameters.

TCP is also a streaming protocol rather than the datagram based UDP. This means that two processes communicating with TCP are connected simply by a pipe, where data are sent in one end and received out of the other. There is no relationship between the size of the data blocks sent into this pipe, the size of the datablocks read from the pipe and the size of the data blocks as they are sent over the network.

Flow Control

In the simple case the sender could wait for every packet sent to be acknowledged or timeout before sending the next packet. This, while being relatively easy to implement, is rather inefficient on bandwidth use.

For example a 100Mbps pipe between two geographically separate endpoints with a round trip time of say 5ms, will only allow one new packet every 5ms, if an acknowledgement has to be awaited between packets. For 1000byte packets this will be an effective bandwidth use of only $\frac{(1000 \times 8)}{0.005}$ bps, which is 1.6Mbps, i.e. 1.6% of the available bandwidth.

On the other hand, a sending stack cannot simply send the data as fast as possible or else there will be problems in both the network and the receiving system. If a fast sender is sending to a slow receiver then the data may all arrive, but not be processed fast enough. This can lead to the situation where data are completing their journey across the network, but being dropped at their destination. In order to prevent this TCP uses *flow control*.

Flow control employs a mechanism called a sliding window, see figure 635. All data falls into one of the following categories:

- data that have been sent and acknowledged,
- data that have been sent but not acknowledged,
- data that can be sent but haven't been yet,
- data that can't yet be sent.

The data in the center two categories represent those data that the receiver is willing to receive. These data are encompassed by a window—the sliding window. As acknowledgment are received for data at the left hand edge of the window the whole window can slide to the right bringing further data into the category of data that can be sent.

[Figure 637 about here.]

Congestion Control

In the discussion of flow control above we recall that there can be network related problems to senders sending too fast. If a sender with a high bandwidth connection to its network is sending to a receiver, and at some point in the path there is a lower bandwidth link, then as soon as the sender starts sending too fast for the lower bandwidth link there will be loss. This is called congestion. Congestion can also occur when a number of connections are sharing a link and each is trying to make as much use as possible of the available bandwidth. TCP congestion control makes sure that a TCP connection in either of the two above cases receives a fair share of the available bandwidth. Definition of fair in these cases is generally a contentious issue.

Congestion control employs another window in the data to be transmitted. This window is called a congestion window and controls to the amount of data that can be sent into the network unacknowledged. The congestion window increases in size while the network is uncongested and decreases when congestion is detected. Currently congestion is detected by detection of a lost packet. Explicit Congestion Notification (ECN) is a technique where congestion in the network is detected and the sending sites informed by setting of a TCP header bit in the returning data stream.

The increases and decreases in the congestion window size are specified in RFC 2581. A TCP connection can be in one of two states: slow start or congestion avoidance. In slow start, the Congestion Window increases in size by doubling after an amount of data equal to the congestion window's size has been sent, until a threshold is reached. After this, congestion avoidance is entered when the congestion window increases by the value of the MSS. If at any point loss is detected then the threshold is set to half the current congestion window size and the congestion window size set to the MSS. The connection returns to slow start. This is TCP TAHOE. TCP RENO is an enhancement where at the onset of congestion the connection reduces the threshold as before, but sets the congestion window to the threshold and enters congestion avoidance.

7.4.2 HTTP request

We will use the example of an HTTP request to illustrate the operations of TCP in the Linux protocol stack. The request is a single html page from a local web server. Figure 636 shows an edited version of the strace output from the web browser (`lynx`

in this case). Figure 637 shows the output from `tcpflow`, which is a program that prints out application layer data from snooped packets. Figure 10.6 shows the output from `tcpdump` for this operation.

[Figure 638 about here.]

[Figure 639 about here.]

[Figure 640 about here.]

In order to see the strace of execution on a server process we will employ the `nc`

program. First we set up `nc` as a `tcp` listener on port `5656` and connect to it using a `telnet` session. The session is then closed down. The resultant strace on the `nc` session is shown in figure 638

[Figure 641 about here.]

Examining figure 10.6 closely we can see that the first three packets are the connection setup 3-way handshake, the next four are the data transfer, one packet in each direction and their acknowledgments and the final three the connection close down. Note that the acknowledgment of the first `FIN` packet is piggybacked onto the client's `FIN`.

From the strace output in 636, we can see that the sequence of significant operations on the socket is `connect`, `write`, `read` and `close` (the socket will already have been created by this point).

On the server side (from figure 638) the significant operations are `socket`, `bind`, `listen`, `accept`, `read` and `close`. Most servers will also write in response to a `read`.

7.4.3 TCP header

The `TCP` header has a large number of fields and is shown in figure 639

[Figure 642 about here.]

7.4.4 TCP implementation

The `TCP` stack has to respond to a number of external and internally generated events. These are:

User requests —These come through the socket layer as `connect`, `send`, `recv`, `close` (and other) calls.

Network Events —These come from the network layer as datagrams arrive.

Time Events —These come via interrupts from timers running in the kernel.

files `linux/net/ipv4/tcp_*.c`

Unlike `UDP`, `TCP` needs to maintain state about each connection. With `UDP` the message to be sent passes through the kernel into a single (possibly fragmented) datagram on its own. With `TCP`, because reliability, flow and congestion control need to be maintained then the data have to be buffered and information concerning the current state of the connection stored. This is done in the process control block.

7.4.5 TCP states

TCP can be in a number of states, these are listed in 640. The process of connection setup and tear down is controlled as in the State Transition Diagram shown in figure 641. This is copied directly from RFC 793, which details the transition between these states.

[Figure 643 about here.]

[Figure 644 about here.]

7.4.6 TCP Structures

These are all coordinated per TCP connection through the Transmission Control Block data structure. The values are described in RFC793, and key values will be illustrated in the text below. The `sock`

structure is a large structure where these values are held. This contains a number of other structures, including a `struct tcp_opt`

called `af_tcp`

in the union `tp_pinfo`

. This structure is usually referred to as `tp`

, by an assignment at the start of each function that uses it, see figurefig:tp

[Figure 645 about here.]

[Figure 646 about here.]

[Figure 647 about here.]

[Figure 648 about here.]

[Figure 649 about here.]

[Figure 650 about here.]

The `tcp_skb_cb`

structure contains information about the current TCP connection on a per segment basis. Access to this structure is via the `TCP_SKB_CB`

. See figure ??

[Figure 651 about here.]

7.4.7 TCP Code

This section takes a walk through of the salient features of the TCP stack. The order is the intuitive of the typical TCP operations rather than architectural or procedural. We initially look at connection setup, from a server and client perspective and the demultiplexing of individual segments to their socket owners. Then we go on to cover tear down, which will complete a look at the state machine in the previous figures. The next stage to examine is the sending of data, the actions taken on receiving segments and what to do in the event of timeouts. Finally we look at other features of TCP, including congestion control, PAWS (Protection against Wrapped Sequence numbers), window scaling and SACK (Selective Acknowledgments).

Connection Setup

The client side of connection setup causes a call to `tcp_connect`

. Here various fields in the `tcp_opt`

structure are initialised and an socket buffer for the SYN segment filled.

This is transmitted at the end of the function.

Meanwhile the server has created a socket, bound it to a port (`bind`) and has called `listen` to wait for a connection. This changes the state of the socket to listening. The `tcp_rcv_state_process`

function processes the incoming data.

When the server replies with its own SYNACK packet this will be processed by `tcp_rcv_synsent_state_process`

TCP demultiplexing

The socket structures for listening and established connections are stored for searching in the `tcp_hashinfo`

structure. This is searched on the arrival of a segment to determine which socket is required. This search which is done by `__tcp_v4_lookup`

(and other functions with similar names) is for established connections based on a has value of the source and destination addresses and ports—`tcp_hashfn`

. For listening connections the hash value, calculated by `tcp_lhashfn`

is simply based on the listening port number.

Send and Receive

The application sends data by writing to the socket. This passes through the socket linkage and becomes a call to `tcp_sendmsg`

in `linux/net/tcp.c`

, see figure 649 The current MSS is determined. The data to be send is within the `msg`

structure and assigned to `iov`

, which is an array of `iovlen`

blocks. If the socket is not in an errored state or a half shutdown state then transmission can begin.

[Figure 652 about here.]

[Figure 653 about here.]

We now loop though the message blocks and build segments. As TCP is byte oriented the blocks can be divided and joined in order to achieve full segments where possible. First the socket buffer at the end of the write queue is examined. If this doesn't exist, (say because it has been transmitted and acknowledged) or it is full, then a new socket buffer is allocated. If there is room then the space left is calculated. By the end of figure 651 `copy`

contains the number of bytes that can be added to this `sk_buff`

.

[Figure 654 about here.]

This data now needs to be copied into the socket buffer, some memory may need to be found. Data is copied from the user using the `csum_and_copy_from_user`

function, called either from `skb_add_data()`
or `tcp_copy_to_page()`
, figure /reffig:sendloopmemory1- 653.

[Figure 655 about here.]

[Figure 656 about here.]

The rest if the function is shown in figure ??- ?? . If the length of the data in this socket buffer is still not as big as the MSS then the loop is continued.

[Figure 657 about here.]

[Figure 658 about here.]

The socket buffer are sent to the network layer with `tcp_transmit_skb`
, which at this stage is ether called via `tcp_push_one`
or `__tcp_push_pending_frames`
and `tcp_write_xmit`

. This latter function attempts to send each packet in the send queue, see ??.

Prior to transmission of a packet there is a call to `tcp_snd_test`

, see figure 657. Note that `TCP_SKB_CB(skb)->end_seq`
is the sequence number of the last byte in the socket buffer, `tp->snd_una`
is the next expected acknowledgment number and `tp->snd_wnd`
is the current advertised receive window from the other end of the connection.

This is the sliding window protocol in action.

[Figure 659 about here.]

[Figure 660 about here.]

`tcp_transmit_skb`

is the function that builds the TCP header including any option fields, eventually checksumming the packet via the `tp->af_specific->send_check()`

call, which maps to `tcp_v4_send_check`

, and transmitting the buffer with `tp->af_specific->queue_xmit(skb)`

, which is actually `ip_queue_xmit`

. Note that a clone of the buffer passed which means that the original is saved in the case of a retransmission being required.

[Figure 661 about here.]

[Figure 662 about here.]

[Figure 663 about here.]

[Figure 664 about here.]

We now consider the receiving of data at the other end of the connection. Packets are received by the IP layer and forwarded to the TCP layer via the `tcp_v4_rcv`

, which is part of the `net_family` structure for TCP. See figure 662– ??.

[Figure 665 about here.]

[Figure 666 about here.]

First the packet's header information (from `th->`) is copied to the socket buffers `TCP_SKB_CB` block. The the packet's socket is found, see section 7.4.7. Now the socket buffer is either added to the socket backlog, for future processing, or passed to `tcp_v4_do_rcv`

. The `fig:tcppreque` provides for an optimisation if the current user process is the one owning the socket, see ??

[Figure 667 about here.]

`tcp_v4_do_rcv`, see figure ?? in the connection setup section, fast paths if the socket is in an established state to `tcp_rcv_established`, this being the most common state, and also that which requires fastest processing.

[Figure 668 about here.]

[Figure 669 about here.]

[Figure 670 about here.]

[Figure 671 about here.]

[Figure 672 about here.]

The fast path in this function uses the header prediction in RFC 1323, i.e. is the expected sequence number the same as that of the incoming packet. Here the timestamp option if present is decoded, PAWS (see later) checked. If the packet is empty, then we call `tcp_ack`

. If it is too small then it is discarded and if it contains data then it is queued for the owning socket. There is another optimisation here if the current process is the task that owns the socket, as the data can be copied directly to the `iovec`. The checksum is checked in this function, and the `tcp->rcv_nxt` value updated.

In the slow path first PAWS is checked (see later). Then whether the packet is in sequence is checked with the `tcp_sequence` function, figure 670 and 671.

[Figure 673 about here.]

[Figure 674 about here.]

If an out of sequence packet occurs then a duplicate ACK is sent, with a call to `tcp_send_dupack()`

- . If the successful case a call to `tcp_ack_snd_check`
- (and `__tcp_ack_snd_check`
-) will send an immediate ack or a schedule a delayed ack. Figure 672.

[Figure 675 about here.]

When an acknowledgement packet is received a call is made to `tcp_ack`, see 673 and 674.

[Figure 676 about here.]

[Figure 677 about here.]

At the end of `tcp_rcv_established`
`tcp_data_queue`

is called. This is show in figures 675– 678. Here the fast path is the most common packet, i.e. in sequence and in the current window. Again there is an optimisation if the current task is that owning the socket, otherwise memory must be reserved and the socket buffer queued. The `rcv_nxt`

variable can be updated for the current connection and if the socket buffer contained some data then `tcp_event_data_rcv`

is called to schedule an ack. If there are packets on the out-of-order queue then this segment may have filled a gap, in which case an immediate ack should be scheduled. Eventually the socket buffer can be freed if the data has already been copied (i.e. `eaten > 0`

) otherwise the `sk->data_ready`

is called which causes the task owning the socket to awake.

The next fastest path is receipt of a retransmitted packet, which is indicated to the sender and then dropped.

The next fastest is a zero window probe, see later.

The remaining part of the function deals with out-of-order arrivals.

[Figure 678 about here.]

[Figure 679 about here.]

[Figure 680 about here.]

[Figure 681 about here.]

TCP is generally driven by receiving acks from the far end of the connection or data from the user. However, there will be situations where timers are needed to keep things going. There are three main timers in TCP

- The retransmission timer—used if a packet is unacknowledged to cause a retransmission, this is also used if the receive window from the other end is zero to keep probing and give the opportunity for the window to grow again.
- The delayed ack timer—used to send a delayed ack, if it hasn't been sent for another reason.

- The keep alive timer—used to keep a very low level of traffic over an idle connection.

These functions are in `tcp_timer.c`

. See figure 679

[Figure 682 about here.]

Congestion Control

The sender's congestion window is stored in the `tcp_opt`

structure as `snd_cwnd`

. This is initialised in `tcp_init_metrics`

either on receipt of a SYN segment (for a host receiving a connection) in

`tcp_rcv_state_process`

, or on acknowledgment of a sent SYN segment (for a host initiating a

connection) in `tcp_rcv_state_process`

.

The congestion window is checked, along with other things, as a buffer is sent see ??.

If the connection window is smaller than the slow start threshold (`ssthresh`) the connection is in “slow start” state. If larger then it is in “Congestion Avoidance”. The size of the congestion window increases in different ways depending on this state. Figure 680 shows the increase in the congestion window, this is called once per ack from `tcp_ack`

.

[Figure 683 about here.]

Selective Acknowledgement—SACK

Modern tcp/ip implementations such as the one described in this book employ a selective acknowledgment mechanism known as SACK RFC2018. Sack employs a tcp options field to indicate received segments that cannot be simply acked, as there would be gaps in the acked stream. The `tcp_opt`

structure uses the following variables in SACK:

- `eff_sacks`
to indicate the number of sacks to send with next packet.
- `selective_acks`
an array of sack blocks (each sack block is simply the sequence number of the beginning and end of the block)
- `dsack`
to indicate a duplicate sack should be sent
- `duplicate_sack`
information about this D-SACK.

These are set by the tcp receive data functions, figure 681– 683 and used to add options to generated packets figure 684

[Figure 684 about here.]

[Figure 685 about here.]

[Figure 686 about here.]

[Figure 687 about here.]

When a SACK option comes in the relevant socket buffer is marked as received and not retransmitted. A receiver still has to option to drop the packet (an act known as renegeing), see figure 685

[Figure 688 about here.]

When a segment is acknowledged by the usual tcp acknowledgement mechanism the fact that they have been sacked need no longer be recorded.

Protection against wrapped sequence numbers—PAWS

This is an high performance extension to TCP described in RFC 1323. On high bandwidth long distance links there is a danger that the sequence count will wrap round while there are still valid older segments in the network. The PAWS algorithm prevents this by using a timestamp option, which carries an non-decreasing integer. The basic idea is that a segment can be discarded if its timestamp is less than some other recently received timestamp. The following variables in `tcp_opt`

are used:

- `rcv_tsval`
The timestamp in this segment
- `ts_recent`
The most recently received timestamp
- `ts_recent_stamp`
The time `ts_recent` became valid

[Figure 689 about here.]

Round trip estimation

In order to get the most out of a TCP connection in terms of bandwidth utilisation, an accurate method to determine the round trip time for timeout purposes is needed. The mechanism is described in [8], and shown implemented in figure 687– 688

[Figure 690 about here.]

[Figure 691 about here.]

[Figure 692 about here.]

Window Scaling

The TCP header allows for 16 bits to indicate the received window size. With ever increasing bandwidth and distance this is no longer sufficient to keep efficiency. The TCP window scale option on synchronization allows for this value to be scaled. This is described in RFC1323. The option is sent only in SYN packets, as it is only these that can be guaranteed to be carried to the receiver. See figure 690 and figure 691.

[Figure 693 about here.]

[Figure 694 about here.]

7.4.8 TCP behaviour

Looking at some detail at TCP, latest theory and measurement by Padhye et al. (1998), shows how it really behaves.

[Table 15 about here.]

$$B = \min\left(\frac{W_m}{RTT}, \frac{1}{RTT\sqrt{\frac{2bp}{3}} + T_0\min(1, 3\sqrt{\frac{3bp}{8}})p(1 + 32p^2)}\right) \quad (7.1)$$

This equation is for long lived transfers. There is more recent work by Cardwell and others on short lived transfers (which the majority of the Web usage consists of today). However, just looking at this equation for now, we can see a lot of single instance specific parameters which affect the performance radically. For example W_m , and the range of T_0 and b in implementations could be quite large. Another problem here which we should consider when thinking about pricing is the range of values for RTT - in some theoretical work, RTT is used as a measure of resource use. However, it is actually a real value measured from the path delay, and as such, is suspect when used for comparability of e.g. satellite versus terrestrial hops. Another problem with such proposals is that the loss rate p is considered as congestion feedback, and therefore as a proportional resource utilisation indication. However, it depends on the instantaneous load in the queue seen by a single packet, not the average over some period. Loss is also due to interference on wireless networks, and so is a very noisy “signal”. Proposed Internet replacements for loss such as Explicit Congestion Notification also need specifying with regard to the sampling interval: if a switch measures congestion over some interval, this must be known to the sources that receive congestion feedback, out of band, so that they can estimate the significance of some number of packets arriving with the ECN bit set.

Reliable multicast protocols (for games, share dealing and software and news distribution) such as the family of protocols using Handley and Floyd’s “TCP Friendly Multicast Congestion Control” use similar adaption techniques to that which TCP employs. However, one additional level of complexity is that these protocols also have self-organising repair server mechanisms. These will alter the traffic patterns again.

Similar evolution is occurring in the streamed multimedia traffic flows that emanate from Internet Radio and TV sites, with TCP-like adaption being the apparent goal for Real Audio and Video, the most commonly used product for such services in the Internet.

7.5 Management

The following commands on linux can be used to monitor the transport layer:

- Netstat - can display active connections, for example `netstat -a` will display all active and listening connections.
- Tcpdump - can trace packets on wire by putting system into *promiscuous* mode at driver level (says “recognize all frames as *for me*”) and passing up through a filter to an application which can further process or dump to a screen.
- ethereal (www.ethereal.com) will provide a graphical display of network traffic.

7.6 TCP vs UDP

As we have seen TCP overcomes many of the problems with UDP, but at what cost. TCP is a much more heavyweight protocol, than UDP. Indeed it can be seen simply from the number of pages of this book how much more effort is needed in its description and implementation. However, there are some circumstances where UDP is the better transport mechanism. These include situations where memory and resources are limited such as initial boot-off-network operations on diskless workstations where a full TFTP (Trivial File Transfer Protocol) implementation, which runs over UDP, can be implemented in a few bytes of ROM code and minimal RAM requirements. Other times where use of UDP may benefit the situation are when the lightwightness can provide a performance advantage. An example here is DNS over a LAN, when a quick name resolution is required and in a LAN environment there is often enough resources to be relatively assured of a successful communication. UDP can also be used when another mechanism is responsible for portioning out the bandwidth for example communication over RSVP established sessions. The Real-Time-Protocol (RTP) is a protocol designed to carry real-time traffic, such as video and audio, over the Internet. It uses UDP as its transport mechanism.

7.7 RTP—Real Time Protocol

The Realtime protocol is designed to carry multimedia flows over the Internet. Whether RTP is a transport layer protocol or an application layer protocol is a commonly debated point. For the argument that RTP is application layer are: RTP data packets are carried in UDP segments (a transport layer service), and RTP is usually implemented in user space. For the contrary argument are the fact that RTP provides a number of additional services to application layer, these are synchronisation and framing. RFC1889 describes RTP and a number of other rfc's describe different payload formats for particular media types. RTP is generally implemented as a user library and not in the kernel so is beyond the scope of this work.

7.8 Writing a new Transport Layer

If you are developing a new transport layer which uses IP as its networking layer and sockets as the application API, then linking it with the existing protocol suites becomes simply a matter of writing the appropriate linkage functions and filling in the function holding structures. see section 7.3.4

7.9 Summary

This chapter examined transport protocols and their implementation. The facilities provided by the transport layer, i.e. Multiplexing, Reliability, Flow and Congestion Control, Synchronisation, Framing, have each been examined and their implementations detailed. A comparison of popular transport protocols: TCP, UDP and RTP has been made and instructions on writing your own transport layer given.

Chapter 8

Routing

“I’m not sure that the ability to create routing diagrams similar to pretzels with mad cow disease is actually a marketable skill.” - Steve Levin

8.1 Roadmap

This chapter is about Internet plumbing and path finding. Here we discuss how we interconnect links of different types and, having done so, how we find our way from A to B across a complex topology. Linux supports a variety of routing protocols, and Linux systems, running on PC or other hardware platforms, are in widespread use as low cost solutions for low to medium performance routers (though not entirely modest - up to 100baseT local, and OC-3 wide area links are easily supported, and there are no obvious reasons not to build up to a Gbps router!).

There are two main components to a router: Forwarding and Routing. The forwarding task is carried out on each and every packet and involves looking up the destination (and possibly other data from the packet) in a forwarding table, and using the result to decide which interface to transmit the packet on. This is a kernel procedure. The routing task involves the business of building and maintaining the forwarding table, and is carried out asynchronously, usually by a user space process. There are a variety of routing protocols which are used in the internet, but the basic tasks are the same.

In this chapter, we review the components of the routing system and then look at Linux kernel support for the forwarding information base, and control of it from routing protocols in user space.

8.2 Introduction

An IP routing system takes the work of finding a path away from end systems. As with the postal service, end users and systems need only know the location (address) of a recipient to get information there. An end system discovers an address by being given it. Typically, an end user uses a DNS name, which is looked up in the name server system, and results in the application software being given an IP address. The end system software then uses this address as the handle for communication with the “far end”. The infrastructure of the

Internet Protocol layer, in end systems routers and links, conspire to deliver the packets to the right place, based on the destination address in IP packets. The routing system may optionally consult other fields in the IP header to provide enhanced services.

To do all this, routers have to consult maps. The maps have to be constructed somehow, and there are a variety of approaches to doing this.

Forwarding Decision Frequency The forwarding decision is taken most often. We can decide which route a packet will take separately for each packet, or we might fix our decision for a number of packets in a row, or even for a number of packets for a given user, perhaps for some configured amount of time. Whichever way a forwarding system worked, it seems that we do not want to carry out any complex calculation, and we especially want to avoid any exchange of information with any other routers just so as to be able to forward one packet.

Information about the world cannot be distributed infinitely quickly. In a physical transport system (e.g. postal service or freeway system for cars) where information can be sent about remote conditions by radio, in a communications network, the state of a link some way away can only be learned as quickly as the users' own data can travel. In other words, our information about the rest of the world is out of date. This means that it does not make sense to change what route we use too often. At one extreme, we can fix the route taken by packets manually, via some configuration tool, or through network management. In fact, it is quite common to do this effectively in parts of the network where there are no choices anyhow - this is called setting up a default route. As one looks "further into the network", this does not make sense, since there are alternative possible paths through the network, and it would seem a shame not to be able to use them if our main (default) choice link or path fails. Thus it is usual to configure fixed, or even default routes only on end-systems or non multi-homed (see later) site access routers.

Routing Calculation Frequency We could re-calculate routes every time there was a new flow (e.g. a new TCP session, or a new Web download). This might make sense if flows lasted a long time, since we might also be able to carry out some clever load balancing, provided we were able to tell the "rest of the world" about our new flow, and which paths we were considering for it. In practice, on the Internet flows are short lived, so most current routing protocols do not consider re-calculating routes simply based on short term information. It is more common to carry out a routing calculation based on one of two factors:

1. topology change - a link outage or repair, or new link installation triggers an update. Of course it may be necessary to dampen down the trigger if a link has an intermittent status, to avoid putting too much traffic load (and computational effort) onto other routers.
2. long term traffic change - a long term traffic monitoring system may input data into the route computation - currently, this is regarded as somewhat of a black art (also known as a "research topic").

It is also usual to have periodic exchanges (“soft state” refreshes) to make sure that everyone has a consistent view of the world.

Routing Update Data Sources The input data for a routing calculation can be delivered from a central monitoring system to each router, or from the router’s neighbours, hop-by-hop, or may be distributed by each router flooding information to all the other routers in an area. Each approach has its advantages. Central systems may fail, depriving the routers of updates, but a centralised system takes this load off the routers. A hop-by-hop advertisement is simple, but leads to potential inconsistency. A flooding approach can lead to a consistent view, but requires some traffic control for the route updates.

Routing Update Data Distribution The way the routing data is distributed is partly a consequence of the sources of data for routing calculation. However, in general, routing data is high priority, since without it, user data is unlikely to be delivered to the right place, or via the right path. Routing data may flow over different channels than the users’ data. If so, it is important for a routing protocol to maintain status information about the users’ actual data path, since it is no longer implicit in the reception of routing update data over a link that the users’ data will travel so easily!

If a flooding approach is used to distribute routing data, from or to a management node, or from each router to all others in an area, the flood must be constrained not to loop. This may entail a simplified baseline routing computation (e.g. building a spanning tree amongst all the routers) first.

Routing Calculation Location Having accumulated information about the state of the network, we need to compute the best routes. This might be done at a central node which would then distribute actual path information to each and every router, or, more realistically, can be done autonomously at each router. CPU power is cheap. Memory is reasonably cheap. Thus it makes sense to gain the advantage of timeliness, and independence from failure of other nodes, by distributing (and to some extent, effectively replicating) the computation of the routes at every router.

Routing Update Data Semantics Route updates carry link status information. At its simplest, this is a binary up/down indication. However, increasingly network operators wish to select routes based on more complex *metrics*. These measurements can include: propagation delay; current mean queueing delay; link capacity; monetary cost; loss statistics, and many other possibilities.

In special circumstances, other counter-intuitive approaches have been taken in the design of routing systems. For example, in ad hoc radio networks, random routing is sometimes used. In systems with very limited computational resources (e.g. CPU, memory), Hot Potato routing is used, where packets are forwarded based only on current local information, such as choosing the output link with the shortest current queue.

In practice, at the moment, Internet routing has largely distributed and adaptive. Forwarding decision (frequent) is largely decoupled from routing calculation (occasional). These design choices made make the forwarding task

relatively simple, but the routing protocol and map computation task (RIB and FIB) relatively complex: A few other tasks need to be accomplished:

Neighbour Discovery A router starts out either configured with information about “who is down the end of them there links” or else has to find it out by probing or listening.

status A router maintains some idea of the state of its neighbours, again typically by periodically saying hello - this will sometimes be authenticated. It is part of a specific routing protocol what the exact details are for this.

Distribution Routers collaborate. They exchange views with each other about the state of the net as well as each other. There are actually two fundamentally different approaches to the actual details of this. The common “cheap and cheerful” way is called *distance-vector*. Each router tells its neighbours everything it knows and has learned so far. This is discussed later in this chapter. The more respectable method is called *link-state*. Here, each router *floods* reports about the state of links that it looks after, to all the other routers in a domain, or an area.

Scaling As the Internet grows, the potential load on the links, router CPU and memory systems also grows. Since network growth in the last two decades has been exponential, this would not bode well for network service providers or operators. Luckily, several techniques exist to reduce the load on the network routing and forwarding system.

- Hierarchical addressing

Most networks employ some kind of hierarchical addressing. In many networks, such as the postal system and the telephone network, addresses in use are variable in length. Only as much of an address as necessary need be quoted to make it unique for the area within which it is being used. When constructing routes for the whole world, however, the full address must be used. Thus when sending a letter to someone else in the same city, the author doesn't have to quote the city name or country name, when the post offices of the world construct their routing tables to be able to deliver letters to anywhere in the world, they need to be able to distinguish 10 Downing St, London from 10 Downing St, Cambridge and both of these from 10 Downing St, New York, NY, USA.

In the Internet, addresses are fixed in length, but a technique called Classless Inter-Domain Routing is used to provide the hierarchical addresses within a fixed length field. This relies on associating a variable length mask with an address in use in a routing area, and exchanging the mask as well as the address when building up maps of the world. We look at this more below.

- Hierarchical routing

In large networks, it is generally the case that different regions in the network have more or less connectivity. If a region is connected to the rest of the world only via a single link (“single homed”), then it seems unnecessary to bother the rest of the world with the details of the internal topology of the region.

It is also the case that we often deploy new routing protocols in part of the network at a time rather than having a “flag day”, when we upgrade (or downgrade) the entire net.

Thus it is important to separate routing into regions. This is done in two different ways in Internet Routing. Firstly, routing protocols are divided into *intra-domain* and *inter-domain*. Here, the term “domain” refers to the autonomous management of routing, and not any other aspect of networking. Intra-domain routing domains are separated from each other by inter-domain routing protocols. In IP networks, these are known as IGP and EGP. Secondly, within a domain, a routing protocol may have explicit configuration management mechanisms for dividing routers and links into sub-domains. In the internet, there are several examples of this, ranging from subnet routing, up to area routing (in OSPF and ISIS). We will revisit this briefly at the end of this chapter.

Routing Table Construction It is an internal matter for a routing daemon how it constructs its routing information base. The entries will essentially list all the peer routers, and typically all possible destination prefixes, as well as some destinations known about but currently unreachable. The most general data structure for this would be a graph of nodes (routers) and edges (links), with the edges labelled with the various metrics for making route (or forwarding) decisions. Associated with any node and edge would be the source of the information about it, and the time that the last information was received. Of course, in practice, given all information arrives either from, or via neighbours, this data could be built around a smaller data structure, such as a list of neighbours, and the list of updates, with a single entry for the last update time - it really depends on the granularity of information in the routing protocol.

Forwarding Table Computation At the end of each update epoch, a computation is carried out by the routing daemon, to derive the new forwarding information base (“FIB”). This is then downloaded into the forwarding engine - in linux terms, the kernel fib is updated by the routing process. This can and often is manual, of course.

In distance-vector algorithms, the data distribution *is* the computation. A router receives a vector of destinations and metrics from a neighbour, and simply replaces the forwarding entry for the next hop with the new value, if the new value is better.

In link state protocols, a more complex procedure is needed to derive the forwarding table from the set of route updates. At the end of the route update flooding epoch, the network graph is built, simply by adding all the updates together in a “jig-saw puzzle”. Subsequently, Dijkstra’s well known graph walking algorithm can be used to carry out a forward search from the current router to all other routers (and the networks attached to those routers). This finds the path from “here to there” with the lowest metric. If paths with different metrics need to be considered (or multiple alternate paths for load balancing or instant failover), then the algorithm can be called multiple times.

Typically, we can imagine that this only occurs for the entire routing system very occasionally. Most updates may result in local parts of the graph being changes, and there are several techniques for incrementally computing a “Dijkstra”, and keeping the CPU cost to a minimum.

Forwarding is simple. However, the lookup of an entry is complicated by the fact that the procedure typically involves an algorithm called “longest match” due to Classless Inter-Domain Routing. There are a variety of data structures that have been used to keep this procedure fast and also to keep the associated forwarding tables small, and we describe the one currently used in Linux in this chapter.

The rest of this chapter discusses simple forwarding and routing in Linux. More complex routing protocols are not part of any standard release, but are rather separate products that are available from the networking community (e.g. gated). In the future, we hope this may change, as open source produces some high quality routing protocol daemons.

A complete discussion of routing protocol design is outside the scope of this book!

In the next section, we take a look at the Linux kernel support for forwarding information.

8.3 Forwarding Table Computation

8.3.1 IP Addressing

Each packet arrives with some information in it about where it wants to go and how it wants to be treated. The simplest data we can lookup in our forwarding table in an IPv4 data packet is the destination address.

IPv4 addresses are 32 bits long. This makes a simple table lookup somewhat unrealistic. Even if each entry in the table just had the 32 bit IP address of the next hop (ignoring other important information!), we would need $4 * 2^{32}$ bytes of memory in every router (16 Gigabytes of memory!). In other network protocol worlds, such as connection oriented packet networks, this approach is taken, since the number of entries used for forwarding each packet is limited to the maximum number of concurrent connections, and this is typically a smaller number. In fact, we will see later how MPLS can make use of this idea. In fact, we will see later how MPLS can make use of this idea. It has the added bonus of being efficient (a single access cycle time to return the data). This means it has potential for low cost hardware implementation. If we know that we are building hardware with a limited number of flows and a limited number of interfaces that the next hop can be reached on, we can limit the table size to something quite tractable. However, this would not be in line with the Internet philosophy of generality, flexibility, or software implementations, which is what we are mainly concerned with here.

Staying with the idea of a “flat” (i.e. unstructured) 32-bit address, a programmer’s first thought when confronted with this type of problem is to look at a *hash table*.

Hash tables rely on some knowledge that the distribution of *used* entries is relatively sparse, and relatively evenly scattered in some sense. With some knowledge of the distribution of actually assigned IP addresses, a *hash function*

can be designed which *maps* the address, from a possible 32-bit value or 2^{32} entries, into a smaller identifier space. For example, if we know that there are only 65536 assigned net numbers in the world, we could have a table with a mere 100,000 entries (a few spare to be on the safe side!), and choose the hash function to be `modulo`, in this case, `modulo 216`.

IP addresses are not assigned in a perfect uniform random distribution, so this will lead almost inevitably to *collisions* or clashes in entries in the hash table. These are dealt with by *chaining* a linked list from the entries that collide. This also needs a marker to say if the entry *is* an entry, or is part of a chain.

[Figure 695 about here.]

The hash function takes somewhat longer to compute than a table offset. It then subsequently requires a table offset in any case, and typically, this results in a possible linked list search too. This is considerably more costly in CPU than in memory. However, the relative costs of CPU and memory are in favour of this kind of approach as a basis for modern route lookups for IP like networks. Considering trends in current memory performance and prices, it would be a decade or more away before we could consider direct route lookup. If you consider IPv6 addresses, which are 128 bits long, any likely full table is completely out of the question.

Of course, life is more complex than this: IP addresses are effectively structured into several parts. IPv4 unicast addresses are in two parts. IPv6 addresses may have more components depending on the scheme in use. IP multicast addresses are yet another kettle of fish, and we look at IP multicast later in this chapter,

For now, let's just consider IPv4. An IPv4 has a *network* part and a *host* part, together making up the full 32 bits. Historically, a site was allocated IPv4 network address taken from one of three *classes* of the address space. The idea was that there were typically three different size networks: the core networks such as the ARPANET and SATNET - these would be given 8-bit network addresses (class A), and there could be only ever up to 256 of them; the site networks such as Berkeley, MIT, UCLA and UCL - These were given 16 bit network numbers, (class B) and there could be 65536 of them, each with 65536 hosts on!; finally, there would be small local networks, with only 256 hosts on them - but there could be up to 2^{24} of these, that is 16,777,216 Class C network numbers.

There are two big problems with this approach: firstly, not only does "one size fits all" not apply, but in fact three sizes didn't fit all either; many networks have a topology that allows more than 256 hosts to be attached locally, but if given a Class C network, not all these hosts would be reachable; conversely, many networks only attached 100 hosts to a class B network! More critically, the routing system had to know about all the addresses, yet in practice (and this is not unrelated to the first point), many sites had a single way to reach the whole world, even when they had several networks "internally".

The first step away from a globally visible network numbering system was the introduction of *subnet addressing*. The second step away from a globally visible topology was the hierarchical allocation of addresses to organisations. Consideration of the size and topological location of an organisation would lead to the assignment of a network number, together with a mask, which indicates

how many bits of the address are actually free to be used for allocation to hosts, and how many are for advertising the network as reachable to the next “level” of the topology.

The downside of this is that it leads to the requirement for hosts, and routers, and routing protocols to know or discover the mask associated with their region of the network. The upside is that it leads to the possibility that address prefixes may now be hierarchically aggregated. In other words, in the “edge” of the network, we only care about routes towards the “core”, while in the core of the network, we only need now about the highest levels of aggregation. Hopefully, these would usually be long masks, or shorter network prefixes, leading to a reduction in routing table requirements.

[Figure 696 about here.]

The scheme has worked very well for local and global routing table management for several years. Recent trends in multi-homing have emerged from the requirement for sites to have high availability. If multi-homing for a site is done “transparently” this requires network providers to advertise the site prefix through at least one path along which the address is not aggregatable.

An important consideration for implementors of classless inter-domain routing is that a router must now compare destination addresses in packets with multiple possible routing entries. Which routing entry is the correct one to pick? Clearly, the *most specific* entry is the one “nearest” the destination. Given the way that addresses are now hierarchically allocated and assigned, this means that the best match is the *longest* match, or in other words the one with the longest number of bits matching in the prefix in the destination address and the routing entry address after the routing entry mask has been applied.

A naive approach to implementation might be to carry out a linear search of the routing table. This may be all right in local routers, but in the routers nearer the core of the Internet, we find tens of thousands of route entries. If a single hash table is used, we do not know the order with which entries with different length prefixes will occur. Other solutions are required, otherwise we end up with a linear search, which results in the same order of number of lookups, *for every single packet to be forwarded* as there are route entries, which can be very large.

Trees and Tries The old BSD Unix approach to the forwarding table was to keep three levels of information: host, route, and default. The search involved hashing into the host table, and if a miss occurred, hashing into the route table, and then finally, if that failed, searching the list of default gateways. Later, in 4.3BSD, radix trees were developed[?]. These are a form of Patricia Trie - a trie is just a name for the types of tree data structure which stored information at each internal node as well as at the leaves. The 4.3BSD Radix tree is a binary tree, which, starting at the root, stores the prefix address and mask “so far”. As you move down the tree, more bits are matched going one way down the tree. If they don’t match, the other branch holds the entry required. This sort of data structure can result in efficient average performance for forwarding table lookup times, on the order of $\ln(\text{number of entries})$, which for large routing tables is quite a gain (e.g. for 65536 route entries, average of

around 16 comparisons). Of course, the worst case performance is the length of the address, or 32 comparisons for an IPv4 address.

Caches The next stage in the evolution of forwarding table lookup schemes was the idea of adding a cache of most recently used routes in front of the standard FIB. Before looking in the standard FIB, the forwarding code will look up a destination in a route cache, which is hopefully a smaller table, and can be searched more quickly. This idea sounds good, until you realise that the performance advantage of caching depends on the pattern of access over time. Specifically, caching relies on temporal locality - this means that if a sequence of lookups resulting do in fact tend to reference the same subset of destinations, and this subset only changes much slower than the rate of actual packets arriving, then a cache can be a big win. In practice, it depends where you are in the net whether this is true or not - certainly, according to measurements made by some experts for core routers, there doesn't appear to be a great deal of temporal locality. The Internet is "all over the place" [14]. On the other hand, a given site router may see packets go by with a very focussed set of destinations.

Multiple Hash Tables It was observed that one could have a hash table for each possible prefix length. A refinement of this is to see that, rather than searching all of these different tables linearly, one can do a binary search by prefix lengths, starting with the most common [12]. This is straightforward computing. Even with an IPv6 address of 128 bits, this will take 8 hash lookups to find the longest matching entry.

Trie Compression A more recent approach still is to try to improve over the radix tree by compressing the trie in some way [13]. This scheme involves reducing the number of levels in the trie by increasing the degree of the nodes.

Content Addressable Memory - CAMs A CAM is a custom piece of hardware that is addressed by its content - an 32-bit wide CAM would be a device which is accessed by presenting it with an input value (e.g. the IP destination) and returns another value (i.e. the next hop) if the input value is in its contents. Generally CAMs get expensive if the width is large, and require an exact match. Due to the interest in fast IP packet forwarding (and therefore fast lookup) there are a number of refinements to these devices, such as *ternary CAMs*. Ternary CAMs allow 3 values for each input bit, including a "don't care". This allows multiple matches to be done in parallel - the order of the results typically implies preference.

8.3.2 FIB Purpose

Forwarding tables are consulted for every packet to be sent from a system. An entry is indexed by IP destination address, and has to store this and the associated mask, as well as information that may be used to time out an entry, and metrics. The returned value from looking up an entry will typically be the next hop IP router. Local routing (i.e. link level mechanisms) are then used to deliver a packet to the next IP hop.

The forwarding table data structures in Linux are pretty sophisticated, as well as being close to state-of-the art in efficiency.

8.3.3 FIB Implementation

The diagram below illustrates how the FIB data is stored in the Linux 2.4 kernel.

[Figure 697 about here.]

8.3.4 FIB Interface

In the `rtattr` data structure⁶⁹⁵, we can see the generality of information that could be added in the FIB. In principle, we have source and destination based routing, input and output interfaces, priority, multipath, protocol based and so on.

The fields are typically added into a FIB entry by a call via the `rtnetlink` socket, from a routing daemon, or from the superuser running the `IP` command, which we look at later.

[Figure 698 about here.]

As described before, the final result after the long smart haul of looking up a forwarding information base entry is the *next hop*. This data structure⁶⁹⁶ is relatively simple either we are told the IP address of the next hop gateway, or output device (or both) - the output device is all that is needed if we have a point-to-point link, for example, where the actual IP address of the next hop is irrelevant. Linux also supports multipath routing, where we can load balance traffic across several links. It also indicates the distance (“scope”) to the destination.

[Figure 699 about here.]

FIB information is not just a single “next hop”. There is other data, as shown in 697. This includes the possibility of multipath, source based preferences and some entry maintenance data that we’ll look at later.

[Figure 700 about here.]

As described before, the lookup in the FIB essentially returns one or more next hops - the case for multiple next hops is generally currently only when we have multi-path routing operating, when traffic is being balanced over more than one link. We can see the different metrics, and other parameters that need to be matched for some types of lookup, such as protocol, preference and priority, and then there are a number of internal management fields such as the tree reference, client reference, and liveness (`fib_dead`) and flags fields.

[Figure 701 about here.]

The overall structure ⁶⁹⁷ is complex. Usually, FIB related functions return the `fib_result` type⁶⁹⁸ as a reference parameter, with the `fib_info *fi` field pointing at the appropriate entry. return

The `fib_table` structure is a more generic interface to the fib, including pointers to the various FIB access functions, so that new FIB functions can be written per routing system. Currently, only the set of lookup functions in the `fib_hash` module are defined so the pointers are only initialised to point at the respective `fn_` functions.

[Figure 702 about here.]

8.3.5 FIB Code

The FIB implementation is split into four main parts. There is a front end (`fib_frontend`), which is the externally visible interface to the FIB. Then there is the `fib_rules` module, which implements the set of ways that we pick a next hop - we need to discriminate one from another potentially by policy or metric, as well as simply on longest match. Thirdly we have the `fib_semantics` module, which deals with the arcane definitions of what `nexthop` really means - this is well described in the comment 728. Finally, there is the `fib_hash` module, which implements the set of hash functions described earlier in this chapter.

The Frontend

[Figure 703 about here.]

[Figure 704 about here.]

[Figure 705 about here.]

[Figure 706 about here.]

[Figure 707 about here.]

[Figure 708 about here.]

[Figure 709 about here.]

[Figure 710 about here.]

[Figure 711 about here.]

Accessing FIB Statistics

[Figure 712 about here.]

[Figure 713 about here.]

[Figure 714 about here.]

[Figure 715 about here.]

[Figure 716 about here.]

[Figure 717 about here.]

[Figure 718 about here.]

FIB Event Handler

[Figure 719 about here.]

[Figure 720 about here.]

[Figure 721 about here.]

[Figure 722 about here.]

Semantics - what it all means

[Figure 723 about here.]

[Figure 724 about here.]

[Figure 725 about here.]

[Figure 726 about here.]

[Figure 727 about here.]

[Figure 728 about here.]

What is a “next hop”?

[Figure 729 about here.]

[Figure 730 about here.]

The FIB Big Picture

The comment below goes a long way to explaining how this works.

[Figure 731 about here.]

Checking for next hop

[Figure 732 about here.]

[Figure 733 about here.]

Creating a new a FIB entry

[Figure 734 about here.]

[Figure 735 about here.]

[Figure 736 about here.]

[Figure 737 about here.]

[Figure 738 about here.]

Looking up FIB entry

[Figure 739 about here.]

[Figure 740 about here.]

Accessing more FIB statistics

[Figure 741 about here.]

[Figure 742 about here.]

[Figure 743 about here.]

Taking an RT netling request

[Figure 744 about here.]

[Figure 745 about here.]

[Figure 746 about here.]

Interface state changes, update relevant FIB

[Figure 747 about here.]

[Figure 748 about here.]

[Figure 749 about here.]

Multipath Routing

[Figure 750 about here.]

[Figure 751 about here.]

FIB Proc FS

As well as RTNetlink, we also have the proc fs interface to modules.

[Figure 752 about here.]

Rules is rules

Now we can look at the rules for forwarding:

[Figure 753 about here.]

[Figure 754 about here.]

[Figure 755 about here.]

[Figure 756 about here.]

[Figure 757 about here.]

[Figure 758 about here.]

[Figure 759 about here.]

[Figure 760 about here.]

[Figure 761 about here.]

[Figure 762 about here.]

[Figure 763 about here.]

[Figure 764 about here.]

[Figure 765 about here.]

Lookup - Making it all go fast!

[Figure 766 about here.]

[Figure 767 about here.]

[Figure 768 about here.]

[Figure 769 about here.]

[Figure 770 about here.]

Zone Information

Next we need to look for the right zone.

[Figure 771 about here.]

Then we look for the actual entry.

[Figure 772 about here.]

Or find a dead one.

[Figure 773 about here.]

or a default:

[Figure 774 about here.]

[Figure 775 about here.]

Adding a FIB Entry

Now we look at the case of adding (updating) entries.

[Figure 776 about here.]

[Figure 777 about here.]

[Figure 778 about here.]

[Figure 779 about here.]

[Figure 780 about here.]

[Figure 781 about here.]

Adding a FIB Entry

[Figure 782 about here.]

[Figure 783 about here.]

[Figure 784 about here.]

[Figure 785 about here.]

Management

[Figure 786 about here.]

[Figure 787 about here.]

[Figure 788 about here.]

[Figure 789 about here.]

[Figure 790 about here.]

Starting it all up

[Figure 791 about here.]

8.4 Multicast

The IP Protocol supports one-to-all, one-to-many and many-to-many delivery of packets. Linux supports these facilities through normal host support for broadcast and multicast addressing and through multicast forwarding support in the kernel. Again, as with unicast routing, if you want a routing daemon, you have to currently go elsewhere (Gated has support for most of the common multicast routing protocols).

The Internet Standard Multicast service provides what some people refer to as “any-to-many” delivery of IP packets. In the same way that anyone can send a normal IP packet to any IP unicast destination address, anyone can send an IP multicast addressed packet to any group

A group is defined by an IP multicast address and the current set of receivers who have expressed an interest in the group. As we discussed in chapter six, membership is achieved via the Internet Group Management Protocol.

Many applications exist for which there is a single sender and many recipients. So long as the number of recipients is less than the number of hosts at a site, or on a network, then multicast is useful. Some applications exist where there are also several senders (usually, in fact almost always, these senders are also receivers).

A group is defined in terms of a set of hosts. The effect of multicast routing is that a single transmission, results in reception at many receivers Optimise transmissions: e.g. reduce duplication

Multicast addresses, at least in IPv4, are Class D and are in the dotted decimal notation, in the range 224.0.0.0 - 239.255.255.255, as discussed in chapter one and five. Instead of referring to a host interface, an IP multicast address refers to a group of interfaces. IGMP and the socket API allows applications to indicate their membership of groups. Some addresses are reserved for internal management of multicast (e.g. for IGMP messages and for multicast routing messages).

There’s a wide variety of applications for multicast. Many are single-source, but there are a few important ones that have multiple senders. For example, voice and multimedia conferencing is quite often touted as a good example of a many-to-many application, although some higher level systems implement it as a set of point-to-point flows, between clients and a distribution server which uses a one-to-many IP multicast channel back to all the clients.

Classic single source applications include software update and distribution, and news distribution.

Other multiple sender applications may in the future include multi-player games and distributed simulations. Currently, though, most of these are server based.

IP multicast support can be separated into three components: local network support; wide area internet router support, and general traffic management, including scoped transmission using the IP TTL header field and other facilities.

Features of IP multicast are that a group of hosts is identified by a Class D

address. Applications use a transport protocol (*not* TCP, but something usually based on UDP, or else some novel transport protocol such as PGM) to carry out any higher level end-to-end functions required over IP multicast (reliability, synchronization, session management etc etc).

Multicast systems can be divided into leaf nodes (hosts) and intermediate nodes (routers). The routing system conspires to construct delivery *trees*, rather than simple paths. It is critical to understand that IP multicast supports dynamic membership with a leaf-initiated join paradigm. This means that there is a lot of potential control traffic as hosts join and leave a multicast group. Another (typical in line with the Internet design philosophy) feature is that a non-group member can send to group (subject to any firewalls etc).

Most IP routers in the world are multicast capable. However, not all of the Internet has multicast enabled - particularly true in dial-up and other low bandwidth access links.

[Figure 792 about here.]

As described in chapter six, IGMP is used to coordinate group membership between hosts and *first hop* routers on the local net. On a given local link, just as with IP unicast, there is a need to translate or map IP addresses to MAC address. For a point-to-point link this is trivial, but for an IEEE 802 type net, there's a bit more to it: Unlike unicast addresses where ARP is typically used, IP multicast employs algorithmic resolution from layer 3 to layer 2 addresses since it is quick, easy, and distributed, and doesn't lead to any of the broadcast storms associated with misconfigured ARP services.

The MAC address format is illustrated in the diagram 16. The IANA has carried out a MAC address allocation, and the algorithm is to take the last 23-bits of Class D address, and superimpose these on the MAC prefix assigned. Note that unfortunately, this is not a 1-1 mapping, which means that potentially, several IP multicast addresses may share a given MAC address - the mapping is designed to make this unlikely, but in this event, host filtering required at IP layer to recognize addresses for which the host *does* have receivers. In fact, due to hardware limitations in many LAN cards, it is quite likely that a host that is a member of more than a few groups will have to do this filtering in any case - this is because the only way to receive multicast to more groups than the LAN card can recognize is to enable promiscuous reception of *all* multicast (or in the worst case, all frames!). In this case, the IP level maintains a list of active groups and rejects silently all other packets addressed to other groups. Of course, this represents a load on the host processor!

[Table 16 about here.]

[Figure 793 about here.]

8.4.1 Dense Mode, Data-driven Multicast Routing

The routers conspire to deliver packets to sites where there are members. In the simplest routing approach, not something used nowadays, but illustrative of a neat idea, the path from sender to receivers is built in a *data driven* manner. As soon as someone sends an IP multicast destined packet, the routers deliver it out all interfaces except the shortest path back towards the sender (i.e. the

interface it came in; although an RPF check can be performed to eliminate loops for packets that traverse a net while the routing is settling down after a change). In this approach, the packets would flood everywhere in the Internet, which is pretty big nowadays. If there were more than a very modest number of multicast data sources, this would be a very bad idea indeed. Thus the multicast scheme described here then employs another trick: Networks with no group members are *pruned* from tree. Routers with no site members send routing messages *upstream* towards the source to indicate that they have no need of traffic for this group. Of course, with dynamic membership, we might suddenly find someone at such a site gains a member for a group where previously they had pruned themselves from the source tree. We must somehow allow tree to re-grow, and this is done by using a new message between dense mode routers called a *prune* message. In some senses this just extends the idea of IGMP join (as a graft message extends the idea of IGMP leave) towards a source.

To keep life safe from lost messages, the *prune state* which suppresses data distribution is periodically subject to timeout and removal, which leads to re-flooding of the data. Downstream nodes without receivers prune again.

Of course, this scheme doesn't scale at all. However, it illustrates the basic idea of distribution trees and state management. It also resulted in a modest, but important change to routing, which is that the forwarding engine needs to update the routing information base as a result of routes being installed (at least for multicast) data driven, as well as because of router-router updates, or applications signalling.

8.4.2 Sparse and Single Source Multicast

The now widely used alternatives to dense mode multicast are sparse mode and single source. These both leverage off the idea of an explicit graft message flowing from downstream nodes to join tree up towards a distribution point, known as the rendezvous point ("RP"). In the latter case, the rendezvous point is at the source (i.e. single only) source itself. In both cases the data driven nature of flooding and pruning is avoided entirely, but the distribution trees may be sub-optimal - switching from RP to source based tree is a configuration option once the rendezvous with a receiver has been made.

8.5 QoS Routing

Traditional routing involves selection by destination address only. This means that it is hard to load balance or provide sensible choices of path or route to accord with non default performance requirements. Traditional routers have one optimal path to destination, and the routing metrics are single values. For QoS routing, multiple paths would be possible, and alternative paths would have different QoS properties. Routing updates need to include QoS parameter information, and the route selection must use destination address, source address, ToS, etc. The whole integrated services system aimed to a lesser extent differentiated services require some way to inform routers about these extra requirements.

Note that data driven route update leads to a kernel-to-user signal or upcall requirement.

8.6 Routing Table Construction

Currently, there is no standard open source user space routing protocol for Linux. There are a number of projects that have produced routing protocol daemons that are widely used. The most prominent full specification system is the Gated package from Merit.

8.6.1 GateD

GateD is a large system which implements a significant fraction of the Internet routing protocol suite. Currently, it consists of three main pieces, comprising a core set of common code, a routing database system, and a collection of protocol modules which implement the following standards:

- **RIPv1 and RIPv2**

RIP is the Routing Information Protocol. It is a distance vector protocol. RIPv2 incorporates masks in the routing updates, and so can support classless inter-domain routing properly. It is useful mainly in small site networks.
- **OSPFv1 and OSPFv2**

OSPF is the Open Shortest Path First routing protocol. This is full industrial strength hierarchical link-state routing system, and scales quite well to large corporate intranets, or modest sized ISPs.
- **IS-IS**

IS-IS is the Intermediate System-Intermedia System protoocl, derived from DECNet routing, and then, via the International Organisation for Standardisation (ISO), introduced into the IETF as a multi-protocol link-state hierarchical routing protocol. It has very similar properties to OSPF, although it is defined (the packet headers) are a little more general. Some operators claim it scales a little better. Others disagree. It is probably a matter of implementation detail (the devil is always in the details!).
- **EGPv2**

EGP is the Exterior gateway Protocol, the original inter-domain reachability mechanism in the 1980s.
- **BGP2, BGP3, BGP4 and BGP+**

BGP4 is the state-of-the-art Border Gateway Protocol, which supports a path-vector routing algorithm, and allows ISPs to claim which destinations they advertise as reachable across their border (whether for ingress, egress or transit. It provided unilateral, and bilateral policy control but not multi-lateral control. A path vector algorithm is like a distance vector algorithm with the entire path spelled out in the update rather than merely listign the destination and a single metric. The BGP3 to BGP4 evolution included the update to provide CIDR.
- **DVMRP**

DVMRP is the Distance Vector Multicast Routing Protocol, now largely a historic milestone in the evolution of multicast. It is a dense mode (flood and prune) mechanism, and so is rather a dangerous beast to let loose outside of a small site network. In fact, its probably fairly dangerous anywhere except on a broadcast medium subnet.

- MSDP, PIM

PIM is the Protocol Independent Multicast routing protocol. It provides Dense Mode, Sparse Mode, and Single Source multicast routing. PIM DM (Dense Mode) is like DVMRP but without its own unicast routing computation to derive the reverse paths from. Instead, it depends on the existence of unicast routing in some other form, and uses some API to extract the reverse paths from the RIB or FIB. PIM SM (sparse mode) is similar, but oriented around a Rendez-vous point, as described earlier in this chapter.

MSDP is the multicast source distribution protocol, which is the nearest thing we have to a multicast inter-domain routing protocol right now. It distributed between designated routers in a domain, the existence (liveness) of a source in one domain, for receivers in another. It does not have good scaling properties.

- RIPv6

RIPv6 is RIP re-engineered to support the larger address space and address allocation schemes in IPv6.

- ICMP and IGMP

GateD supports ICMP and IGMP, but then if it runs on a Linux box, these are native to the kernel in anycase.

GateD (“GATEway Daemon”) was implemented at a time when the US federal agency the National Science Foundation, funded and operated a US continent -wide research and education network as part of the Internet. At that time, state-of-the art routers did not really support the linespeeds or full range of protocols needed for such a complex system, so it was necessary for the agency to commission it own. GateD is *not* part of the open source world, and as such, we will not discuss it in great detail here. However, it is fairly comprehensive, and has been used as the basis for further development by a number of folks, so is worth of attention. It runs on most common platforms including Linux.

A similar package called Zebra is also available - this supports BGP-4 protocol as described in RFC1771 as well as RIPv1, RIPv2 and OSPFv2. While previously under the GPL, at the time of writing, the status of this system was unclear.

There are a number of good books on IP routing which we refer the reader to for details, e.g [15].

8.6.2 Route Control and Traffic Control Maintenance

A routing protocol on Linux runs in user space. In some senses it is an application, which makes use of a transport protocol (UDP for some routing protocols,

TCP for others, e.g. BGP) to exchange messages with peers.¹

However, the fortunate side-effect of a routing protocol is to compute a forwarding information base (FIB) from the Routing Information Base (RIB) that it gleans from its exchanges with its peers.

As far as the interface between the routing daemon and other routing remote peer daemons, normal socket programming is just fine. Whether the daemon is multi-threaded or not is not of interest here.

Management tools such as routing daemons need a control channel to access the forwarding information base. This channel is also needed for traffic control tools to be able to configure packet classifiers for customising forwarding treatments as discussed in the next chapter, as well as for setting up filters for security and access control, as discussed in chapter ten.

Lets review communication between user space and kernel. In chapter one we had a look at how system calls allow user processes to use kernel services. However, the kernel is really a conduit through which the user process accesses an external resource (file systems, communications with other computers, input/output to the user etc etc).

What is different here is that the Linux does not use a simple map of the kernel memory the way that other Unix systems have done. While it does provide /proc file system, which is a lot more convenient than earlier schemes such as reading /dev/kmem, there is a requirement for a *protocol* to communicate in both directions between the kernel and user space. The scheme is part of a history starting with a mechanism called `cmsg` which allows applications to look at ancillary data associated with a packet or a socket, but which are not normal parts of the socket payload. For example, the normal socket interface (as described in chapters four and nine) does not allow access to the TTL field of the IP header of an inbound, yet for some applications this is important or at least useful. Subsequent to this, a more general mechanism called `netlink` was developed, which allows a new type of socket `PF_NETLINK` to be created for communication between application space (e.g. a routing daemon such as *gated*) and kernel space. This provides an unreliable message protocol, on top of which a particular application must build its own level of reliability (and flow and resource control). It also has the beginnings of support for callbacks. Finally, the current interface is `rtnetlink`.

rtnetlink API

All `rtnetlink` messages consist of a netlink message header and appended attributes. There are a set of convenient macros for accessing the attributes.

```
rtnetlink_socket =
    socket(PF_NETLINK, int socket_type, NETLINK_ROUTE);

int RTA_OK(struct rtattr *rta, int rtabuflen);
void *RTA_DATA(struct rtattr *rta);
unsigned int RTA_PAYLOAD(struct rtattr *rta);
struct rtattr *RTA_NEXT(struct rtattr *rta, unsigned int
```

¹In fact, routing protocols are the best example of a peer-to-peer application. Almost all other applications until the recent emergence of gnutella, have been client-server based. Routing is the most inherently, fundamentally distributed application.

```

rtabufLen);
unsigned int RTA_LENGTH(unsigned int length);
unsigned int RTA_SPACE(unsigned int length);

```

Here is an example of the use of Creating a rnetlink message to set a MTU of a device.

```

struct {
struct nlmsg_hdr nh;
struct ifinfomsg if;
char          attrbuf[512];
} req;

struct rtattr *rta;
unsigned int mtu = 1000;
int rtnetlink_sk = socket(PF_NETLINK, SOCK_DGRAM, NETLINK_ROUTE);
memset(&req, 0, sizeof(req));
req.nh.nlmsg_len = NLMSG_LENGTH(sizeof(struct ifinfomsg));
req.nh.nlmsg_flags = NLM_F_REQUEST;
req.nh.nlmsg_type = RTML_NEWLINK;
req.if.if_i_family = AF_UNSPEC;
req.if.if_i_index = INTERFACE_INDEX;
req.if.if_i_change = 0xffffffff; /* ???*/
rta = (struct rtattr *)(((char *) &req) +
NLMSG_ALIGN(n->nlmsg_len));
rta->rta_type = IFLA_MTU;
rta->rta_len = sizeof(unsigned int);
req.n.nlmsg_len = NLMSG_ALIGN(req.n.nlmsg_len) +
RTA_LENGTH(sizeof(mtu));
memcpy(RTA_DATA(rta), &mtu, sizeof(mtu));
send(rtnetlink_sk, &req, req.n.nlmsg_len);

```

Some rnetlink messages have optional attributes after the initial header:

```

struct rtattr
{
    unsigned short rta_len;      /* Length of option */
    unsigned short rta_type;    /* Type of option */
    /* Data follows */
};

```

These attributes should be only manipulated using the RTA_* macros or libnetlink. Here are all the message types:

```

RTM_NEWLINK, RTM_DELLINK, RTM_GETLINK
Create, remove or get information about a specific
network interface.

```

RTM_NEWADDR, RTM_DELADDR, RTM_GETADDR

Add, remove or receive information about an IP address associated with an interface.

RTM_NEWROUTE, RTM_DELROUTE, RTM_GETROUTE

Create, remove or receive information about a network route.

RTM_NEWNEIGH, RTM_DELNEIGH, RTM_GETNEIGH

Add, remove or receive information about a neighbour table entry

RTM_NEWRULE, RTM_DELRULE, RTM_GETRULE

Add, delete or retrieve a routing rule.

RTM_NEWQDISC, RTM_DELQDISC, RTM_GETQDISC

Add, remove or get a queueing discipline.

RTM_NEWTCLASS, RTM_DELTCLASS, RTM_GETTCLASS

Add, remove or get a traffic class.

RTM_NEWTFILTER, RTM_DELTFILTER, RTM_GETTFILTER

Add, remove or receive information about a traffic filter.

The Route Interface code that handles this

[Figure 794 about here.]

[Figure 795 about here.]

[Figure 796 about here.]

[Figure 797 about here.]

[Figure 798 about here.]

[Figure 799 about here.]

8.7 Host routing

A Linux end system host needs to choose a route over which to send packets, but we do not really wish to incur the whole lookup overhead that is associated with the complex control for a full blown router. Thus there is another level API to the routing system which keeps a cache of recently used routes - this would not be efficient for a forwarding engine unless its workload showed remarkable locality of reference, but is likely to work well for client and server machines where the length and number and locality of flows means that many packets will hit the cache. The cache is maintained in the kernel and linked to from the socket structure (as described in chapter five). It is looked up by a simple hash key, which is matched in the table itself:

[Figure 800 about here.]

The table is as follows:

[Figure 801 about here.]

[Figure 802 about here.]

[Figure 803 about here.]

[Figure 804 about here.]

[Figure 805 about here.]

[Figure 806 about here.]

[Figure 807 about here.]

[Figure 808 about here.]

[Figure 809 about here.]

[Figure 810 about here.]

[Figure 811 about here.]

8.8 Managing IP level state information - the IP command

From user space, the `ip` command acts to glue all the `rtnetlink` API calls together into a single application for manual configuration, or access to statistics. For those familiar with BSD like Unix, this command inprobrates the combined functionality of the `ifconfig`, `route`, `netstat`, `arp` and other applications. Many of the things managed by this command have been covered in chapter six.

The command is well documented in the system releases, so we just summarise its capabilities here.

```
ip [ Options ] Object [ Command ] [ Arguments ]]
```

Options `-V`, `-s`, `-f -r`, for version, statistics, family (`inet`, `inet6` or `link`), and use DNS instead of IP addresses.

Object this is what we are trying to manage - each corresponds roughly to a separate command on other systems, with the exception of firewalls and traffic control, which are managed through separate commands which we discuss in chapter nine and chapter ten respectively. This can be *link*, *address*, *neighbour*, *route*, *multicast address*, *multicast route* or *tunnel*.

Command takes some action on the object named, typically, `add`, `delete`, `set`, `show` or other object specific command.

Now we look at these in more detail:

link - basically manage link device level information.

```
e.g.    ip link set { dev <name> | [up|down] | arp [on|off] |
multicast [on|off] | dynamic [on|off] | txqlen <n> | mtu <n>
| address <link address> | broadcast link broadcast address>
} or ip link show
```

The latter command can produce a lot of output, for example:

```
1: lo: <LOOPBACK,UP> mtu 3856 qdisc noqueue
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
   link/ether 00:c0:4f:d3:db:c3 brd ff:ff:ff:ff:ff:ff
3: tap0: <BROADCAST,MULTICAST,NOARP> mtu 1500 qdisc noqueue
   link/ether fe:fd:00:00:00:00 brd ff:ff:ff:ff:ff:ff
4: teql0: <NOARP> mtu 1500 qdisc noop qlen 100
   link/void
5: dummy0: <BROADCAST,NOARP> mtu 1500 qdisc noop
   link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
6: bond0: <BROADCAST,MULTICAST,MASTER> mtu 1500 qdisc noqueue
   link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
7: eql: <MASTER> mtu 576 qdisc noop qlen 5
   link/slip
8: tunl0@NONE: <NOARP> mtu 1480 qdisc noop
   link/ipip 0.0.0.0 brd 0.0.0.0
```

```

9: gre0@NONE: <NOARP> mtu 1476 qdisc noop
   link/gre 0.0.0.0 brd 0.0.0.0
10: sit0@NONE: <NOARP> mtu 1480 qdisc noop
   link/sit 0.0.0.0 brd 0.0.0.0

```

The output shows the interface name and index, the device link operation mode (e.g. LOOPBACK, BROADCAST, MULTICAST) and status. the maximum transfer unit size in operation, the queueing discipline in use (if any, with queue length configured), the MAC address type and value and broadcast value, and so on.

Another example of this is if we run `ip-s link show eth0` on the machine above:

```

2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
   link/ether 00:c0:4f:d3:db:c3 brd ff:ff:ff:ff:ff:ff
   RX: bytes  packets  errors  dropped  overrun  mcast
        2805243643 31255469 28      0        28      0
   TX: bytes  packets  errors  dropped  carrier  collsns
        1423998005 15674317 0       0        1       815994

```

Here we can see the packet level statistics seen by the device. This shows link level specific properties too such as collisions, overruns and carrier lost counts, for ethernet, and for a SLIP interface this would also show compressed packets.

address - configure or list address(s) of interface/device.

```

This can be used as ip address add { [dev <name>| local
<address>] | peer <address> [default] | broadcast <address>|
label <name> | scope <scope value>] }

```

Here the only new value is `scope`, which can be `global`, `site`, `link` or `host`.

neighbour - setup or examine an ARP entry or neighbour discovery cache information

```

For example, ip neighbour add {to <address> | dev <name> | nud
<state> } Here the only non-obvious parameter is nud which refers to
an "neighbour unreachahle detection" and allows you to configure the way
in which the system decides (or doesn't decide) a neighbour has become
unreachahle.

```

route - look at or add routing entries

Here we can set route attributes and types as well as just configuring entries. Types of route include `unicast`, `unreachahle`, `blackhole`, `prohibit`, `local`, `broadcast`, `throw`, `nat`, `anycast` and `multicast`. NAT is explained in chapter ten.

```

ip route add {to <prefix>| to <type> <prefix> | tos <TOS> |
<dsfield TOS> | metric <n> | preference <n> | table <table
id> | dev <name> | via <address> | src <address> | realm
<realm id> | mtu <n> | window <n> | rtt <n> | rttvar <n> |

```

```
ssthresh <n> | cwnd <n> | advmss <n> | nexthop <next hop> |  
scope <scope val> | protocol <rtproto> | onlink | equalize }
```

The only new things here are `rtproto`, which describes whether a route is acquired by redirect, from kernel autoconfiguration, at boot time, through static addition or by router discovery; and the multipath parameter, `equalization`.

maddress - manage multicast addresses

mroute - manage multicast routing cache entries These two are similar to the unicast address and route management command options.

tunnel manage tunnel entries

rule manage tunnel entries These last two are covered in more detail in chapter ten.

8.9 Summary

In this chapter, we have looked at routing and the forwarding lookup support needed in a modern system, for fast packet processing. We will see later on that there are other ways in which packet headers need to be matched. In particular in chapter ten, we take a look at firewall protection, where filters can be defined to exclude (or exclusively include) packets of certain patterns.

In the next chapter, we look at another part of the forwarding process, that of providing specified performance (“treatments”) to different packet flows.

Chapter 9

Forwarding

“Sometimes in order to be correct an implementation must meet certain performance constraints.” - Radia Perlman, 1998.

9.1 Roadmap

This chapter is about how traffic is controlled in the Internet. The key concept associated with Internet traffic is the idea of a *flow*. A flow is a sequence of packets from a source or related set of sources to a receiver or set of receivers. The simplest instance of a flow is a single TCP connection. A more complex flow might be a Virtual Private Network, which might be implemented as a collection of paths from a collection of sites requiring some overall resources (minimum rate) but no specific treatment for an particular packet or TCP connection.

There are three main models for traffic management in general:

end-to-end end-to-end traffic management is either adaptive or non-adaptive - adaptive applications and users employ congestion avoidance, as implemented by TCP, for example, as described in chapter seven. TCP reacts to congestion signals from packet loss, or from explicit congestion notification. These signals are generated by routers through queue management procedures which we look at in this chapter.

Non-adaptive applications and users simply send what they have when they have it - usually, video, audio and gaming applications are non-adaptive, and employ UDP (usually with RTP above it) to carry packets. These applications require some type of network path that is engineered for their requirements. This can be done by provisioning the path well enough when it is installed, or by some type of dynamic resource reservation near the time the path is needed. Resource reservation requires traffic to be given some type of preferential scheduling in routers. We'll look at this too in this chapter. Resource reservation is carried out using some type of signaling protocol such as RSVP or else is done out-of-band by phone call, fax or email to the network provider who then uses network management to configure routers along relevant path or paths. This installs *state* in these routers associated with the sources and destinations requiring this preferential treatment.

edge-to-edge It is currently common for networks to have plenty of capacity in the “core”, but be limited in the access (dial ip and network access points). In this sort of situation it is possible to manage traffic in many places in the networks in large *aggregations*. This type of traffic management is called “Differentiated Services” and is the subject of much work in the Internet today. As discussed in the previous chapter, routers already aggregate table entries for the basic forwarding task. It is possible that preferential performance requirements for traffic for many sets of sources and sinks follow similar aggregation patterns, in which case the *state* associated with giving this traffic different scheduling and forwarding treatment can be kept in small tables. If this is not the case, and one needs this information for specific flows, then this is usually referred to as “Integrated Services”.

hop-by-hop The basic building block that implements different treatment for different flows is the actual forwarding treatment within a given hop or router. A router can apply a variety of different scheduling or queueing algorithms to a sequence of packets. The default schedule is FIFO, or “first in first out”. The default queue management procedure is to drop packets that arrive at a full queue.

There are lots of ways the queues can be implemented. For example, routers can queue on input or output or a hybrid mixture. Most systems, Linux included, are output queued. There are other procedures required to monitor the queue occupancy, to decide what to do on overload, to mark or drop traffic that has behaved badly, to re-condition a sequence of packets to reduce its burstiness (“shaping”) and to coordinate these actions with other routers to provide a consistent and fair service

This chapter covers those aspects of the forwarding treatments described above currently in the Linux 2.4 kernel.

9.2 Introduction

The problem with vanilla IP is that it provides merely a “best-effort” service. When we say best-effort, what we really mean is that the service offers no guarantees. The word “best” is misleading. In practice, a network path made out of arbitrary links and routers, with an arbitrary mix of traffic has unpredictable performance. The available capacity (whether in bits per second, or packets per second) will vary packet by packet. The delay experienced at each hop, and end to end, will vary packet by packet. The chances of a packet being lost due to noise or congestion will vary moment by moment.

In practice, the situation are not that terrible. While the performance is unpredictable, most applications use TCP for end-to-end services, which provides reliability and carries out congestion avoidance and control, as described in chapter seven. There are two main contributions to traffic variation, which will influence the network resources available to a given IP flow, and therefore the performance seen by a user: firstly, when TCP connections start, they expand the rate they send at fairly quickly; secondly, at some times of the day, there are a large number of connections starting (or finishing) at roughly the same time (e.g. start of day, or just after a new web page is announced at a

site or on a popular email list). Connection durations vary - most of the Internet traffic is normally web downloads, and web pages follow a Zipf distribution of page-size. Having said that, we should allow for web page popularity as well. Most of the time, there is a mix of shorter and longer TCP connections, known colloquially as *mice* and *elephants*, with the total number of connections varying relatively slowly. This means that traffic in the center of the network varies relatively slowly, unless there are correlated events. Actually characterising the behaviour of a mix of TCP flows is quite a large topic of research, and well outside the scope of this book. To make matters more complex, there are a number of applications which are emerging which are based on UDP, which do not generally respond to congestion signals necessarily. If traffic from these grows, then “best effort” may become even less predictable.

Traditionally, then, IP data transfer has followed the traditional view that datagrams are strictly individual packets, with no relationship between one packet and the next. There was no recognition of flows: the fact that a sequence of packets with some fields constant might be recognized, or even externally classified and specified as belonging to some session or set of sessions that want consistent treatment, and some kind of performance guarantees was overlooked. In fact the requirement for this sort of specification is present not only for reasons of performance, but also for security, as we will see in chapter ten.

Thus the spirit of the connectionless network, with no signalling, was preserved for over two decades of the Internet. This enabled very simple minded implementation of the forwarding task in a router, based on per-datagram forwarding, with extremely simple table look-ups. We saw in the previous chapter that the sole basis for figuring out what to do with a packet is normally just the destination address of the packet. We also saw in chapter three that the simplest way that a packet can be forwarded is to append the buffer to an output queue. In this scenario, there is no examination of the type of traffic. This means that there can be special treatment, or no priority traffic. Of course, in an underloaded network, as each packet is appended to a queue, the packet from the head of the queue is being transmitted, or has already been sent, so the variation in delay can be quite low. Also, the available capacity to a given flow is, by definition, more than adequate in an over-provisioned network - supply exceeds demand.

In practice, most links in the Internet are not over-provisioned. Some core networks may be, but simple economics mean that access links are always close to what is affordable. Even in wealthy parts of the world, given we often provision links on quite long intervals (e.g. months or years) and demand keeps growing, it's likely that near the end of any provisioning period a link is overloaded. Dial-up and other slow links are always lower capacity than any modern computer system can transmit at. TCP sources will always attempt to fill up such links.

Another facet of the Internet that leads to unpredictable performance is also a result of a design strength. The Internet routing system is opportunistic. What this means is that almost as soon as a better route appears (perhaps because a new link is installed, or a broken one repaired), the routing system may alter the path taken by packets from source to destination, leading to a step up (or down) in performance, e.g. in delay or throughput or errors.

Dynamic routing changes occur reasonably frequently in today's Internet. Given that a typical route can only route packets for a given destination to a

single next hop at any one time, this means that we do not see fixed-paths. The consequence of this is that we also see no fixed quality of service (QoS). QoS is a much over-used term - what we really mean is not quality, but quantitative performance guarantees, which may be absolute, or relative percentage of capacity, or delay, or may be expressed in terms of statistics (averages and chances of deviations).

As we imply above, the combination of source behaviour (whether TCP or UDP based), the traffic patterns made by the collection of active sources and the destinations they talk to, and the dynamic routing behaviour of the network, leads to the performance seen by any given flow as highly variable. At the level of a given hop, this is because the packets are being mixed together after the routing decision has been made into a single, first-in, first-out (FIFO) queue.

We can start bottom up, to fix this, by adding more sophisticated scheduling in the routers. Rather than FIFO queues, which offer a first come, first served (FCFS) model and no examination of type of traffic, we could start to offer some differentiation in the way that we *treat* different packets belonging to different flows.

At the most intuitive level, we can think of the problem as being the lack of priority for traffic. When a link is busy, who gets to go next? Do packets form a simple “fair” queue behind each other, or do we think in terms of keeping some history about whose packets are whose, and make sure that flows get a fair share of a link? If we don’t do this, some source sending packets twice as fast will get twice the chance at the link. Another consideration is that we may *wish* to give packets an advantage over others. If so, how do we mark packets to indicate priority?

Luckily, a field in the IP packet header was set aside early on for indicating Type of Service. Although this was not widely used across Internet in early implementations of host and router software, it is now seeing a new surge of interest.

From some users perspective, the requirement is not to give priority or fair service simply to a single flow. It is often more immediately important to offer some service guarantees to aggregations of traffic, For example, an organisation may need some minimum throughput for all the traffic from a particular source, but to any destination, or all traffic to a particular destination, but from any source, or both from and to specified sources and destinations.

The IP FIB holds data for figuring out where to send traffic next based only on destination IP prefix which is one form of aggregate. What other indication of aggregation might be useful? Over the last decade there have been many attempts to answer the questions:

- Can we do better than best-effort?
- What support do real-time flows need in the network?
- What support can we provide in the network?
- Alternatives to FCFS?

At the same time, the Internet community has also been adding facilities for mobility, and many to many communication, which should not be excluded from enjoying any enhanced service models. To access these enhanced services, we need to consider end-system application-level interfaces, remote access to router

level support through in band signaling, out of band signaling, or management. Above all, no new mechanism should threaten the scalability of the Internet too much to be worth offering. The other side of the QoS coin is that there must be incentives or pricing schemes for services that control use.

In practice, IP runs on links, and there are many different link level technologies employed in the Internet. In their native form, these link layer technologies offer a wide range of different capabilities in terms of performance guarantees. For example a naked Ethernet LAN has non deterministic delay. On the other hand, a point-to-point serial line has a well known delay made from two components: the time to serialise a packet, and the propagation time for the packet over the distance the link traverses.

The Internet is inherently heterogeneous (aggressively so!). There is no common layer two, thus there is no well defined support for QoS without defining the mapping between any IP level mechanism and each and every link layer in use on any path that is to offer an enhanced service over best effort.

9.2.1 Integrated Services

One of the great debates in the Internet community has been about QoS requirements. It is certainly not well understood what users will use (and pay for). In the early days, the first steps towards enhanced services was the definition of the Integrated Services Packet Network (ISPN). Here a standard set of QoS service-levels was defined, with detailed service type descriptions. The Service interface includes a complex signaling protocol called RSVP, which we look at briefly later. This is out-of-band in the sense that it is sent separately from users data packets. In telecom terms it is “in-band”, in that it travels down the same wires as the users data, but it is additional to the users data. Such a signaling protocol is assumed to be the interface to a network level mechanism called *admission control*. Admission control is basically a decision function that uses the information from a signaled request, together with current and recent information about the conditions in the network to compute whether the service requested can be satisfied. If so, the reservation, or “call”, is accepted or admitted, otherwise it will be rejected. In IP networks, if a reservation is rejected, this does not stop someone sending packets. It merely means that they will only be forwarded with best effort. If a reservation is accepted, then all the routers along a path must respect the new service level for the traffic - in other words, they all have to be able to recognize the traffic belonging to the reservation (call or flow - all the terms are interchangeable). If the reservation needs preferential treatment, all the routers along the path need to apply the appropriate scheduling rules to packets from the flow, offering prioritisation and differentiation of its traffic.

As we will see later in this traffic, Linux offers a very sophisticated choice of scheduling mechanisms to achieve this.

This a QoS service-level is associated with quite a lot of additional functionality, including application flow description, traffic description, packet handling as well as policing to make sure users or applications dont misuse their reservation.

The service interface for integrated services has a set of common procedures, data structures and parameters, including a signalling protocol, the admission control function to check request can be honoured, and the scheduling systems,

packet classification to provide prioritisation of traffic as well as queue management.

9.2.2 Admission Test for Integrated Services Guaranteed Service

[Table 17 about here.]

$$d = \frac{b}{g} + (N - 1) \frac{P_F}{g} + N \frac{P_N}{r} \quad (9.1)$$

One example of this is illustrated in figure 809, which plots the capacity needed for typical 64kbps PCM coded voice calls packetised using RTP and a range of typical Internet line speeds, a delay bound as needed for interactive voice, and the over-allocation needed. Note the presence of hop count as part of the equation.

[Figure 812 about here.]

The problems with this equation based source specification are many and various. Some include the following:

Bucket Rate An application can choose to send at many different rates - audio, video, gaming and data applications can all be configured to correspond loosely to some user perceived quality. However, this correspondence has to be characterised for each and every application, and for a wide range of values of the data rate. As new coding and compression techniques emerge rapidly, this means that it is very hard for the user to choose a meaningful number here.

Burstiness Just as the rate is hard to assess, the burstiness is even harder - this is also dependent on content as well as implementation.

Packet size The packet size, and other parameters of note are an artifact of the network design as well as host computer software. Why should the user be concerned with choosing or setting these?

One problem with this type of source model is that it is over-specified. Another more serious problem is that it appears to be oriented more towards optimising for the network provider, not the user. Lastly, asking the user to state parameters including network specific ones requires the provider to reveal internal factors about their network - this may undermine their competitive edge with other providers.

9.2.3 Network Architecture

There are quite a few assumptions in the Internet work on adding other service models about network hierarchy. In general one of the main assumption to make any sort of traffic management tractable is that the network is organised into well defined levels. This then allows appropriate techniques to be deployed in the right tier of the network.

Access network Here there is low multiplexing (tens or hundreds of flows at any given link or router), and on any given link low traffic rates (compared to typical computing systems performance).

Distribution network Here, ISPs provide interconnectivity at local level, and traffic rates are medium. Again, there is a low level of multiplexing on any given link (hundreds or thousands of flows).

Core network backbone This is where we see a high volume of traffic, and very high degrees of multiplexing. Here is where in today's Internet, we see systems approaching the limits of electronics (and pure hardware switching) let alone software systems.

[Figure 813 about here.]

Between these various tiers, we see administrative boundaries. Each tier is known as an “Autonomous System” (AS). This simply means that it is managed irrespective of the management rules internal to the other systems. Typically, an AS makes its own choices of intra-domain routing schemes, internal policies for security and traffic engineering, including routing metrics and link preferences. Intra-domain protocols might be RIPv2 or OSPFv2 or ISIS.

Between these domains, we provide interconnection of ASs via inter-domain routing. Here interconnectivity information is exchanged (and constrained) using the Border Gateway Protocol (BGP).

When enhancing services over best effort, the hierarchy and structure of the network interconnect, and differences in considerations at different levels, and intra- versus inter- domain have all to be understood and respected.

In general, resource reservation (sitting above, or “after” routing), and traffic engineering (sitting below, or “before” routing) are complex research topics.

9.3 Mixing Traffic onto the wire

In a better-than-best-effort Internet, different applications will expect different performance. First we have to understand what different applications offer in terms of traffic (generation) profiles. This can be captured by looking at the packet size distributions, and inter-departure times for packets from a source. We then need to understand how these timings should be reproduced at the far end so as to meet traffic timing constraints.

Routers that use FCFS queues have no knowledge of application, and no knowledge of traffic patterns. If different traffic types share same network path, we can see what occurs by considering three different applications as illustrated in 811.

[Figure 814 about here.]

At the router, we see 3 input lines. In an FCFS system, these would be serviced round-robin at router. Let's assume the traffic is mixed onto a single output line with a single output buffer. Assuming the line speed on output is the sum of input line speeds, we can see what happens to the traffic. The steady audio rate at the source ends up being severely disrupted by the variable rates and larger packet sizes of the other applications, even though the average rate

of all the applications is respected. This is a scenario without any congestion. If there was any overload, we would also incur loss, which could be very bad for an audio application..

[Figure 815 about here.]

If we chose to consider different output line speeds, it would not affect the pattern of the relative spacing of packets - just the absolute. At one extreme, the line speed could be so fast that the largest packet possible takes less time to serialize and transmit as the smallest packet on the input lines. In this case, the second (and other) audio packets would “work ahead” of where they “should be”.

In a full network, we need to consider these sorts of pictures but with the overall traffic pattern. Different applications have different source models, and the receive side expects some similar arrival pattern. There is also the possibility that there are many uses of an application (some e-mail users even expect near realtime turnaround of e-mail responses, while some audio users do not mind long delays before audio response or playout).

The rules for traffic aggregation, as you move through the tiers in the network hierarchy are not obvious. One might expect life to be easy if the network traffic follows the hierarchy. In other words, provided we see more multiplexing as we view the network from access towards the core, then perhaps the chances of disruption caused to one flow by others might decrease. However, it is very hard to model this. In fact it is not at all obvious that this is right except in very simple, and from experience, rare circumstances. The Internet has a habit of forming hot spots which move with alarmign frequency - this makes such assumptions frequently invalid.

9.3.1 Performance Parameters

We can take a top-down view of how to improve over best-effort service, by thinking about user needs, instead of just router mechanisms. There are some obvious performance parameters we need to consider. Some of these are input to the performance decision process, others are results of it:

Packet sizes A large packet size is good for general data, since it amortizes the packet header overhead (and packet handling overheads) efficiently over a lot of data. It is router friendly, but leads to higher chances of slow down for real-time traffic,

On the other hand, a small packet size is good for real-time data since it entails less delay at source (collecting audio or video samples, or text input), and lower hop -by-hop serialisation times which leads to less end-to-end delay. For a given stream of flow, small packets also have a better tolerance to loss. They will possibly lead to less jitter. Howeverm they are less efficient in terms of header overhead, and not router-friendly (in terms of input interrupts/packet processing time).

Delays The end-to-end delay, the delay a user is concerned with is made of several components: The propagation time (at the speed-of-light or close);

The transmission time, made from the data rate, and the network elements' (router, switch) contributions including buffering (queuing), processing and so on, as well as pure end-system processing, which will be application specific.

Delay bounds Could an enhanced service include a possible delay bound?

Well, there is no choice in the speed of light! Also, in today's Internet, paths are not known by the end user in advance, and are subject to change due to dynamic routing. The influence of other traffic and varying traffic patterns, including localised traffic and time-of-day effects means that deterministic delay is often impractical, but not impossible.

delay variation and End-to-end jitter Variation in delay is the per-packet delay changes. It can be contributed to by effects at receiver, such as variable packet arrival rate and variable data rate for flow. For non-real-time applications, this does not represent a problem. For real-time applications, a receiving application may need jitter compensation.

Causes of jitter include Media access (LAN, and Modem access delays for asymmetric codec lines) and FIFO queuing where there is no notion of a flow. For traffic aggregation, a mix of different applications the variation in load on routers due to localised load/congestion and dynamic path changes can all lead to jitter.

loss End-to-end loss can be compensated for by Non-real-time by re-transmission, e.g.:TCP. For Real-time applications, it is possible to mask loss by using forward error correction and redundant encoding. It can also be this can be via media specific fill-in at receiver. If loss persists, it is often an indication of congestion. Many newer real-time applications are adaptive: they adjust flow construction, and the resulting output rate according to a "TCP-rate-equivalent" formula, so that they look, to a network provider, like a smoothly varying TCP source.

Causes of loss include packet-drop at routers due to short or medium or long term congestion, or traffic contract violations caused by mis-behaving sources. More systemic, but less sinister causes of problem can be source synchronisation, and excessive load due to failure in another part of the network or simple abnormal traffic patterns, e.g. new download.

Note that packet re-ordering, due to misbehaviour of a load balancing scheme, or misconfiguration of routers, or odd interactions between re-transmission and re-routing, may be seen as loss by some receivers.

Throughput Throughput is also often referred to as bandwidth, wrongly but we're stuck with that.

The end-to-end data rate is subject to short-term changes, during the lifetime of a flow over say seconds (on the order of round trip times) as well as long-term changes, during the course of a day on the order of hours

This is partly a result of protocol behaviour such as TCP congestion control and flow control, partly because of human reactions to perceived performance, and partly because of the normal daily patterns of life and Internet usage.

Changes to the data rate available on a path occur for several specific reasons. Different connectivity may become available, the dynamic routing system may then choose different paths, and it may even do so in response to router-based measurements of loss and/or delay.

9.3.2 Probing or measurement based admission control?

Can we use probes to detect delay, jitter loss and data rate?

There are of course a well known set of network tools such as ping, traceroute and pathchar, which will report the current end-to-end delay and loss statistics, the delay hop by hop along a path and even estimate the link capacity on all the hops on a path.

Probes load the network and therefore they affect the system being measured. Measurement is tricky! They are useful for medium- to long-term diagnostics, and will give some indication of whether a particular application will get its required network performance for the nearterm future. However, they don't offer any differentiation or protection against vagaries of other peoples behaviour.

9.4 Elastic and Inelastic Applications - tolerance for variation?

We can broadly partition applications into these two classes.

Elastic Examples of elastic applications (patient!) include:

E-mail, which is asynchronous (in the very general, user-centered meaning of the term). A message is not real-time, delivery in several minutes is acceptable; File transfer, which is interactive service, and requires quick transfer but a slower transfer may be acceptable; Network file service, which is an interactive service, similar to file transfer, and where a fast response is required (usually over LAN); WWW, which is interactive, and is really a combination of a user interface and a file access mechanism(!), where a fast response is required; in the latter case, of course, one might switch to a different model when QoS sensitive content existed on WWW pages.

[Table 18 about here.]

Inelastic Examples of inelastic applications (impatient!) include:

Streaming music, video and voice, which are not interactive, and where end-to-end delay not important, and also end-to-end jitter not important, and finally the data rate and loss *are* very important; Real-time voice, which is person-to-person, and interactive and it is extremely important to control the end-to-end delay, the end-to-end jitter, the end-to-end loss and the end-to-end data rate.

[Table 19 about here.]

We can fill in some of the detail for these sets of application (and usage) requirements by specifying the parameters for QoS for them:

Delay As we discussed earlier, it isn't really realistic to request maximum delay value because there is no simple user control over end-to-end network path in IP. It *is* possible to find actual values for the maximum end-to-end delay, DMAX in some cases, as well as the minimum end-to-end delay, DMIN.

Jitter It is not possible in general to request end-to-end jitter value since this would essentially be equivalent to turning a packet switched network into a Time Division Multiplexing system, with all the associated inefficiencies. However, we can approximate the maximum jitter from DMAX and DMIN. We can evaluate DMIN dynamically, and choose some sensible statistical bound for DMAX, for example the 99th percentile. The jitter value has a contribution from the transport-level and below, so protocols there have to convey this information (see the discussion on RTCP in chapter eleven), as well as a contribution from the application-level.

Loss Loss is not really a QoS parameter for IP networks. It is not something you really want to request as such, it is more something you want to avoid. How would a router honour a request for a specified loss level? Loss is in any case related to the data rate. It is very hard to guarantee. Probabilistic loss levels may be feasible. As a better understanding of the relationship between loss and the congestion feedback signal emerges for TCP and TCP-like transport protocols, we can imagine using loss (e.g. via RED; see later) to provide some level of control of best effort, achieving more scalable and accurate traffic management and congestion control.

Packet size Path MTU discovery (see chapter six) reveals the smallest maximum transfer unit that the links on a given route support. The application may use smaller packets, but using larger packet sizes than the MTU (and incurring fragmentation) is regarded as a bad idea. The user's packet size choice, and the path MTU, may be used by routers to help with buffer allocation and are also vital parameters to the delay evaluation process in admission control.

Data rate This is the key performance parameter, but how can we specify this for applications that are not fixed (constant bit) rate? Data applications are inherently bursty. Voice and other multimedia sources use variable rate compression and silence suppression, so they do not send at a single rate. How do we capture the rate information?

We can specify the mean data rate, the peak rate, the length of time a source spends sending traffic at the peak rate. How do we then allocate resources? If we allocate at peak rate, we waste resources when an application sends slower. If we allocate less resources, we take a risk.

Real-time flows may be constant bit rate, but can also be variable bit rate. A given application-level flow is made of a number of application data units (ADU) and we can specify a rate in terms of these. We would like the API for programmers for this to be both application-friendly as well as technology neutral.

9.4.1 Network Source behaviour Characterization: Leaky and Token Buckets

The most common ways to capture the definition of a traffic source behaviour is via a leaky or token bucket specification:

Leaky Bucket This has two parameters:

$$B : \text{bucketsize}[Bytes] L : \text{leakrate}[B/sorb/s] \quad (9.2)$$

Data pours into the bucket and is leaked out, with traffic always constrained to rate L . B/L is maximum latency at transmission.

[Figure 816 about here.]

Token bucket This has three parameters:

$$b : \text{bucketsize}[B] r : \text{bucketrate}[B/sorb/s] p : \text{peakrate}[B/sorb/s] \quad (9.3)$$

Bucket fills with tokens at rate r , starts full. The presence of tokens allow data transmission, and a burst is allowed at rate p , while the amount of data sent is limited to be $< rt + b$

[Figure 817 about here.]

9.5 Packet Forward Scheduling and Queue Management

Linux has a complex support for scheduling and queue management. Let's start out by looking at these in the abstract:

In the simple router schematic below in 815 we see a router with a number of input lines. These have no input buffering (at least nothing outside of what is on the actual NIC). We see a packet classifier, which can employ information from policy-based classification, to select the correct output queue, once the forwarding/routing tables have indicated the right output interface. Between the input network interface card and output network interface card, there is a switching fabric. Of course, on a PC platform, this is nothing so grand - its a PCI bus. Packets are taken from the output buffer (queue) by a scheduler, which chooses which packet from the output queue is to be serviced next. The packet scheduler (in linux) is driven from interrupts, of which we'll say more later.

[Figure 818 about here.]

9.6 Queueing Disciplines

The default, simplest queueing discipline we can imagine is probably FIFO (First In First Out) or FCFS (First Come First Served).

This employs a Null packet classifier. Packets are queued to outputs in order they arrive, and there is no packet differentiation. There is no history or state, and, as a result, no notion of flows of packets. Anytime a packet arrives, it is serviced as soon as possible. FCFS is what is called a “work-conserving” scheduler. This is defined as a scheduler which is never idle so long as any packets are waiting. Work conserving schedulers have the property that if you wish to reduce the delay of one flow, you necessarily increase the delay of one or more others. An interesting useful result to remember is that we can not give all flows a lower delay than they would get under FCFS with any other scheduling scheme.

One can imagine “non-work conserving” disciplines. These are schedulers which can be idle even if there are packets waiting. In principle, this allows for some possible smoothing of packet flows. Instead of serving a packet as soon as it arrives, we wait until the packet is eligible for transmission. This eligibility could be defined by a fixed time per router, or a fixed time across network. The result of this (for a given flow) is less jitter. It makes the downstream traffic more predictable and the output flow is controlled, leading to less bursty traffic. Another consequence then is that we need less buffer space. In the router, work conserving schedulers require less buffer in output queues, and the reduction in jitter means that end-systems need little or no de-jitter buffering. However, it necessarily leads (for the same network) to higher end-to-end delay, and can be complex in practice, possibly requiring clock-synchronisation at routers. In fact, at one extreme, a time-division-multiplex (digital circuit switched network) is an example of this approach.

There are a number of scheduling dimensions:

Priority levels If we implement a priority system how many levels should there be? It is not obvious why you want more than two or three. Since higher priority queues service first, this approach can cause starvation lower priority queues, unless we have some sort of admission control for the overall occupancy of the higher priority service in the network, or on any given path.

Work-conserving or not We must decide if delay/jitter control is required. Is the cost of implementation of delay/jitter control in network acceptable? In practice it isn't.

Degree of aggregation What is the flow granularity? Do we have per-application flow or per-user or per end-system? It is a question of tradeoff between the cost of implementation of the scheduling system (in terms of state, state setup messages, classification frequency and complexity) and the granularity of control.

Servicing within a queue Can we get away with FCFS within queue? Should we check for other parameters? What other added processing overhead may there be (e.g. network management/logging). What kind of queue management do we need in an “emergency”?

9.6.1 Common Scheduling Disciplines

Let's now look at some of the more common systems for picking which packet to send next, and when to send it.

Priority Queueing We have K queues: $1 \leq k \leq K$

Queue $k + 1$ has greater priority than queue k . In other words, higher priority queues are serviced first. This is very simple to implement and has a low processing overhead. It offers relative priority in practice (assuming we keep no history), and this offers no deterministic performance bounds. In terms of fairness and protection, it is not max-min fair and it hazards the risk of starvation of lower priority queues.

Generalized Processor Sharing - GPS This is a work-conserving scheme, and on paper: it serves an infinitesimally small amount of data from flow i and then visits the next flows round-robin. This provides a max-min fair share to each flow. It can provide weighted max-min fair share, however, in practice it is not directly implementable. Instead, it is used as a baseline abstract reference for comparing other schedulers.

Weighted Round Robin This is the simplest attempt at an implementable approximation to GPS. The queues are visited round-robin in proportion to weights assigned. Since different flows have different mean packet sizes, the weight has to be divided by mean packet size for each queue. Because mean packets size unpredictable, this may cause unfairness.

The service is max-min fair over long timescales. We must have more than one visit to each flow/queue. This means that there are deviations from fairness for short-lived flows.

Deficit Round Robin DRR does not need to know mean packet size. Each queue has deficit counter (dc): initially, this is set to zero. The DRR scheduler attempts to serve one quantum of data from a non-empty queue:

```
if size <= quantum + dc
packet at head served
dc = quantum + dc - size
else
dc += quantum
```

Queues that are not served during a round, build up credits. The quantum is normally set to be the maximum expected packet size. This ensures that one packet is served per round, per non-empty queue. *RFB* : $3T/r$ ($T = \text{maxpktservertime}, r = \text{linkrate}$) It works best for flows with small packet sizes and is fine for small number of flows.

Weighted Fair Queueing The WFQ scheduler is also based on GPS. It is effectively a GPS emulation, which is designed to produce finish-numbers for packets in the queue. The simplification is that scheme is a packet timescale instead of the GPS timescale which serves packets bit-by-bit round-robin. The finish-number is the time that a packet would have completed service, under the ideal (bit-by-bit) GPS. Packets are tagged with the finish-number, then the smallest finish-number across the queues is served first in the next round.

The round is a cycle of execution defined by an idealized bit-by-bit round-robin server. The finish-number calculated

from round number. If the queue is empty, the finish-number is the $numberofbitsinpacket + round - number$. If queue non-empty, the finish-number is $highestcurrentfinishnumberforqueue + numberofbitsinpacket$.

When a flow completes (empty queue), there is one less flow in the round, so R increases more quickly. This, as more flows complete, R increases more quickly etc. etc. Because of this, WFQ needs to evaluate R each time packet arrives or leaves, which can be a significant processing overhead.

In addition to the scheduler, we have to have a buffer drop policy to handle overloads. If packets arrive at full queue, we can drop packets already in queue, in order of decreasing finish-number for example. This can be used for best-effort queuing, or for providing guaranteed data rate and deterministic end-to-end delay. WFQ is used in real world, as is DRR. There are several alternatives also available, including self-clocked fair-queuing (SCFQ), worst-case fair weighted fair queuing (WF2Q) and a neat statistical approximation, stochastic fairness queueing (SFQ).

9.6.2 Class Based Queueing

As well as organising scheduling with a traffic class, we need to have a way of managing a set of different traffic classes, potentially with a range of different, appropriate scheduling algorithms. There are two specific approaches to this commonly today, and Linux supports both within a more general framework as we shall see later. One is Class Based Queueing, and the other is the so-called “Clark/Shenker/Zhang” architecture (after the three well-known researchers who devised it).

What is needed is a framework for organising hierarchical link sharing. Firstly, link capacity is shared between organisations, with differing levels of aggregation. There is no reason to suppose that the scheduler and aggregation level follow the same hierarchy, so it is necessary to support both class-based allocation, and policy-based class selection. Since the Internet is all about sharing resources, it should be possible, within the constraints of policy, to use the class hierarchy to assign capacity/priority to each node, but also to allow a node to borrow any spare capacity from parent. Fine-grained flows are possible. CBQ is this is a queuing mechanism which employs a set of schedulers with the classification framework, as illustrated below in 816.

[Figure 819 about here.]

9.7 Queue Management

Having revised scheduling, which addresses the problems of which output queue to visit, and which packet to transmit from output queue, let’s now take a look at queue management.

Queue management entails the tasks of ensuring buffers are available, using memory management to look after resources, and deals with the details of organising packets within queue, and especially packet dropping when queue is full, for example when triggering congestion control.

Congestion is typically caused (as we have seen before) by a range of misfortunes including: misbehaving sources; source synchronisation; routing instability and network failure causing re-routing.

Congestion could hurt many flows, so we need to manage it somehow. Typically, we drop packets. We can also mark packets with the newly defined ‘Explicit Congestion Notification’ bit. Key questions are do we drop new packets until queue clears, or do we admit new packets, drop existing packets in queue? How much history, explicitly or implicitly concerning “misbehaving flows”, do we keep?

Packet drop mechanisms include:

Drop-from-tail Given a FIFO queue, tail-drop is clearly easy to implement.

Of course delayed packets at within queue may expire.

Drop-from-head This may be harder to implement (especially if some of a transmit queue is no longer owned by the IP level but is in the buffer owned by (or even on) the NIC. The idea here is that old packets purged first, and this ought to be good for real time, and even better for TCP. In practice its not widely implemented

Random drop Random drop, especially in its Random Early Detection guise, is popular.

If the random number is chosen carefully, to be uniform (proportional to the queue length), this is in some senses, fair, since if all sources behaving they are equally likely to see the drop; if there are misbehaving sources, then they will be proportionally more heavily penalised.

Flush queue When a queue is overloaded, there is also the option to drop all packets in queue. This is simple, and would mean that all the flows that are adaptive should back-off. However it is inefficient, and possibly unsafe, since all the backed off flows will startup again in a synchronised fashion, unlike random drop.

Intelligent drop Various Fair drop mechanisms have been proposed, sometimes based on level four (transport specific) information such as port numbers. These may need a lot of state information, although it should be possible to make them fairer.

The reaction to packet drops is well defined for non-real-time applications using TCP. Packet drop is to be interpreted as congestion which leads to a slow down in transmission rate, as described in chapter seven under Slow start and congestion avoidance. This leads to a happy network!

Real-time UDP flows have less obvious choices. For transient loss, packet drop can be concealed by an application specific fill-in at receiver. However, we still need to employ an application-level congestion control. Nonetheless flow data rate adaptation may not be suited to audio/video. Real-time flows may not adapt but then this hurts adaptive flows. Queue management could protect adaptive flows, so perhaps smart queue management required.

9.7.1 Random Early Detection

Random Early Detection is a refinement of simple random drop. The intention is to spot congestion before it happens. The idea is that a dropped packet can

be used to provide a pre-emptive congestion signal. Since this means that the source slows down, it prevents real congestion. Which packets to drop? Well, we can monitor flows specifically, but that leads to a high cost in state and processing overhead vs. overall performance of the network.

The probability of packet drop is related to queue length. If we monitor the queue length value via an exponential average, we smooth the RED process reaction to small bursts. This punishes sustained heavy traffic. Packets can be dropped or marked as offending, via ECN. RED-aware routers are more likely to drop offending packets. Source must be adaptive. Setting RED parameters is quite complex.

9.8 Linux Forwarding Treatment Architecture

The bulk of the code in the Linux implementation is by Alexey Kuznetsov, and is an excellent example of the use of C for both low level efficiency, and high level clarity. Much along the lines discussed above, the code provides comprehensive support for best effort, differentiated and integrated services forwarding, as well as clean interfaces for adding experimental facilities.

The code is heavily oriented around an “Object Oriented” structure, which can be confusing since the word “Class” is used in a special way in QoS, for example consider the terms: Quality of Service, Class of Service and Grade of Service.

We’ll refrain from using strict object oriented programmers terminology when talking about this code (especially since it is actually written in C). However, it is useful to bear it in mind if you have been trained that way.

9.8.1 Output Interface

The way that the sequence of queueing disciplines is invoked in Linux is illustrated in figure ???. Once the route has been decided for a packet, the output interface is known. Then the packet is queued for output on the device by calling the `enqueue` function for the device. This invokes the `wakeup` function in an effort to send the packet just sent. Of course it is entirely possible that there are earlier packets queued that were too early, or that this packet is too early. So the next step is that `wakeup` calls `restart` which attempts to find any packet from the queueing discipline for the device that is ready. If there are any, `restart` calls the device specific `hard_start_xmit` function to transmit the packet. If there are output failures, the packet is put back in the pool of packets ready to go by calling `requeue`.

[Figure 820 about here.]

There are (as is typical in networking code) two other ways that a packet may get pulled out of its queue and put through this process. Firstly, a timer based event may fire that will call `wakeup`. Certain queueing disciplines require this as we’ve seen earlier. Secondly, the `net_bh` will invoke `wakeup` anytime it has finished processing any packets that it queues. These two together make sure that we never forget any packets in the queue!

9.8.2 Queue Discipline

Any queueing discipline involves a standard collection of functions. The actual specifics depend on the disciplines, which we have described early and will list and walk through below. Here we just describe the generic `Qdisc_ops` interface:

init Initialize the queueing discipline

destroy tear down and remove the queueing discipline, clearing the classes, filters and events, and freeing up associated resources.

reset clears all queues in a discipline, cancels all timers and restores it to a pristine condition.

dump accesses configuration and state data from the queueing discipline.

enqueue adds a packet to the queueing discipline - see later for a discussion of how a class based queueing discipline works.

dequeue removes the appropriate packet from the queueing discipline (presumably for sending!). There may, of course, not be any packets ready.

requeue restores a dequeue packet to its original place in a queueing discipline, typically because of some output problem.

drop remove a packet from the queue.

Queueing disciplines are registered through the scheduling API (see later), or they register themselves if they start as modules. There is an interface between the route attribute structure (`rtattr`) and the queueing disciplines, which allows information to be exchanged via the `rtnetlink` interface to traffic management applications in an analogous way to the way that routing table data is exchanged between user space routing daemons and the kernel forwarding tables (fibs, described in the previous chapter).

9.8.3 Class/Type

One of the really nice things about the Linux traffic management facilities is the flexibility of the queueing disciplines. Part of this is due to the facility to have *class based queueing* disciplines. A queueing discipline can assign its own class, or the class can be defined by a user. Essentially, the class identifier space is normally used to create a classification hierarchy (although arbitrary graphs of classes of queues would in some sense be possible - its meaning, on the other hand, would be hard to ascertain!).

graft this adds a new queueing discipline to a class.

delete this makes sure no one is using the class, and then removes it.

change this modifies the properties of the class

dump_class this is the analog of dumping a queueing discipline - it returns class data.

get this looks up a class based on its class identifier and returns the internal identifier - it increments the usage count for aclass.

put this decrements the usage count for a class - if it reaches zero, the class may be removed.

walk this iterates over the classes of a queuing discipline and returns diagnostic data for all the classes in the queuing discipline.

bind_tcf this associates or *binds* an instance of a filter to a class.

unbind_tcf this removes an instance of a filter from a class. bind and unbind are similar to get and put (see code later).

tcf_chain this returns the head of a list of filters for a class.

9.8.4 filter, or classifier

Packets that are enqueued for a class based queuing discipline need to be classified before the correct queue can be chosen to put them on.

There are two main ways that this can happen. Firstly, a packet may be classified by the user, using the `skb->priority` field to fold a class identifier (seetable from a local application via a `setsockopt()` call). More normally, `enqueue` calls `tc_classify()` to find the correct classification.

This works by walking the list of filters associated with the classed based queuing discipline.

[Figure 821 about here.]

Filter functions:

init initialize a filter

delete removes a filter

dump return state information for a filter or filter element

get look up a filter

put

change modify filter properties

walk iterate over all elements of a filter and call

classify classify a packet according to a filter

Filters are also structured by IP protocol (`skb->protocol`).

[Figure 822 about here.]

9.9 The Classifier Code

9.9.1 Classifiers

Linux has a quite rich mechanism for classifying packets into the various queuing disciplines. First we look at the API to this system.

[Figure 823 about here.]

[Figure 824 about here.]

[Figure 825 about here.]

[Figure 826 about here.]

[Figure 827 about here.]

[Figure 828 about here.]

[Figure 829 about here.]

[Figure 830 about here.]

[Figure 831 about here.]

[Figure 832 about here.]

[Figure 833 about here.]

[Figure 834 about here.]

Classifier Statistics

[Figure 835 about here.]

[Figure 836 about here.]

[Figure 837 about here.]

[Figure 838 about here.]

Universal Classifiers

The u32 classifier is such a common thing to need that it has its own form.

[Figure 839 about here.]

[Figure 840 about here.]

[Figure 841 about here.]

And there are lookup and other functions:

[Figure 842 about here.]

[Figure 843 about here.]

Starting up the Classifier System

[Figure 844 about here.]

[Figure 845 about here.]

[Figure 846 about here.]

[Figure 847 about here.]

[Figure 848 about here.]

[Figure 849 about here.]

[Figure 850 about here.]

Managing the classifier, including walkign a classification tree and calling a given callback (e.g. to dump out statistics).

[Figure 851 about here.]

[Figure 852 about here.]

[Figure 853 about here.]

RSVP

[Figure 854 about here.]

[Figure 855 about here.]

[Figure 856 about here.]

[Figure 857 about here.]

[Figure 858 about here.]

[Figure 859 about here.]

[Figure 860 about here.]

[Figure 861 about here.]

[Figure 862 about here.]

[Figure 863 about here.]

[Figure 864 about here.]

[Figure 865 about here.]

[Figure 866 about here.]

[Figure 867 about here.]

[Figure 868 about here.]

[Figure 869 about here.]

Why oh why (alexey?) is all this in the header file not the C file!!!???

[Figure 870 about here.]

Includes the header above!

Route Classifiers

This type of classifier works by recognizing the *route* that this packet is taking.

[Figure 871 about here.]

[Figure 872 about here.]

[Figure 873 about here.]

[Figure 874 about here.]

[Figure 875 about here.]

[Figure 876 about here.]

[Figure 877 about here.]

[Figure 878 about here.]

Firewall Classifiers

This classifier is based on a firewall rule.

[Figure 879 about here.]

[Figure 880 about here.]

[Figure 881 about here.]

[Figure 882 about here.]

[Figure 883 about here.]

Changing the Classifier

[Figure 884 about here.]

[Figure 885 about here.]

[Figure 886 about here.]

[Figure 887 about here.]

TC Classifiers

These classifiers follow the traffic control rules.

[Figure 888 about here.]

[Figure 889 about here.]

[Figure 890 about here.]

[Figure 891 about here.]

[Figure 892 about here.]

[Figure 893 about here.]

[Figure 894 about here.]

[Figure 895 about here.]

[Figure 896 about here.]

[Figure 897 about here.]

9.10 queueing discipline

Linux 2.4 supports a wide variety of queueing disciplines, ranging from simple elementary schedulers, through to the complex, multi-tiered compound systems. The current list is:

FIFO - This represents the default, best effort Internet Service.

Priority - This is the simplest way to achieve some level of traffic differentiation

RED - This is the well-known queue management technique for distributing fairness amongst rate-adaptive (“TCP-friendly”) flows.

Generalized RED - A Linux special version of RED

Differentiated Service Marker - The queueing discipline needed to do marking rather than dropping.

Stochastic Fairness Queue SFQ is an approximate technique for fairness.

Token Bucket Flow - A Token Bucket policing queue.

Traffic Equalizer - Traffic Equalization is an EPFL special

CBQ - CBQ and CSZ are techniques for combining other queueing disciplines into a structure. This is not necessarily hierarchical, but usually).

CSZ David Clark, Scott Shenker and Lixia Zhang carried out some of the earliest Integrated Services research.

ATM VC Manager In the closing years of the 2nd millennium, Internet core networks were often built around level 2, ATM switched infrastructure. This is a way to manage the IP QoS overlay ontop of the ATM layer - other techniques (e.g. MPLS) are evolving.

9.11 Scheduler Framework

The API to the entire scheduling system looks like this:

9.11.1 Scheduling API

[Figure 898 about here.]

[Figure 899 about here.]

[Figure 900 about here.]

[Figure 901 about here.]

[Figure 902 about here.]

[Figure 903 about here.]

[Figure 904 about here.]

[Figure 905 about here.]

[Figure 906 about here.]

[Figure 907 about here.]

[Figure 908 about here.]

[Figure 909 about here.]

[Figure 910 about here.]

[Figure 911 about here.]

[Figure 912 about here.]

[Figure 913 about here.]

[Figure 914 about here.]

[Figure 915 about here.]

Changing scheduler discipline parameters

[Figure 916 about here.]

[Figure 917 about here.]

[Figure 918 about here.]

RTNetlink to Scheduler Interface

[Figure 919 about here.]

[Figure 920 about here.]

[Figure 921 about here.]

[Figure 922 about here.]

[Figure 923 about here.]

[Figure 924 about here.]

[Figure 925 about here.]

[Figure 926 about here.]

[Figure 927 about here.]

Actually scheduling packets.

[Figure 928 about here.]

[Figure 929 about here.]

[Figure 930 about here.]

Starting up Packet Scheduling

[Figure 931 about here.]

[Figure 932 about here.]

[Figure 933 about here.]

9.11.2 Scheduler Framework

[Figure 934 about here.]

[Figure 935 about here.]

[Figure 936 about here.]

[Figure 937 about here.]

[Figure 938 about here.]

[Figure 939 about here.]

[Figure 940 about here.]

[Figure 941 about here.]

[Figure 942 about here.]

[Figure 943 about here.]

[Figure 944 about here.]

Generic Scheduler

Here is a generic example of a scheduler which acts to try to Help clarify the API.

[Figure 945 about here.]

[Figure 946 about here.]

[Figure 947 about here.]

[Figure 948 about here.]

[Figure 949 about here.]

[Figure 950 about here.]

[Figure 951 about here.]

[Figure 952 about here.]

[Figure 953 about here.]

[Figure 954 about here.]

[Figure 955 about here.]

9.12 Class Based Queueing

CBQ is a framework, or if you prefer, a meta-discipline within which sub-disciplines are placed. Initially, a filter will run to classifier a packet, which will then be handed to the CBQ discipline - within this discipline, the same filter/classify/queue steps are recursively re-run to decide which sub-class the packet belongs to. This can go on as long as you like. The CSZ framework discussed later is similar.

[Figure 956 about here.]

[Figure 957 about here.]

[Figure 958 about here.]

[Figure 959 about here.]

[Figure 960 about here.]

[Figure 961 about here.]

[Figure 962 about here.]

[Figure 963 about here.]

The Actual Classifier

[Figure 964 about here.]

[Figure 965 about here.]

[Figure 966 about here.]

Using a Class

[Figure 967 about here.]

[Figure 968 about here.]

Marking a Class

[Figure 969 about here.]

CBQ Queueing Actions

[Figure 970 about here.]

[Figure 971 about here.]

The overlimit function.

[Figure 972 about here.]

[Figure 973 about here.]

[Figure 974 about here.]

[Figure 975 about here.]

[Figure 976 about here.]

[Figure 977 about here.]

[Figure 978 about here.]

[Figure 979 about here.]

We need to propagate the occupancty of each child up to the top of the CBQ tree to make sure we get the underlimit/overlimit and borrow actions correct:

[Figure 980 about here.]

Then we propagate this to each level.

[Figure 981 about here.]

[Figure 982 about here.]

[Figure 983 about here.]

If a level is underlimit, we may need to lend capacity.

[Figure 984 about here.]

If we are dequeuing a packet, which is first?

[Figure 985 about here.]

[Figure 986 about here.]

[Figure 987 about here.]

[Figure 988 about here.]

[Figure 989 about here.]

[Figure 990 about here.]

[Figure 991 about here.]

[Figure 992 about here.]

[Figure 993 about here.]

Initialising the CBQ machinery.

[Figure 994 about here.]

[Figure 995 about here.]

[Figure 996 about here.]

Now manage the whole shebang.

[Figure 997 about here.]

[Figure 998 about here.]

[Figure 999 about here.]

Here's the core part of CBQ

[Figure 1000 about here.]

[Figure 1001 about here.]

[Figure 1002 about here.]

[Figure 1003 about here.]

[Figure 1004 about here.]

[Figure 1005 about here.]

Clark/Shenker/Zhang Scheduling

Clark, Shenker and Zhang[19] suggested a different structure for a hierarchical classification architecture.

[Figure 1006 about here.]

[Figure 1007 about here.]

[Figure 1008 about here.]

[Figure 1009 about here.]

[Figure 1010 about here.]

[Figure 1011 about here.]

[Figure 1012 about here.]

The Code and Data Structures are similar to CBQ so we don't cover those here.

9.13 Queue Management

As discussed earlier, queue management is really about what to do when the queue occupancy is becoming a source of problems.

9.13.1 Best Effort (a.k.a FIFO) Scheduler

Here we see the classic set of standard queue management functions for the simplest (default) case, that of FIFO queueing.

[Figure 1013 about here.]

[Figure 1014 about here.]

[Figure 1015 about here.]

[Figure 1016 about here.]

[Figure 1017 about here.]

[Figure 1018 about here.]

Then we have to also have reshaping versions of all of those functions!

[Figure 1019 about here.]

[Figure 1020 about here.]

[Figure 1021 about here.]

[Figure 1022 about here.]

Estimate rates

[Figure 1023 about here.]

[Figure 1024 about here.]

[Figure 1025 about here.]

[Figure 1026 about here.]

[Figure 1027 about here.]

Here we see what the *policer* functions look like:

[Figure 1028 about here.]

[Figure 1029 about here.]

[Figure 1030 about here.]

[Figure 1031 about here.]

[Figure 1032 about here.]

[Figure 1033 about here.]

[Figure 1034 about here.]

9.13.2 Ingress Queueing

Ingress queueing is useful when we want to give a specific forwarding behaviour based on what the input process looks like for packet processing.

[Figure 1035 about here.]

[Figure 1036 about here.]

[Figure 1037 about here.]

[Figure 1038 about here.]

[Figure 1039 about here.]

[Figure 1040 about here.]

[Figure 1041 about here.]

[Figure 1042 about here.]

9.13.3 Differentiated Services Marker

Differentiated services is based on packet marking. The DSCode points have to be configured, according to various filters or classifiers, then acted upon, based on the source behaviour.

[Figure 1043 about here.]

[Figure 1044 about here.]

[Figure 1045 about here.]

[Figure 1046 about here.]

[Figure 1047 about here.]

[Figure 1048 about here.]

[Figure 1049 about here.]

[Figure 1050 about here.]

[Figure 1051 about here.]

[Figure 1052 about here.]

[Figure 1053 about here.]

[Figure 1054 about here.]

[Figure 1055 about here.]

[Figure 1056 about here.]

[Figure 1057 about here.]

[Figure 1058 about here.]

[Figure 1059 about here.]

9.13.4 RED and Friends

Random early detection is one of several scalable queue management techniques.

[Figure 1060 about here.]

[Figure 1061 about here.]

[Figure 1062 about here.]

[Figure 1063 about here.]

[Figure 1064 about here.]

9.13.5 Explicit Congestion Notification

Explicit Congestion Notification (ECN) is an alternative to packet dropping which TCP can act on (as discussed in chapter six).

[Figure 1065 about here.]

RED Enqueue and Dequeue functions

Here's an interesting comment too!

[Figure 1066 about here.]

[Figure 1067 about here.]

[Figure 1068 about here.]

Here is how we map the randomness into the queue position.

[Figure 1069 about here.]

[Figure 1070 about here.]

[Figure 1071 about here.]

[Figure 1072 about here.]

[Figure 1073 about here.]

[Figure 1074 about here.]

9.13.6 GRED

GRED is a generalized version of RED that includes the idea of a 1 bit penalty box (Red In/Out) for TCPs that do not “conform” to their contract. It is structurally very similar to RED so we don’t include a detailed discussion of the code here.

9.13.7 Token Bucket Queue

The token bucket was described earlier in the chapter.

[Figure 1075 about here.]

[Figure 1076 about here.]

[Figure 1077 about here.]

[Figure 1078 about here.]

[Figure 1079 about here.]

[Figure 1080 about here.]

[Figure 1081 about here.]

[Figure 1082 about here.]

[Figure 1083 about here.]

[Figure 1084 about here.]

[Figure 1085 about here.]

[Figure 1086 about here.]

9.13.8 Ingress Queueing

[Figure 1087 about here.]

[Figure 1088 about here.]

[Figure 1089 about here.]

9.13.9 Priority Queueing

Priority queueing is just about the simplest way to implement traffic differentiation.

[Figure 1090 about here.]

[Figure 1091 about here.]

[Figure 1092 about here.]

[Figure 1093 about here.]

[Figure 1094 about here.]

Change some parameters.

[Figure 1095 about here.]

9.13.10 Stochastic Fairness Queueing

[Figure 1096 about here.]

[Figure 1097 about here.]

9.13.11 Priority Queueing

[Figure 1098 about here.]

9.13.12 Stochastic Fairness Queueing

[Figure 1099 about here.]

[Figure 1100 about here.]

[Figure 1101 about here.]

[Figure 1102 about here.]

[Figure 1103 about here.]

[Figure 1104 about here.]

[Figure 1105 about here.]

[Figure 1106 about here.]

[Figure 1107 about here.]

[Figure 1108 about here.]

9.13.13 Link Equalization

[Figure 1109 about here.]

[Figure 1110 about here.]

[Figure 1111 about here.]

[Figure 1112 about here.]

[Figure 1113 about here.]

[Figure 1114 about here.]

[Figure 1115 about here.]

[Figure 1116 about here.]

[Figure 1117 about here.]

[Figure 1118 about here.]

[Figure 1119 about here.]

[Figure 1120 about here.]

[Figure 1121 about here.]

[Figure 1122 about here.]

9.13.14 Profiling

[Figure 1123 about here.]

[Figure 1124 about here.]

[Figure 1125 about here.]

[Figure 1126 about here.]

[Figure 1127 about here.]

Almesberger/EPFL ATM VC Scheduling

ATM is a pretty special case, where a level two network does the real work, but we can interface to this - Almesberger[1] did some nice work here providing a clean integrated interface.

[Figure 1128 about here.]

[Figure 1129 about here.]

[Figure 1130 about here.]

[Figure 1131 about here.]

[Figure 1132 about here.]

[Figure 1133 about here.]

[Figure 1134 about here.]

[Figure 1135 about here.]

Now we have to worry about virtual circuit management too!

[Figure 1136 about here.]

In a way, the level two inherits changes from the level three. If we were really clever, we might think of ATM (and any MPLS type QoS aware network) as being a leaf level below the IP leaf level!

[Figure 1137 about here.]

[Figure 1138 about here.]

[Figure 1139 about here.]

[Figure 1140 about here.]

Here we have more general level of management

[Figure 1141 about here.]

[Figure 1142 about here.]

[Figure 1143 about here.]

[Figure 1144 about here.]

[Figure 1145 about here.]

[Figure 1146 about here.]

[Figure 1147 about here.]

[Figure 1148 about here.]

[Figure 1149 about here.]

[Figure 1150 about here.]

9.14 Managing Forwarding Treatment Information - the TC Command

Linux now has a sophisticated (and complex) traffic control facilities - these could be managed through some distributed resource reservation or traffic engineering algorithm, or manually setup at each site. The TC command is an application that lets you configure the facilities on a specific Linux. system

9.14.1 Traffic Control - Using The Tool

```
*****
#!/bin/sh
TC=/src/iproute2/tc/tc
DEV="dev eth0"
efrate="2Mbit"
MTU="1.5kB"

#$TC qdisc add $DEV handle 1:0 root dsmark indices 64 set_tc_index
#$TC filter add $DEV parent 1:0 protocol ip prio 1 tcindex mask 0xfc shift 2

$TC qdisc add $DEV handle 1:0 root dsmark indices 64
$TC qdisc add $DEV parent 1:0 handle 2:0 prio

#
# EF class: Maximum about one MTU sized packet allowed on the queue
#
$TC filter add $DEV parent 1:0 protocol ip prio 1 handle 0x22 tcindex \
    classid 2:1
$TC qdisc add $DEV parent 2:1 tbf rate $efrate burst $MTU limit 2.5kB
```

```

$TC filter add $DEV parent 1:0 protocol ip prio 2 handle 0x2f tcindex mask
0xff\
    classid 2:2
$TC qdisc add $DEV parent 2:2 handle 3:0 gred setup DPs 2 default 2 prio

#
# AF class
#
$TC filter add $DEV parent 1:0 protocol ip prio 3 handle 0x24 tcindex \
    classid 3:1
$TC qdisc add $DEV handle 3: gred limit 90KB min 55KB max 65KB \
    burst 90 avpkt 1000 bandwidth 10Mbit DP 1 probability 0.05 prio 2
#
# BE class
#
$TC filter add $DEV parent 1:0 protocol ip prio 3 handle 0x26 tcindex \
    classid 3:2
$TC qdisc add $DEV handle 3: gred limit 90KB min 35KB max 50KB burst 90 \
    avpkt 1000 bandwidth 10Mbit DP 2 probability 0.1 prio 3

```

9.14.2 Traffic Control - How the Tool is implemented

[Figure 1151 about here.]

9.14.3 TC Command Usage

The TC command usage is fairly complex, since there are three inter-twined types of object one can set up. As described earlier, we have a filter type, a class, and then the actual queuing discipline, with its parameters. Normally, one uses `tc` to select a queuing discipline for a device, then attaches a filter to the device to direct packets to the appropriate queue. In more complex cases, this recurses via CBQ or CSZ and the use of class identifiers, as well as filters.

```

tc [ OPTIONS ] OBJECT { COMMAND | help }
where OBJECT := \{ qdisc | class | filter \}
    OPTIONS := \{ -s[tatistics] | -d[etails] | -r[aw] \} }

```

TC qdisc command

As discussed in the rest of this chapter, the queuing discipline is currently one of { atm | [p|b]fifo | tbf | prio | hfsc | hpfq | sfq | cbq | csz | red | gred | dsmark | ingress | etc. }

```

tc qdisc [ add | del | replace | change | get ] dev STRING
    [ handle QHANDLE ] [ root | ingress | parent CLASSID ]
    [ estimator INTERVAL TIME_CONSTANT ]

```

```
[ [ QDISC_KIND ] [ help | OPTIONS ] ] }
```

We can be adding a base queue discipline, optionally with a handle, or we can be adding one as a child in a class hierarchy (usually previously defined via `cbq` or `csz`), or we can be adding an estimator. One exceptional case is ingress queueing (see later).

The *estimator* takes an interval and a time constant over which the estimate decays. This is used to monitor queue occupancy, without running too many cycles up doing so!

When we add or modify a given queueing discipline, there are configuration parameters we also must setup. The options each queueing discipline can be configured with are (roughly) as follows:

atm IP over ATM can setup ATM parameters as part of a class hierarchy. These include the type of ATM VC in use, and the VC class (in the ATM sense, in other words, UBR, VBR, etc); The ATM AAL level parameters such as SDU (packet size, usually 9280 bytes), and any ATM level addressing information (e.g. whether E164 or ISO addresses are used to identify the “link” layer next hop address, or in other words the ATM VC end point.

```
atm ( pvc <addr> | svc <addr> [ sap <sap> ] ) | qos <qos> [
sndbuf <bytes> ] | hdr <hex> | excess ( <class id> | clp )
clip
```

fifo FIFO takes one parameter, which is the maximum queue length.

```
[p|b]fifo [limit <n>]
```

tbf The token bucket filter takes a queue length, `limit`, and a latency (in microseconds), or a burst specification in terms of a buffer and a burst length, as well as mean and peak rate parameters. Also, it can take an MTU parameter and a maximum path unit packet size parameter.

```
tbf limit <bytes> burst <bytes>[/<bytes>] rate <kbps> [ mtu
<bytes>[/<bytes>] ]
```

prio Priority queues are configured with a number of bands, then a map of the identifiers (so that they can be used later in a filter).

```
prio bands <n> priomap <P1> <P2> ...
```

hfsc The Hierarchical Fair Servive curve is not part of the current release.

hpfq Hierarchical Packet Fair Queueing is not in the current release at the time of writing.

sfq Stochastic Fairness queueing can be setup with a perturbation and a quantum, which represent the input to the statistical nature of the nature of this approximate fair queueing technique.

```
sfq [ perturb <seconds> ] [ quantum <bytes> ]
```

```

cbq cbq bandwidth <bps> rate <bps> maxburst <npkts> [ avpkt <bytes> ]
      [ minburst <npkts> ] [ bounded ] [ isolated ]
      [ allot <bytes> ] [ mpu <bytes> ] [ weight <rate> ]
      [ prio <n> ] [ cell <bytes> ] [ ewma <log> ]
      [ estimator <interval> <time constant> ]
      [ split <class id> ] [ defmap <mask>/<change> ]

```

or

```

cbq bandwidth <bps> avpkt <bytes> [ mpu <bytes> ]
      [ cell <bytes> ] [ ewma <log> ]

```

csz is not currently implemented.

red RED sets up the parameters for random early detection, basically in line with the research that describes red - for example the minimum and maximum threshold and average number of packets in the queue that trigger and set the drop (or marking) probability (markign if ECN is set).

```

red limit <bytes> min <bytes> max <bytes> avpkt <bytes>
      burst <n packets>      red probability <probability> bandwidth
      <kbps> [ ecn ]

```

gred GRED includes a range of drop probabilities suitable, for example for the differnetiated services AF class.

```

gred DP drop-probability limit <bytes> min <bytes> max
      <bytes> gred avpkt <bytes> burst <n pkts> probability <prob>
      bandwidth <kbps> gred [prio value] gred setup DPs <no. of
      DPs> default <default DP> [grio]

```

dsmark Differnetiated Services Marking, involves setting up a set of indices to a set of mask/value pairs.

```

dsmark indices <indices> [ default index <default index> ] [
      set.tc_index ]

```

ingress takes an IP address as a parameter.

`tc qdisc list` will show the current set of queueing disciplines instantiated.

TC class command

```

tc class [ add | del | change | get ] dev <string>
      [ classid <class id> ] [ root | parent <class id> ]
      [ [ <qdisc kind> ] [ help | <options> ] ] }

```

Essentially, class identifiers are attached to queueing disciplines, and are either a root, or are attached to parent classes. A class can also have an estimator attached.

TC filter command

```
tc filter [ add | del | change | get ] dev <string>
    [ pref <priority> ] [ protocol <protocol> ]
    [ estimator <interval> <time constant> ]
    [ root | classid <class id> ] [ handle <filter id> ]
    [ [ <filter type> ] [ help | <options> ] ]

tc filter show [ dev <string> ] [ root | parent <class id> ]
```

Where:

```
FILTER_TYPE := { rsvp | u32 | fw | route | etc. }
FILTERID := ... format depends on classifier, see there
OPTIONS := ... try tc filter add <desired FILTER_KIND> help
```

Filters are either setup as per RSVP flow specification, or on specific unsigned 32 bit identifiers, or according to a route classifier, or finally as a firewall filter. Filters have a handle, and can have a parent, a priority, a preference and possibly a protocol, as well as possibly also adding an estimator.

9.15 BSD

Recent releases of BSD Unix have added a class based queueing which was developed by Kenjiro Cho in the Wide Project in Japan. This system predates the Linux system, and is quite similar, although less general in its current form. As in the Linux code, it has a classifier, a queueing discipline, and a policing system (called a *regulator*). The pipeline for these components is the same as in the Linux CBQ structure, but less general in that the order is not re-programmable.

Readers familiar with the BSD network device output interface will see immediately how it fits in:

```
ip_output()
calls
if_output()
which
classifies
enqueues

if_start()
dequeues a packet and
regulates it
```

Other functions manage the queue parameters such as maximum length. There are a variety of possible regulators, but the default one is a token bucket one, which has a rate and bucket size as in the Linux case..

9.16 Summary

In this chapter we have looked at the various forwarding treatments that Linux offers as a routing platform, and the tools to manage the packet forwarding scheduler, queue manager and associated drop behaviours.

In the next chapter, we look at Inter-Process Communication.

Chapter 10

Security

10.1 Roadmap

In this chapter we will consider some of the security facilities offered by the Linux kernel. The main security features are implemented as part of the netfilter work. This enables firewalls and masquerading routers to be built. Towards the end of the chapter there will be a discussion of IPsec and SSL.

10.2 Security Scenarios

10.2.1 Firewalls

A firewall is a mechanism placed between the internet and the operating system and applications of the host you wish to protect. Traditional firewalls were placed on or near the main gateway router, but there is now a trend of personal firewalls which run on the system being protected. The purpose of a firewall is to block traffic which may be malicious in nature. Such traffic could be attacks that exploit holes in the operating system or attempt denial of service attacks on a host. Using a firewall is akin to locking the front door of your home, rather than fitting individual locks on each of the insecure doors inside. One main door can protect a large number of smaller doors. In this way a site system administrator can protect their users without necessarily involving the users in the protection.

10.2.2 NAT—Network Address Translation

NAT is a scheme for translating parts of an IP datagram—notably the source and destination addresses and ports. The purpose of NAT is to provide other services such as Port Forwarding, Transparent Proxying and Masquerading. NAT works by monitoring certain parts of packets and if they match a particular pattern then modifying the packet accordingly before forwarding it.

Masquerading This is a scheme where a LAN containing many hosts can be connected to the Internet even if the ISP has provided only a single IP address for a single host. One machine is configured as masquerading router and this translates packets in and out of the network acting on behalf of

(masquerading as) individual hosts on the LAN. Consider the network in figure 1149. Here a masquerading router “*M*” is routing packets from 3 hosts on a private network “*A*”, “*B*” and “*C*”. Consider the case when HOST *A* wishes to download a web page from some site on the internet, called say *W* with an address of 3.4.5.6. *A*’s packets will be initially sent towards *M*, with the IP source address of *A* and an ephemeral port *a*, and with destination address of *D*, port 80. *M* will rewrite the packet inserting its own IP address as the source and its own ephemeral source port (*m*). The packet will travel across the Internet and eventually a response packet will be found. This will be addressed towards *M* : *m*, but, the NAT code in *M* will match this packet and rewrite it addressing it toward *A* : *a*. See figure 1150, and note that *A*’s address and port never make it outside of the masqueraded LAN.

[Figure 1152 about here.]

[Figure 1153 about here.]

Port Forwarding Consider the situation where a corporate web server is behind a masquerading, fire-walling router. Any we request from outside the organisation needs to be directed towards this machine’s web server port. However, as this machine is within a masqueraded network, then its IP address is not globally valid. The solution is to publish the firewall’s address *F* as the web service and forward requests arriving at the firewall to the actual machine, using NAT. See figure 1151. Here an incoming web request from a machine out on the Internet *R* is forwarded to the web-server *W*. *W* generates its reply and sends the result back to *M* which translates it for its return path to *R*.

[Figure 1154 about here.]

Transparent Proxying A transparent proxying server allows for a service to be proxied on an intermediate host, with the knowledge, and therefore reconfiguration of the client machine. By proxying we mean an additional service being performed on behalf of the client. The usual example of proxying is a caching web server. Here a server monitors requests and stores (caches) the responses. If a second or further request for the same web page arrives this can be responded to with the saved data immediately without waiting for a response from the origin host. Another benefit of proxy-caches is that they can run on a firewalling machine forwarding requests from a protected domain within an organisation. See 1153. Here host *A* is using *M* is a proxying web cache. Therefore, any web page requests are sent to *M* using port 3128 in this example. *M* then makes a check of its stored objects and if the requested object is not available then it itself request the page from the actual site *W* : 80 (this information will be in the HTTP GET message). The site responds and *M* relays the response to *A*. Host *A* must be configured to allow this mode of operation by informing the browser to request pages from the web cache rather than the origin web site. A transparent web cache employs port forwarding to

prevent this need for reconfiguration and thus save system administrators time and improve ISP profits (or reduce losses). See figure 1153. The web cache machine takes all packets with 80 as a destination port (i.e. most web traffic) forwards it to some application level caching software. If there is a match in the cache then response can be formulated and the corresponding IP header information placed in the response packet. I.e. place the destination address/port of the request in the response source address/port. If the page is not cached then the cache can request it using the address of the original requesting machine. To both the web server and the browsing host the operation of the cache goes unnoticed except possibly the improvement or reduction in performance.

[Figure 1155 about here.]

[Figure 1156 about here.]

It is possible for masquerading, firewalling and transparent proxying functionality to be combined on a single machine. This is left as an exercise for the reader.

Each of the above functionalities is available through the netfilter system in linux which is described now

10.3 Netfilter

10.3.1 Introduction

The netfilter code allows hooks to be placed in the packet flow. If we consider a linux box acting as a router then a hook can be placed at the following places (see 1154:

1. after input before the routing stage `NF_IP_PRE_ROUTING`
2. after input if the packet is for the local machine `NF_IP_LOCAL_IN`
3. if the packet is to be forwarded `NF_IP_FORWARD`
4. before the packet is output `NF_IP_POST_ROUTING`
5. if the packet is created locally `NF_IP_LOCAL_OUT`

[Figure 1157 about here.]

Netfilter allows the following actions to be performed at a filtering hook.

- **ACCEPT**—Indicates that the packet should be allowed to continue, i.e. the next hook should be called.
- **DROP**—The packet should be dropped by the kernel.
- **STEEL**—The packet will be handled by another process.
- **QUEUE**—The packets should be queued for processing, usually this is to pass a packet to user space.
- **REPEAT**—repeat the call to this hook.

10.3.2 IP Tables

IPtables is a mechanism where a list of rules can be applied to selected packets. Included within the linux kernel are a set of default tables which support:

- filtering
- mangling
- connection tracking
- nat

taking filtering as example, see `net/ipv4/netfilter/iptable_filter.c`. Here a structure is build which is hooked into just the `NF_IP_LOCAL_IN`, `NF_IP_FORWARD` and `NF_IP_LOCAL_OUT`

hooks. rules are then added to the table under control of user space programs which indicate which packets match and should be dropped.

10.3.3 Summary

IPtables is a complex and extendable security system involving both user and kernel space code. This section has concentrated on the deep kernel code. For more information on the user space code and setting up IPtables secured systems refer to *****IPtables refs*****

10.4 IPsec

IPsec is an IP datagram layer security system that introduces extra headers for authentication (Authentication Header—AH) and encryption (Encapsulation Security Protocol—ESP) of individual packets. IPsec is described in RFCs2401–2412. An implementation of IPsec is not currently part of the Linux kernel.

10.5 SSL

The secure sockets layer uses a combination of public key and symmetric key cryptography to provide a secure transport layer between hosts. Its most common implementation is within web browsers for access to secure web sites and the SSH program to allow secure communication and logins between hosts. Currently SSH is entirely implemented by user space programs.

10.6 Summary

This chapter has taken a look at netfilter code in the linux kernel, and how it can be used to provide security for sites and users. A brief mention has been made of IPsec and SSL, but these are not part of the linux kernel at the time of writing, although the FreeSWAN project does provide an IPsec implementation.

Afterword

When I started writing this, I compared it to the task faced by the translators who prepared the famous King James' 1st, Bible - they were no the first to attempt a translation from Latin, nor the last. Nor was a large part of the book originally written in Latin (but rather in Aramaic and Greek in the old and new testaments respectively). Similarly, this book is in English, but is about the Linux communications stack, which is written in C, and implements a system designed by the Internet pioneers in RFCs, written in the semi-formal language used by protocol specifiers. Furthermore, there already is a "King James' translation" of one C implementation, so I guess I have to think of this as a new english version!

As is the case, much has been lost in the translation, and while I (like Spike Milligan and mathematics) speak C like a native, and have written RFCs, that only means that I must take the blame even more readily for any such errors.

Thats all for now, folks!!

Glossary

- AAL - ATM (see ATM) Adaptation Layer
AALs are framing and protocol layers between the cell transmission layer and layer three (IP).
- ACK - common abbreviation for Acknowledgement
- Adspec
An Adspec is an object in a Path message that carries a package of OPWA advertising information. See OPWA.
- Admission Control - Call Admission, CAC, Decision Traffic (packets) are allowed to flow because of some decision based on past, current and expected conditions.
 1. The decision can be based on parameters in the packet (simplest is the destination address- so called destination based routing), ToS, Precedence, current traffic load (e.g. buffer overload calls for "call blocking" or packet dropping. Which packet gets dropped from an overloaded queue is an interesting question (see RED). Keeping queues short (under load) is a good thing.
 2. The decision can also be based on out-of-band signaled parameters about the traffic flow (see QoS and RSVP), and on downstream conditions (is QoS available all the way(s) to the destination?). Such information on downstream conditions is typically disseminated by QoS routing protocols.
- ADU - Application Data Unit
The unit of transmission or reception from an application protocol such as FTP or HTTP.
- AF - Assured Forwarding
One of the Differentiated Services classes or behaviours. Defined in terms of loss preference.
- Aggregation (see Scale) Traffic can be aggregated - the Internet does not muxing currently - it also aggregates state about traffic flows for destination based routing by aggregating addresses hierarchically (see CIDR). Other forms of aggregation are possible -

1. routing considerations (link state advertisements) can be constrained to areas, and then inter-area routing considered separately.
 2. traffic flow conditions downstream can be aggregated by routing area.
- AIMD - “Additive Increase, Multiplicative Decrease”
The control law for a TCP like protocol when seeing no or some congestion indication.
 - Alternate Path Routing
A routing technique where multiple paths, rather than just the shortest (typically opportunistic) path, between a source and a destination are utilized to route traffic. One of the objectives of alternate path routing might be to distribute load among multiple paths in the network (or, conversely to reduce dependence on single mode/path failure).
 - API - Application Programming Interface
The set of functions and parameters needed by a higher level protocol to use a lower level protocol.
 - Appletalk - a Proprietary family of protocols defined by said manufacturers systems.
 - ARP - Address Resolution Protocol
Used to map dynamically between layer three (IP) addresses and layer two (MAC) addresses.
ARPA, DARPA - Defence Advanced Research Project Agency -US DoD research funding agency that underwrote a lot of the invention and discovery of the Internet.
 - ARPANET
The research network funded by the Advanced Research Project Agency (ARPA) on which the Internet protocols were first developed.
 - AS - Autonomous System
An AS runs a single routing protocol (e.g. RIP or OSPF) and interfaces to other ASs via an inter-domain routing protocol, today, BGP.
 - ASM - assembly
mnemonic version (human readable) of machine program language
 - Assured Forwarding
PHB for Differentiated Service that allows multiple forwarding classes with (orthogonally) multiple drop priorities. Aimed at supported mixes of RTP/UDP and TCP traffic. Might use RED
 - ATM - Asynchronous Transfer Mode - c.f. B-ISDN - a “link layer” technology which may be used to optimise packet transfer times by cut-through packet forwarding (aka IP Switching) - has its own entire architecture (c.f. B-ISDN) for traffic management.

1. VCI/VPI -
Virtual Circuit Identifier - Path Identifier - used as a 2-level tag for a flow.
 2. UBR, VBR(rt—nrt), ABR, CBR
Un-specified, Variable, Available and Constant Bit Rate service classes
 3. MCR, PCR, SCR
Mean, Peak, Sustainable Cell Rate
 4. CDV, BT
Cell Delay Variation, Burst Tolerance
 5. RM cells
Resource Management
 6. PNNI, IPNNI
Private Network Network Interface - signaling and QoS routing from the ATM Forum
- ATMPVC Permanent Virtual Circuit - a configured resource on an ATM level two network.
 - Autonomous System (AS)
A routing domain which has a common intradomain routing protocol and administrative authority.
 - Class - aka. profile, template, c.f. generalized Port Specification
A traffic class is a contract *type* - a contract is a contract type, plus a set of values for the parameters for that type - the contract type specifies the parameters of the admission control...for example:
 1. Best Effort
has no parameters (beyond source+destination address and ToS and precedence!)
 2. Controlled Load
Specifies a contract which looks to the source(sink) like a lightly loaded network with the specified mean rate... borrowing of capacity between a flow that has a burst at a higher rate and ones that are currently lower rate is possible...the control of this over the short time frame is a factor in selecting a packet scheduling algorithm. (c.f. schedule)
 3. Guaranteed Service
This is very like the ATM CBR contract...but it is also like the VBR rt service - it specifies delay bounds as well as throughput parameters - these are actually implemented end to end and hop by hop, so are somewhat different from the CDV parameter which is a traffic behaviour specification rather than a contract request!
 4. Slack
A term in RSVP for how to divvy up variation along the hops of a path

5. Controlled Delay This and the next two are three proposed service contracts
6. Committed Rate
7. Predictive

- AX25

The amateur packet radio network's version of the old X.25 protocol, used to mask errors on the packet radio network, so that it can carry IP packets more effectively.

- BGP - Border Gateway Protocol

The inter-domain routing protocol in use worldwide today.

- Blockade state

Blockade state helps to solve a "killer reservation" problem.

- BPF - Berkeley Packet Filter

Programmable mechanism for general packet classification.

- BSD - Berkeley Standard Definition

One of the main family trees of Unix alongside System V and Linux, and others.

- CAM - Content Addressable Memory

CAMs are useful for various "associative" type lookup operations where we want to find the index position of something via its value rather than the other way around.

- CBQ - Class Based Queueing

This is an architecture for integrating a number of service levels implemented by a variety of queueing disciplines, and associating them with collections or aggregates of users (e.g. by agency or address prefix).

- CIDR - Classless Inter-Domain Routing

Address Aggregation - currently used to compact destination based forwarding tables - could be used for sources or even for logical (e.g. multicast class D or mobile) addresses, but isn't yet.

- Classifier - an implementation of a rule to decide the provenance of a packet, and assign it to a class.

- Congestion

A state of the network and its offered load where traffic is partially or largely not being carried usefully - e.g. traffic flows across a link to a bottleneck only to be dropped. Also, in multiservice networks with reservations, can refer to high call blocking probability - actually, this is simply scarcity of resource.

- Crankback
A technique where a flow setup is recursively backtracked along the partial flow path up to the first node that can determine an alternative path to the destination.
- CSMA - Collision Sense Multiple Access
Building block for shared media network mediation.
- CSZ - Clark, Shenker and Zhang
Alternative model to CBQ (see above).
- cwnd - congestion window
This is an important parameter in the TCP congestion avoidance algorithm discussed in chapter seven.
- DECNET - the Digital Equipment Corporation's old network architecture
- Destination Routing
Current standard way that route is chosen for each and every packet in the Internet - the destination address is used to look up the "shortest" (by some metric, whether hop count, highest throughput, lowest delay or what) path out of the graph that is the network mesh map, which may have multiple potential different paths. To select more than one path, one could use external state (e.g. random number, traffic load from other sources) but this might lead to high degrees of packets arriving out of order at a destination - to use more than a single path, one needs to distinguish flows based on other information - e.g. source, or source + ToS (+Precedence(+signaled QoS)) or generalized port information. Alternatively, a fast lookup key associated with this information, whatever it is, could be installed - e.g. by the signaling protocol, or by implicit recognition of the existence of a "flow", by a "tag" or "flow label" installation protocol. such tags might be hop specific.
- DHCP - Dynamic Host Configuration Protocol
Used to autoconfigure a computer when its connected or booted on a new network.
- Differentiated Services
Class of Service rather than Quality of Service. Uses edge devices and Per Hop Behaviours instead of hop-by-hop and end to end services. Typically, subscription based rather than signaled. Allows aggregation by class.
- DMA - Direct Memory Access
Means for an I/O device to send and receive data from memory without involving the CPU.
- DNS - Domain Name System
The service that maps human-useful names for computers and services to IP addresses.

- DRR - Deficit Round Robin
A common and efficient approximation to Weighted Fair Queueing.
- DVMRP - Distance Vector Multicast Routing Protocol.
a deprecated routing protocol for IP multicast
- Daemon - “Always-on” unix user-space process that handles various service requests, and enhances kernel level functionality.
- Datagram - an IP packet.
- E.164 - the addressing used in ATM networks
- ECN - Explicit Congestion Notification.
A means to feedback information to end systems about incipient congestion at a router in the Internet by explicit (binary) marking of IP packets.
- ECONET - - low performance, low price local net technology.
- EGP - Exterior Gateway Protocol
Old term and protocol for inter-domain routing.
- Entry policing
Traffic policing done at the first RSVP- (and policing-) capable router on a data path. aka ingress policing
- Ethernet - a family of Local Area Network technologies in very widespread use.
- Expedited Forwarding
A PHB for diff serve designed to achieve virtual leased line. typically, uses priority queue.
- Explicit sender selection
A (reservation) style attribute; all reserved senders are to be listed explicitly in the reservation message. See also ”wildcard sender selection”.
- FCFS - First Come First Served -
Effectively the same as FIFO-
- FF style
Fixed Filter reservation style in RSVP, which has explicit sender selection and distinct attributes.
- FIFO - First In First Out
A simple queueing discipline.
- FilterSpec
Together with the session information, in RSVP defines the set of data packets to receive the QoS specified in a flowspec. The filterspec is used to set parameters in the packet classifier function.

- Flow

A sequence of packets from a source to a destination (possibly a group) constituting a single activity (e.g. transfer of a file, a web page, an audio/video stream etc etc) that are usefully temporally as well as spatially correlated!
- Flow Label

A shortcut key for looking up a flow rather than the set of typical IP (TCP and UDP) protocol fields that routers seem poor at looking at (though hosts seem to manage somehow:-)
- Flow Identifier

An IPv6 field that might serve as a globally allocated flow id, or might not...
- Flow set-up

The act of determining the path for a flow, and attempting to establish state in routers along the flow path to satisfy its QoS requirement.
- Flow descriptor

The combination of a flowspec and a filterspec.
- Flowspec

Defines the QoS to be provided for a flow. The flowspec is used to set parameters in the packet scheduling function to provide the requested quality of service. A flowspec is carried in a FLOWSPEC object. The flowspec format is opaque to RSVP, and is defined by the Integrated Services Working Group.
- Flow Admission Control (FAC)

A process by which it is determined whether a link or a node has sufficient resources to satisfy the QoS required for a flow. FAC is typically applied by each node in the path of a flow during flow set-up to check local resource availability.
- Gateway - older term for router.
- GPL - Gnu Public License

A license often used to protect open source software.
- GPS - Global Positioning System

Satellite based location system.
- GRE - Generic Routing Encapsulation

generalized recursive tunnelling.
- GUI - Graphical User Interface

X Windows etc.
- HTML - Hyper-Text Markup Language

- HTTP - Hyper-Text Transfer Protocol
- IANA - Internet Assigned Numbers Authority - assigns codepoints in Internet Protocols.
- ICMP - Internet Control Message Protocol - IP layer diagnostic and error reporting protocol
- IETF - Internet Engineering Task Force - Internet standards working groups body.
- IGMP - Internet Group Management Protocol - IP multicast management protocol.
- IGP- Interior Gateway Protocol - generic term for protocols used within a single AS (see above) - e.g. OSPF, ISIS, etc.
- Integrated Services Internet
- Integrated Service over Specific Link Layers

[Table 20 about here.]

- IP flow (or simply "flow")
An IP packet stream from a source to a destination (unicast or multicast) with an associated Quality of Service (QoS) (see below) and higher level demultiplexing information. The associated QoS could be "best-effort".
- IPC - Inter-Process Communication - mechanism such as sockets.
- IPSec - Internet Protocol Security- term for both architecture and mechanisms.
- IPX - Novell network layer three protocol..
- IPv4, IPv6 - main two versions of IP extant.
- IRQ - Interrupt ReQuest - logical channel for interrupt from a device on PC hardware.
- ISP - Internet Service Provider - generic term for organisations and companies that provide carriage for IP.
- ISR - Interrupt Service Routing - software handler for dealing with interrupts
- ITU - International Telecom Union - standards body for telecommunications.
- Generalized Port Specification
RSVP uses this to identify an IPv4 flow. The component of a session definition that provides further transport or application protocol layer demultiplexing beyond DestAddress. See "session".

- Higher-level admission control

A process by which it is determined whether or not a flow set-up should proceed, based on estimates of the overall resource usage by the flow. Higher-level admission control may result in the failure of a flow set-up even when FAC at each node along the flow path indicates resource availability.
- Killer reservation problem

The killer reservation problem describes a case where a receiver attempting and failing to make a large QoS reservation prevents smaller QoS reservations from being established.
- LAN - Local Area Network
- LIH

The LIH (Logical Interface Handle) is used to help deal with non- RSVP clouds.
- LILO - LInux LOader - the first stage disk bootstrap program for Linux.
- Linux - Linux is not Unix, really.
- Local repair

Allows RSVP to rapidly adapt its reservations to changes in routing.
- LRU - least recently used - cache replacement strategy
- Police

When traffic specifies a service contract, it should keep to it (especially if there are tariff (whether per use or per subscription based) considerations. To check this, traffic is typically monitored as it is placed into appropriate queues for appropriate schedules - if a contract is exceeded, packets can be treated in a number of ways:

 1. Move to a "lower" class

This can be tricky given the lower class may have a policy associated with it....e.g. maybe it gets an overall minimum share...
 2. Drop

This is easy, esp. with RED.
 3. Mark as in "breach of contract"

No field currently in IP to do this, unless the packet was in a class because of priority or precedence, in which case the field could be overwritten, but this would invalidate the IP (v4 header) checksum unless restored after the current piece of network...
- Merge policing

Traffic policing that takes place at data merge point of a shared reservation.

- Merging
The process of taking the maximum (or more generally the least upper bound) of the reservations arriving on outgoing interfaces, and forwarding this maximum on the incoming interface.
- MFC - Multicast Forwarding Cache - fast lookup structure for commonly used IP multicast routing entries.
- MMU Memory Management Unit
- MPLS - Multi-Protocol Label Switching - generalized version of ATM.
- MPOA - Multi-protocol Over ATM
- MSDP - Multicast Source Distribution Protocol - part of IP multicast inter-domain routing framework.
- MSS - Maximum Segment Size - largest packet TCP will try to send whatever the size of the User Request. Generally set or discovered to try to avoid or minimize IP fragmentation (see MTU). Allows for TCP header.
- MTU - Maximum Transfer Unit - largest packet IP can send without fragmentation (see MSS).
- Multicast path (or Multicast Tree)
A subtree of the network topology in which all the leaves and zero or more interior nodes are members of the same multicast group. A multicast path may be per-source, in which case the subtree is rooted at the source.
- Nagle - researcher, known in TCP community for algorithm used to reduce IP/TCP packet count for slow intermittent applications.
- NAT - Network Address Translator - Mechanism to map between locally significant IP addresses and globally allocated ones so that a site can move without renumbering its hosts, or can share a small pool of addresses amongst a small set of active hosts drawn from a larger internal pool.
- Netfilter - mechanisms for packet filtering in Linux (see chapter ten)
- Netlink - mechanism for user to kernel communication to manage network layer systems.
- Netstat- commonly used application to get IP and lower level statistics.
- NFS - Network File System - popular unix file sharing protocol
- NHOP
An RSVP object that carries the Next Hop information in RSVP control messages.
- NIC - Network Interface Card (e.g. 3Com 3c509 Ethernet board).
- NIS - Network Information Service - local only service like DNS only more extensible.

- NNTP - Network News Transfer Protocol - for Bboards.
- NSF - National Science Foundation - popular US research funding agency.
- NTFS - NT File System. A file system type supported by Linux
- OPWA

Abbreviation for "One Pass With Advertising". Describes a reservation setup model in which (Path) messages sent downstream gather information that the receiver(s) can use to predict the end-to-end service. The information that is gathered is called an advertisement. See also "Adspec".
- OSPF - Open Shortest Path First - commonly used Intra-domain IP routing protocol.
- OpenSSL - Implementation of Secure Socket Layer system.
- Quality-of-Service (QoS)

A set of service requirements to be met by the network while transporting a flow. Quality of Service or route:

RSVP defines some fields that carry QoS descriptions - the actual meanings of these are as per the int-serv classes/templates. Given a service class, a flow also is covered by:

 1. Tspec The traffic spec - what the flow actually looks like
 2. a leaky bucket spec Rspec The QoS desired.
 3. a leaky bucket spec ADSpec Advertised flow spec used by OPWA
 4. OPWA - One Path With Advertising see below
- Path state

Information kept in routers and hosts about all RSVP senders.
- PathErr

Path Error RSVP control message.
- PathTear

Path Teardown RSVP control message.
- PHB

Per Hop Behaviour - a service description based around behaviour rather than mechanism, and local, rather than path global. See Expedited and Assured Forwarding groups.
- PHOP

An object that carries the Previous Hop information in RSVP control messages.
- PIM - Protocol Independent Multicast - prevalent routing protocol now for IP multicast packets.
- PPP - Point-to-point protocol - widely used link layer for carrying IP on serial lines.

- PVC, PVP, Permanent Virtual Circuit/Path - configured VC or VP on an ATM network.
- Qdisc- queueing discipline - see below.
- QoS-based routing
A routing mechanism under which paths for flows are determined based on some knowledge of resource availability in the network as well as the QoS requirement of flows.
- Queue Management
Queues exist to smooth traffic in a statistically multiplexed network. In a TDM multiplexed network, queues don't hard ever happen.
- RED - Random Early Detection
A technique for controlling best effort queue lengths - might also be used on out-of-contract packets within a class better than best effort.
- Reservation style
Describes a set of attributes for a reservation, including the sharing attributes and sender selection attributes.
- Resv message
Reservation request RSVP control message.
- ResvConf
Reservation Confirmation RSVP control message, confirms successful installation of a reservation at some upstream node.
- ResvErr
Reservation Error control message, indicates that a reservation request has failed or an active reservation has been preempted.
- ResvTear
Reservation Teardown RSVP control message, deletes reservation state.
- RIP - Routing Information Protocol - simple, but effective intra-domain routing protocol commonly used on modest sized site networks.
- Rspec
The component of a flowspec that defines a desired QoS. The Rspec format is opaque to RSVP, and is defined by the Integrated Services Working Group of the IETF.
- Route (ToS, QoS, QoR)
Type of service is a well understood field in the IPv4 header - some systems even implement setting it - some forwarding software (e.g. slip drivers) use this, but fire routers. QoS - refers to a route derived from consideration of network topology, edge costs and traffic conditions. QoR refers to routes derived solely from edge fixed parameters...

- Route pinning
A mechanism to keep a flow path fixed for a duration of time.
- RSVP - Resource Reservation Protocol
The proposed Internet signaling protocol.
- RTP, RTCP - Realtime Transport Protocol and Real-time Transport Control Protocol
- RTNetlink - controlling and managing the routing systems between the kernel and user space.
- RTT - Round Trip Time- one of the critical performance parameters for a TCP flow.
- SCFQ - Self Clocked Fair Queueing
- SE style
Shared Explicit reservation style, which has explicit sender selection and shared attributes.
- Session
An RSVP session defines one simplex unicast or multicast data flow for which reservations are required. A session is identified by the destination address, transport-layer protocol, and an optional (generalized) destination port.
- Shared style
A (reservation) style attribute: all reserved senders share the same reserved resources. See also "distinct style".
- Soft state
Control state in hosts and routers that will expire if not refreshed within a specified amount of time.
- Signal - see RSVP
- Schedule
A neat way to think about scheduling is to take a picture of a resource share over time and picture how it is *decreased* by someone else asking for special circumstances.
 1. FIFO/FCFS
First In First Out - a strange, but widespread service discipline in IP (and multiprotocol) routers
 2. Drop Policy - Drop Tail, Drop Head, Random (c.f RED)
When the queue is full, which packet gets lost?
 3. Priority Queue
When there is more than one packet in a queue, which gets forwarded?
Note priority queueing can starve lower priority packets of ever seeing service

4. Precedence Queueing
Priority Queue based on the precedence bits in the IP header
 5. Custom Queueing
A (proprietary, but commonly used if by other names) scheme to give certain flows a minimum performance through "hardwired" classifications and minimum queue service rates.
 6. Fair Queueing (FQ)
Round robin
 7. Weighted Fair Queueing (WFQ)
Round robin (rounding error is to allow for variable length packets - "weight" in WFQ sometimes used to refer to unfair share - actually, it's the P in PGPS - see GPS and PGPS!).
 8. Virtual Clock
A service scheme for TDM that still stat muxes.
 9. Generalized Processor Sharing (GPS)
The theory of WFQ - fluid flow
 10. Packet Generalized Processor Sharing (PGPS)
The practice of WFQ - packet flow
 11. Earliest Due Date (EDD)
A neat alternative scheduler for delay bounding
- Service
A single packet cannot be said to see a service contract. A flow of packets can see a statistical properties with distributions such as interarrival and departure times, drop probabilities, re-ordering probabilities, erroring probabilities and so on. The difference between distributions before and after service are the result of the scheduling discipline, combined possibly with the other traffic (especially true for best effort and priority queueing). The state mathematics theory is such that we typically cannot derive anything beyond mean and bounds (if they exist) for typical service classes and worst case traffic. The Internet largely survives because of FIFO and drop tail's amazing ability to survive many traffic arrival patterns and under engineering. RED appears to improve this.
 - Scale (see aggregation)
QoS requires state. State has costs in memory and in messages to establish it. The state may simply be a constant increase over the state needed for destination routing - however, usually destinations are heavily aggregated (e.g. at least by network number, but also by CIDR etc). Flows are per user, and may be present in much larger numbers than host ids (although if the phone net is anything to go by this is not true - they typically have j1
 - Shape
traffic may be within contract, but incur extra jitter due to bursts of other traffic when going through a switch. The jury is still out on whether

one should then re-shape the traffic to its mean rate - typically, if a path has $2 * \text{bandwidth} * \text{delay}$ of buffering, then it can sustain any burstiness - however, there may be some synergy to having a minimum (and maximum) amount of burstiness - there is probably some statistical thermodynamic theory that could show this (e.g. diffusion model, or lattice gas occupancy analogy) but it's beyond me right now...

- Source

A host or router that can be identified by a unique unicast IP address.
 - Source Routing

A technique for selecting a path out of many at the source - requires network to deliver choice to the source, and source to be able to indicate choice in packet, or in signaling. Useful for policies (cf. long haul POTS provider selection) but not for scaling.
 - SSH - Secure Shell.
 - State

Incurred from Route, QoS, QoR and signaled information.
 - Tag Switching

A technique for optimizing the lookup for forwarding packets - reduces router memory limits by a constant factor. See ARIS, flow labeling - can be used on per flow by detection (autocorrelation of destination and time) or by route (output port) or by application...
 - SYN - "Synchronize" - a bit set in the TCP header to indicate part of the 3-way handshake to start a connection (by setting initial sequence numbers and other initial state).
 - Tarrif

A tariff structure has a per parameter, per service class price structure which may include time of day, or even other-traffic related factors. The more complex the service model and the tariff structure, the more expensive it is to collect usage based revenue.
 - Tasklet - ultra-lightweight way to schedule families of low level kernel events including software and other deferred interrupts.
 - TBF, Token Bucket/Leaky Bucket
- [Table 21 about here.]
- TSpec

A traffic parameter set that describes a flow. The format of a Tspec is opaque to RSVP and is defined by the Integrated Service
 - UDP - User Datagram Protocol.
 - Unicast destination

A host or router that can be identified by a unique unicast IP address.

- VFS - Virtual File System
- Virtual Circuit/Channel Call
Not an IP concept.....yet.... or ever, if we can help it...
- VJ-CC - Van Jacobson Congestion Control
End to end arguments - a control theoretic model based scheme for limiting traffic going in to the network based on history - given a delayed feedback loop of what has arrived, what can we say about what can be sent? hard (but not impossible) to apply to multicast flows.
Van has perpetrated this algorithm no less than 3 times:
 1. Congestion Control
 2. Adaptive Playout (in Vat)
 3. Class Based Queueing and RED
- VOIP- Voice Over IP - “killer” application for QoS.
- WFQ - Weighted Fair Queueing
- WF style
Wildcard Filter reservation style, which has wildcard sender selection and shared attributes.
- Wildcard sender selection
A (reservation) style attribute: traffic from any sender to a specific session receives the same QoS. See also ”explicit sender selection”.
- WRR - Weighted Round Robin
- UDP encapsulation
A way for hosts that cannot use raw sockets to participate in RSVP by encapsulating the RSVP protocol (raw) packets in ordinary UDP packets. See Section APPENDIX C for more information.

Bibliography

- [1] “Linux Traffic Control - Implementation Overview” Werner Almesberger, Tech Report, EPFL, Switzerland
- [2] “Linux - Advanced Networking Overview”, Sravanan Rhadakkrishnan, Tech Report, University of Kansas, Lawrence, USA
- [3] “Network Buffers and memory Management”, Alan Cox, Kernel Korner, Linux Journal, Vol 1, No. 30, Oct 1996, <http://www.ssc.com/lj/issue30/kk30.html>
- [4] “Linux IP Networking”, Glenn Herrin, from Masters Project Report, Computer Science, University of New Hampshire <http://pubpages.unh.edu/gherrin/project/linux-net.html>
- [5] “Linux Kernel Internals”, Michael Beck et al, Addison Wesley, Harlow, England 1997, ISBN 0 201 33143 8
- [6] “Linux IP Stacks Commentary”, In depth Code Annotation Coriolis Open Press Stephen T Satchell, H.B.J. Clifford, 2000 ISBN 1 57610 470 2
- [7] “Understanding the Linux Kernel” Daniel Bovet and Marco Cesati O’Reilly, Jan 2001. ISBN 0 596 00002 2
- [8] “TCP/IP Illustrated, Vol 1, 2 and 3” W Richard Stevens, Addison Wesley Professional Computing Series, 1992, 1994, 1996 ISBN 0 201 63354-X
- [9] “End-to-End Arguments in System Design” JH Saltzer, DP Reed, DD Clark ACM Transactions on Computer Systems, 2, 4, pp 277-288 November 1984
- [10] “Unix Systems for Modern Architectures” Symmetric Multiprocessing and Caching for Kernel Programmers Curt Schimmel, 1994 Addison Wesley Professional Computing Series ISBN 0 201 63336 8
- [11] “The GNU Coding Standards” Richard Stallman et al, http://www.gnu.org/prep/standards_toc.html
- [12] “Scalable High Speed IP Routing Lookups”, Marcel Waldvogel (ETH), George Varghese (Washington University), Jon Turner (Washington University), and Bernhard Plattner (ETH) Computer Communication Review, a publication of ACM SIGCOMM, volume 27, number 4, October 1997. ISSN # 0146-4833

- [13] "Small Forwarding Tables for Fast Routing Lookups", M. Degermark, A. Brodnik, S. Carlsson and S. Pink, ACM SIGCOMM, Cannes, France, vol. 27, no. 4, pp. 3-15, Oct. 1997.
- [14] "Trends in Wide Area IP Traffic Patterns", Sean McCreary and Kimberly C. Claffy, in Proc. of ITC Specialist Seminar on IP Traffic Measurement, Modeling and Management, (Monterey, California), Sep. 2000.
- [15] OSPF: Anatomy of an Internet Routing Protocol John T. Moy, Addison-Wesley, 1998, 368 pp. ISBN 0-201-63472-4
- [16] "Generalized Processor Sharing", Abhey Parekh, PhD Thesis, MIT Laboratory for Decision Systems, MIT 1993.
- [17] "Modelling TCP Throughput: A Simple Model and its Empirical Validation", J. Padhye, V. Firoiu, D. Towsley, J. Kurose, Proceedings of Sigcomm'98, Vancouver, CA, September 1998.
- [18] "Strawman specification for TCP friendly (reliable) multicast congestion control (TFMCC)", M. Handley, S. Floyd, B. Whetten, IRTF, reliable multicast working group, work in progress, 1999.
- [19] "Supporting real-time applications in an integrated services packet network: architecture and mechanism," David D. Clark, Scott Shenker and Lixia Zhang, in sigcomm, (Baltimore, Maryland), pp. 14-26, Aug. 1992.
- [20] "Link-sharing and Resource Management Models for Packet Networks" Floyd, S., and Jacobson, V.. IEEE/ACM Transactions on Networking, Vol. 3 No. 4, pp. 365-386, August 1995
- [21] "Managing Traffic with ALTQ", Kenjiro Cho, in 1999 USENIX Annual Technical Conference, (Monterey, California, USA), Jun 1999.
- [22] "Implementing Real-Time Packet Forwarding Policies using Streams," Ian Wakeman, Atanu Ghosh, Jon Crowcroft, Van Jacobson and Sally Floyd, in USENIX 1995 Technical Conference, (New Orleans, Louisiana), Jan. 1995.
- [23] "Operating System Concepts" A Silberschatz, J Peterson, 1988, Addison Wesley, ISBN 0-201-18769-4
- [24] "Computer Organisation and Design" The Hardware Software Interface, D A Patterson, J L Hennessy, 1998 Morgan Kauffman Publishers ISBN 1-55860-491-X
- [25] "Computer networking: A Top Don Approach Featuring the Internet" J Kurose, K. Ross, 2001, Addison Wesley, ISBN 0-201-47711-4
- [26] "The Click modular router", Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek ACM Transactions on Computer Systems 18(3), August 2000, pages 263-297.
- [27] "Linux Routers: A primer for Network Administrators", Tony Mancill, 2001 Prentice Hall, ISBN 0-13-086113-8

- [fire] 'Firewalls and Internet Security : Repelling the Wily Hacker", by William R. Cheswick, Steven M. Bellovin (Contributor), Paperback 2nd edition (October 2001) Addison-Wesley Pub Co; ISBN: 020163466X

RFCs Cited in code.

RFCs are the famous “Request for Comment” series of documents about the Internet Protocol Suite. Some are standards and some are purely informational (and some are historical or just mere entertainment!). There are many points in the Linux kernel code (and elsewhere) that implementors refer to the relevant RFC. Here we give the list. See <http://www.ietf.org> for availability, and status information about these documents and the protocols and services they describe.

- 0791** Internet Protocol. J. Postel. Sep-01-1981
- 0793** Transmission Control Protocol. J. Postel. Sep-01-1981.
- 0815** IP datagram reassembly algorithms. D.D. Clark.] Jul-01-1982.
- 0826** Ethernet Address Resolution Protocol: D.C. Plummer.] Nov-01-1982.
- 0950** Internet Standard Subnetting Procedure. J.C. Mogul, J.] Postel. Aug-01-1985.
- 0961** Official ARPA-Internet protocols. J.K. Reynolds, J.] Postel. Dec-01-1985.
- 1042** Standard for the transmission of IP datagrams over IEEE] 802 networks. J. Postel, J.K. Reynolds. Feb-01-1988.
- 1048** BOOTP vendor information extensions. P.A.] Prindeville. Feb-01-1988.
- 1051** Standard for the transmission of IP datagrams and ARP] packets over ARCNET networks. P.A. Prindeville. Mar-01-1988.
- 1063** IP MTU discovery options. J.C. Mogul, C.A. Kent, C. Partridge,] K. McCloghrie. Jul-01-1988.
- 1112** Host extensions for IP multicasting. S.E. Deering.] Aug-01-1989.
- 1122** Requirements for Internet hosts - communication layers.] R.T. Braden. Oct-01-1989.
- 1191** Path MTU discovery. J.C. Mogul, S.E. Deering.] Nov-01-1990.
- 1201** Transmitting IP traffic over ARCNET networks. D.] Provan. Feb-01-1991.
- 1323** TCP Extensions for High Performance. V. Jacobson, R. Braden,] D. Borman. May 1992.
- 1337** TIME-WAIT Assassination Hazards in TCP. R. Braden. May] 1992.
- 1390** Transmission of IP and ARP over FDDI Networks. D. Katz.] January 1993.
- 1414** Identification MIB. M. StJohns, M. Rose. January] 1993.
- 1469** IP Multicast over Token-Ring Local Area Networks. T.] Pusateri. June 1993.

- 1483** Multiprotocol Encapsulation over ATM Adaptation Layer 5.] Juha Heinanen. July 1993.
- 1490** Multiprotocol Interconnect over Frame Relay. T. Bradley,] C. Brown, A. Malis. July 1993.
- 1512** FDDI Management Information Base. J. Case, A.] Rijsinghani. September 1993.
- 1577** Classical IP and ARP over ATM. M. Laubach. January] 1994.
- 1584** Multicast Extensions to OSPF. J. Moy. March] 1994.
- 1589** A Kernel Model for Precision Timekeeping. D. Mills. March] 1994.
- 1626** Default IP MTU for use over ATM AAL5. R. Atkinson. May] 1994.
- 1701** Generic Routing Encapsulation (GRE). S. Hanks, T. Li,] D. Farinacci, P. Traina. October 1994.
- 1702** Generic Routing Encapsulation over IPv4] networks. S. Hanks,] T. Li, D. Farinacci, P. Traina. October 1994.
- 1762** The PPP DECnet Phase IV Control Protocol (DNCP). S. Senum.] March 1995.
- 1812** Requirements for IP Version 4] Routers. F. Baker. June] 1995.
- 1826** IP Authentication Header. R. Atkinson. August 1995.]
- 1883** Internet Protocol, Version 6] (IPv6) Specification. S. Deering,] R. Hinden. December 1995.
- 1885** Internet Control Message Protocol (ICMPv6) for the] Internet Protocol Version 6] (IPv6). A. Conta, S. Deering. December 1995.
- 1945** Hypertext Transfer Protocol – HTTP/1.0. T. Berners-Lee,] R. Fielding & H. Frystyk. May 1996.
- 2003** IP Encapsulation within IP. C. Perkins. October] 1996.
- 2067** IP over HIPPI. J. Renwick. January] 1997.
- 2131** Dynamic Host Configuration Protocol. R. Droms. March] 1997.
- 2133** Basic Socket Interface Extensions for IPv6. R. Gilligan,] S. Thomson, J. Bound, W. Stevens. April 1997.
- 2207** RSVP Extensions for IPSEC Data Flows. L. Berger, T.] O'Malley. September 1997.
- 2402** IP Authentication Header. S. Kent, R. Atkinson. November] 1998.
- 2460** Internet Protocol, Version 6] (IPv6) Specification. S.] Deering, R. Hinden. December 1998.
- 2481** A Proposal to add Explicit Congestion Notification (ECN) to] IP. K. Ramakrishnan, S. Floyd. January 1999.

- 2516** Method for Transmitting PPP Over Ethernet (PPPoE). L. Mamakos,] K. Lidl, J. Evarts, D. Carrel, D. Simone, R. Wheeler. February 1999.
- 2553** Basic Socket Interface Extensions for IPv6. R. Gilligan,] S. Thomson, J. Bound, W. Stevens. March 1999.
- 2581** TCP Congestion Control. M. Allman, V. Paxson, W. Stevens. April 1999.
- 2582** The NewReno Modification to TCP's Fast Recovery Algorithm.] S. Floyd, T. Henderson. April 1999.
- 2861** TCP Congestion Window Validation. M. Handley, J. Padhye,] S. Floyd. June 2000.
- 2988** Computing TCP's Retransmission Timer. V. Paxson, M.] Allman. November 2000.

Code Authors

The following is a list of addresses that we extracted from the source code - it is possible that some have been missed here, but they, and those here deserve the credit for the system we have been writing about.

chexum@bankinf.banki.hu
Tim_Alpaerts@toyota-motor-europe.com
Aage Kvalnes <aage@cs.uit.no>
Adam Sulmicki <adam@cfar.umd.edu>
Al Longyear <longyear@netcom.com>
Alan Cox, <alan.cox@linux.org>
Alexey Kuznetsov, <kuznet@ms2.inr.ac.ru>
Andrey V. Savochkin <saw@msu.ru>
Arnaldo Carvalho de Melo <acme@conectiva.com.br>, 08/11/1999
Arnt Gulbrandsen, <agulbra@nvg.unit.no>
Charles Hedrick, <hedrick@klinzhai.rutgers.edu>
chihjenc@scf.usc.edu and tsusheng@scf.usc.edu
chris@cybernet.co.nz
Martin Mares <mj@atrey.karlin.mff.cuni.cz>
Corey Minyard <wf-rch!minyard@relay.EU.net>
Dag Brattli <dag@brattli.net>
Dave Airlie, University of Limerick, Ireland <airlied@linux.ie>
David Hinds <dhinds@allegro.stanford.edu>
David S. Miller (davem@caip.rutgers.edu)
Donald Becker, <becker@super.org>
E.M. Serrat emserrat@geocities.com
Eduardo Marcelo Serrat <emserrat@geocities.com>
Emmanuel Roger <winfield@freegates.be>
emserrat@geocities.com
Erik Schoenfelder, <schoenfr@ibr.cs.tu-bs.de>
Florian La Roche <rzsfl@rz.uni-sb.de>
Fred Baumgarten, <dc6iq@insul.etec.uni-karlsruhe.de>
Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
Fred N. van Kempen, <waltje@uwalt.nl.mugnet.org>
Gerald J. Heim, <heim@peanuts.informatik.uni-tuebingen.de>
Herv Eychenne <eychenne@info.enserb.u-bordeaux.fr>
Horst von Brand <vonbrand@sleipnir.valparaiso.cl>
Ian P. Morris <I.P.Morris@soton.ac.uk>
Issam Chehab <ichehab@madge1.demon.co.uk>

J Hadi Salim (hadi@nortelnetworks.com) 1998,1999
Jacques Gelinas (jacques@solucorp.qc.ca)
Jamal Hadi Salim, <hadi@nortelnetworks.com> 990601
Jay Schulist <jschlst@turbolinux.com>
Jay Schulist <jschlst@turbolinux.com>
Jean II - HPL '00 - <jt@hpl.hp.com>
Jean Tourrilhes <jt@hpl.hp.com>
Jrme de Vivie <devivie@info.enserb.u-bordeaux.fr>
Jes Sorensen, <Jes.Sorensen@cern.ch>
Jim Freeman <jfree@caldera.com>
Joerg Reuter DL1BKE <jreuter@yaina.de>
John McDonald <jm@dataprotect.com>
Jonathan Naylor <g4klx@g4klx.demon.co.uk>
Jorge Cwik, <jorge@laser.satlink.net>
Jos Vos <jos@xos.nl> 15/2/1996.
jreuter@yaina.de
Kleen <ak@muc.de>
Laurence Culhane, <loz@holmes.demon.co.uk>
Lawrence V. Stefani, <stefani@lkg.dec.com>
Lennert Buytenhek <buytenh@gnu.org>
Linus Torvalds, <Linus.Torvalds@helsinki.fi>
majordomo@oss.sgi.com
Marc Santoro <ultima@snicker.emoti.com>
Marc Zyngier <mzyngier@freesurf.fr>
Mark Evans, <evansmp@uhura.aston.ac.uk>
Mark.Bush@prg.ox.ac.uk
Marko Kiiiskila carnil@cs.tut.fi
Martijn van Oosterhout <kleptogimp@geocities.com>
Matthew Dillon, <dillon@apollo.west.oic.com>
Michael Neuling <Michael.Neuling@rustcorp.com.au>
Mike <pall@rz.uni-karlsruhe.de>
Mike Shaver <shaver@ingenia.com>
MvS (miquels@drinkel.ow.org)
Olaf Kirch <okir@monad.swb.de>
p.norton@computer.org
Patrick Caulfield <patrick@pandh.demon.co.uk>
Paul Norton <pnorton@cts.com>
Paul Russell <Paul.Russell@rustcorp.com.au>
Pedro Roque <roque@di.fc.ul.pt>
Pekka Riikonen <priikone@poesidon.pspt.fi>
Philip Blundell <philb@gnu.org>
<greg@caldera.com>
by Michael Callahan <callahan@maths.ox.ac.uk>
rco@di.uminho.pt
Report socket allocation statistics [mea@utu.fi]
Richard Underwood <richard@wuzz.demon.co.uk>
Rick Sladkey <jrs@world.std.com>
Ross Biro, <bir7@leland.Stanford.Edu>
Rusty Russell (rusty@rustcorp.com.au)
Sam Lantinga (slouken@cs.ucdavis.edu)

Stefan Becker, <stefanb@yello.ping.de>
Steve Whitehouse <SteveW@ACM.org>
Thomas Davis, <ratbert@radiks.net>,
Thomas Lopatic <tl@dataprotect.com>
Tim Kordas <tjk@nostromo.eeap.cwru.edu>
Tim MacKenzie <tym@dibbler.cs.monash.edu.au>
Trond Myklebust <trond.myklebust@fys.uio.no>
Vineet Abraham <vma@iol.unh.edu>
Wesley Craig <netatalk@umich.edu>
Willy Konynenberg <willy@xos.nl> 10/5/96.
Tim Alpaerts, Tim_Alpaerts@toyota-motor-europe.com

Index

Index

Listings

linux/net/khttpd/waitheaders.c	384
linux/include/linux/ip.h	388
linux/include/linux/tcp.h	390
linux/include/linux/posix_types.h	393
linux/include/asm-i386/types.h	394
linux/include/linux/types.h	395
linux/include/linux/types.h	396
linux/include/linux/types.h	397
linux/kernel/sched.c	399
linux/kernel/sched.c	400
linux/kernel/sched.c	401
linux/kernel/sched.c	402
linux/kernel/sched.c	403
linux/kernel/sched.c	404
linux/kernel/sched.c	405
linux/kernel/sched.c	406
linux/kernel/sched.c	407
linux/arch/i386/kernel/entry.S	408
linux/arch/i386/kernel/entry.S	409
linux/init/main.c	410
linux/init/main.c	411
linux/init/main.c	412
linux/fs/devices.c	413
linux/fs/block_dev.c	414
linux/net/core/dev.c	415
linux/net/core/dev.c	416
linux/net/core/dev.c	417
linux/net/core/dev.c	418
linux/net/core/dev.c	419
linux/net/core/dev.c	420
linux/net/core/dev.c	421
linux/net/core/dev.c	422
linux/include/linux/netdevice.h	424
linux/include/linux/timer.h	425
linux/include/linux/timer.h	426
linux/kernel/timer.c	427
linux/kernel/timer.c	428
linux/net/ipv4/tcp_output.c	431

linux/net/ipv4/ip_output.c	432
linux/net/core/dev.c	433
linux/net/core/dev.c	434
linux/net/core/dev.c	435
linux/net/core/dev.c	436
linux/net/core/sock.c	437
linux/net/ipv4/udp.c	438
linux/net/ipv4/udp.c	439
linux/net/ipv4/udp.c	440
linux/net/ipv4/udp.c	441
linux/net/ipv4/udp.c	442
linux/net/ipv4/udp.c	443
pix/sock.c	448
pix/sock.c	449
pix/sock.c	450
pix/poll.c	451
pix/nbio.c	454
pix/asynch.c	456
linux/include/linux/in.h	457
linux/include/linux/in6.h	458
linux/include/linux/tqueue.h	462
linux/include/linux/tqueue.h	463
linux/include/linux/tqueue.h	464
linux/include/linux/tqueue.h	465
linux/kernel/softirq.c	466
linux/kernel/softirq.c	467
linux/kernel/softirq.c	468
linux/kernel/softirq.c	469
linux/kernel/softirq.c	470
linux/kernel/softirq.c	471
linux/kernel/softirq.c	472
linux/kernel/softirq.c	473
linux/kernel/softirq.c	474
linux/kernel/softirq.c	475
linux/kernel/softirq.c	476
linux/include/linux/net device.h	477
linux/include/linux/net device.h	478
linux/include/linux/net device.h	479
linux/include/linux/net device.h	480
linux/include/linux/net device.h	481
linux/net/core/dev.c	482
linux/drivers/net/3c501.c	483
linux/drivers/net/3c501.c	484
linux/drivers/net/3c501.c	485
linux/drivers/net/3c501.c	486
linux/drivers/net/3c501.c	487
linux/drivers/net/3c501.c	488
linux/drivers/net/3c501.c	489
linux/drivers/net/3c501.c	490
linux/drivers/net/3c501.c	491

linux/drivers/net/3c501.c	492
linux/drivers/net/3c501.c	493
linux/drivers/net/3c501.c	494
linux/drivers/net/3c501.c	495
linux/drivers/net/3c501.c	496
linux/drivers/net/3c501.c	497
linux/drivers/net/3c501.c	498
linux/drivers/net/3c501.c	499
linux/drivers/net/3c501.c	500
linux/drivers/net/3c501.c	501
linux/drivers/net/3c501.c	504
linux/drivers/net/3c501.c	505
linux/net/socket.c	506
linux/net/socket.c	507
linux/net/socket.c	508
linux/net/socket.c	509
linux/net/socket.c	510
linux/net/socket.c	511
linux/net/socket.c	512
linux/net/socket.c	513
linux/net/socket.c	514
linux/net/socket.c	515
linux/net/socket.c	516
linux/net/socket.c	517
linux/net/socket.c	518
linux/net/socket.c	519
linux/net/socket.c	520
linux/net/socket.c	521
linux/net/socket.c	522
linux/net/socket.c	523
linux/net/socket.c	524
linux/net/socket.c	525
linux/net/socket.c	526
linux/net/socket.c	527
linux/net/socket.c	528
linux/net/socket.c	529
linux/net/socket.c	530
linux/net/socket.c	531
linux/net/socket.c	532
linux/net/socket.c	533
linux/net/socket.c	534
linux/net/socket.c	535
linux/net/socket.c	536
linux/net/socket.c	537
linux/net/socket.c	538
linux/net/socket.c	539
linux/net/socket.c	540
linux/net/socket.c	541
linux/net/socket.c	542
linux/net/socket.c	543

linux/net/socket.c	544
linux/net/socket.c	545
linux/net/socket.c	546
linux/net/socket.c	547
linux/net/socket.c	548
linux/net/socket.c	549
linux/net/socket.c	550
linux/net/socket.c	551
linux/net/socket.c	552
linux/net/socket.c	553
linux/net/socket.c	554
linux/net/socket.c	555
linux/net/socket.c	556
linux/net/socket.c	557
linux/net/socket.c	558
linux/net/socket.c	559
linux/net/socket.c	560
linux/net/socket.c	561
linux/net/socket.c	562
linux/net/socket.c	563
linux/net/socket.c	564
linux/net/socket.c	565
linux/net/socket.c	566
linux/net/socket.c	567
linux/net/socket.c	568
linux/net/socket.c	569
linux/net/socket.c	570
linux/net/socket.c	571
linux/net/socket.c	572
linux/net/socket.c	573
linux/net/socket.c	574
linux/net/socket.c	575
linux/include/linux/skbuff.h	577
linux/include/linux/skbuff.h	578
linux/include/linux/skbuff.h	579
linux/net/core/skbuff.c	580
linux/net/core/skbuff.c	581
linux/include/net/tcp.h	582
linux/net/core/skbuff.c	583
linux/net/core/skbuff.c	584
linux/net/core/skbuff.c	585
linux/net/core/skbuff.c	586
linux/net/core/skbuff.c	587
linux/net/core/skbuff.c	588
linux/net/core/skbuff.c	589
linux/net/core/skbuff.c	590
linux/net/core/skbuff.c	591
linux/net/core/skbuff.c	592
linux/net/core/skbuff.c	593
linux/include/net/tcp.h	594

linux/include/net/tcp.h	595
linux/include/net/tcp.h	596
linux/include/net/tcp.h	597
linux/include/net/sock.h	598
linux/include/net/sock.h	599
linux/include/net/sock.h	600
linux/include/net/sock.h	601
linux/include/net/sock.h	602
linux/include/net/sock.h	603
linux/include/net/sock.h	604
linux/include/net/sock.h	605
linux/include/net/sock.h	606
linux/include/net/sock.h	607
linux/include/net/sock.h	608
linux/include/net/sock.h	609
linux/include/net/sock.h	610
linux/include/net/sock.h	611
linux/include/net/sock.h	612
linux/include/net/sock.h	613
linux/include/net/sock.h	614
linux/include/net/sock.h	615
linux/include/net/sock.h	616
linux/include/net/sock.h	617
linux/include/net/sock.h	618
linux/net/core/scm.c	619
linux/include/net/scm.h	620
linux/net/ipv4/ip_output.c	624
linux/net/ipv4/ip_output.c	625
linux/net/ipv4/ip_output.c	626
linux/net/ipv4/ip_output.c	627
linux/net/ipv4/ip_output.c	628
linux/net/ipv4/ip_output.c	629
linux/net/ipv4/ip_output.c	630
linux/net/ipv4/ip_output.c	631
linux/net/ipv4/ip_output.c	632
linux/net/ipv4/ip_output.c	633
linux/net/ipv4/ip_output.c	634
linux/net/ipv4/ip_output.c	635
linux/net/ipv4/ip_output.c	636
linux/net/ipv4/ip_output.c	637
linux/net/ipv4/ip_output.c	638
linux/net/ipv4/ip_output.c	639
linux/net/ipv4/ip_output.c	640
linux/net/ipv4/ip_output.c	641
linux/net/ipv4/ip_output.c	642
linux/net/ipv4/ip_output.c	643
linux/net/ipv4/ip_output.c	644
linux/net/ipv4/ip_output.c	645
linux/net/ipv4/ip_output.c	646
linux/net/ipv4/ip_output.c	647

linux/net/ipv4/ip_output.c	648
linux/net/ipv4/ip_output.c	649
linux/net/ipv4/ip_output.c	650
linux/net/ipv4/ip_output.c	651
linux/net/ipv4/ip_output.c	652
linux/net/ipv4/ip_output.c	653
linux/net/ipv4/ip_output.c	654
linux/net/ipv4/ip_output.c	655
linux/net/ipv4/ip_input.c	656
linux/net/ipv4/ip_input.c	657
linux/net/ipv4/ip_input.c	658
linux/net/ipv4/ip_input.c	659
linux/net/ipv4/ip_input.c	660
linux/net/ipv4/ip_input.c	661
linux/net/ipv4/ip_input.c	662
linux/net/ipv4/ip_input.c	663
linux/net/ipv4/ip_input.c	664
linux/net/ipv4/ip_input.c	665
linux/net/ipv4/ip_forward.c	666
linux/net/ipv4/ip_forward.c	667
linux/net/ipv4/ip_forward.c	668
linux/net/ipv4/ip_fragment.c	669
linux/net/ipv4/ip_fragment.c	670
linux/net/ipv4/ip_fragment.c	671
linux/net/ipv4/ip_fragment.c	672
linux/net/ipv4/ip_fragment.c	673
linux/net/ipv4/ip_fragment.c	674
linux/net/ipv4/ip_fragment.c	675
linux/net/ipv4/ip_fragment.c	676
linux/net/ipv4/ip_fragment.c	677
linux/net/ipv4/ip_fragment.c	678
linux/net/ipv4/ip_fragment.c	679
linux/net/ipv4/ip_fragment.c	680
linux/net/ipv4/ip_fragment.c	681
linux/net/ipv4/ip_fragment.c	682
linux/net/ipv4/ip_fragment.c	683
linux/net/ipv4/ip_fragment.c	684
linux/net/ipv4/ip_fragment.c	685
linux/net/ipv4/ip_fragment.c	686
linux/net/ipv4/ip_options.c	687
linux/net/ipv4/ip_options.c	688
linux/net/ipv4/ip_options.c	689
linux/net/ipv4/ip_options.c	690
linux/net/ipv4/ip_options.c	691
linux/net/ipv4/ip_options.c	692
linux/net/ipv4/ip_options.c	693
linux/net/ipv4/ip_options.c	694
linux/net/ipv4/ip_options.c	695
linux/net/ipv4/ip_options.c	696
linux/net/ipv4/ip_options.c	697

linux/net/ipv4/ip_options.c	698
linux/net/ipv4/ip_options.c	699
linux/net/ipv4/ip_options.c	700
linux/net/ipv4/ip_options.c	701
linux/net/ipv4/ip_options.c	702
linux/net/ipv4/ip_options.c	703
linux/net/ipv4/ip_options.c	704
linux/net/ipv4/ip_sockglue.c	705
linux/net/ipv4/ip_sockglue.c	706
linux/net/ipv4/ip_sockglue.c	707
linux/net/ipv4/ip_sockglue.c	708
linux/net/ipv4/ip_sockglue.c	709
linux/net/ipv4/ip_sockglue.c	710
linux/net/ipv4/ip_sockglue.c	711
linux/net/ipv4/ip_sockglue.c	712
linux/net/ipv4/ip_sockglue.c	713
linux/net/ipv4/ip_sockglue.c	714
linux/net/ipv4/ip_sockglue.c	715
linux/net/ipv4/ip_sockglue.c	716
linux/net/ipv4/ip_sockglue.c	717
linux/net/ipv4/ip_sockglue.c	718
linux/net/ipv4/ip_sockglue.c	719
linux/net/ipv4/ip_sockglue.c	720
linux/net/ipv4/ip_sockglue.c	721
linux/net/ipv4/ip_sockglue.c	722
linux/net/ipv4/ip_sockglue.c	723
linux/net/ipv4/ip_sockglue.c	724
linux/net/ipv4/ip_sockglue.c	725
linux/net/ipv4/ip_sockglue.c	726
linux/net/ipv4/ip_sockglue.c	727
linux/net/ipv4/ip_sockglue.c	728
linux/net/ipv4/arp.c	729
linux/net/ipv4/arp.c	730
linux/net/ipv4/arp.c	731
linux/net/ipv4/arp.c	732
linux/net/ipv4/arp.c	733
linux/net/ipv4/arp.c	734
linux/net/ipv4/arp.c	735
linux/net/ipv4/arp.c	736
linux/net/ipv4/arp.c	737
linux/net/ipv4/arp.c	738
linux/net/ipv4/arp.c	739
linux/net/ipv4/arp.c	740
linux/net/ipv4/arp.c	741
linux/net/ipv4/arp.c	742
linux/net/ipv4/arp.c	743
linux/net/ipv4/arp.c	744
linux/net/ipv4/arp.c	745
linux/net/ipv4/arp.c	746
linux/net/ipv4/arp.c	747

linux/net/ipv4/arp.c	748
linux/net/ipv4/arp.c	749
linux/net/ipv4/arp.c	750
linux/net/ipv4/arp.c	751
linux/net/ipv4/arp.c	752
linux/net/ipv4/arp.c	753
linux/net/ipv4/arp.c	754
linux/net/ipv4/arp.c	755
linux/net/ipv4/arp.c	756
linux/net/ipv4/arp.c	757
linux/net/ipv4/arp.c	758
linux/net/ipv4/arp.c	759
linux/net/ipv4/arp.c	760
linux/net/ipv4/arp.c	761
linux/net/ipv4/arp.c	762
linux/net/ipv4/arp.c	763
linux/net/ipv4/arp.c	764
linux/net/ipv4/arp.c	765
linux/net/ipv4/icmp.c	766
linux/net/ipv4/icmp.c	767
linux/net/ipv4/icmp.c	768
linux/net/ipv4/icmp.c	769
linux/net/ipv4/icmp.c	770
linux/net/ipv4/icmp.c	771
linux/net/ipv4/icmp.c	772
linux/net/ipv4/icmp.c	773
linux/net/ipv4/icmp.c	774
linux/net/ipv4/icmp.c	775
linux/net/ipv4/icmp.c	776
linux/net/ipv4/icmp.c	777
linux/net/ipv4/icmp.c	778
linux/net/ipv4/icmp.c	779
linux/net/ipv4/icmp.c	780
linux/net/ipv4/icmp.c	781
linux/net/ipv4/icmp.c	782
linux/net/ipv4/icmp.c	783
linux/net/ipv4/icmp.c	784
linux/net/ipv4/icmp.c	785
linux/net/ipv4/icmp.c	786
linux/net/ipv4/icmp.c	787
linux/net/ipv4/icmp.c	788
linux/net/ipv4/icmp.c	789
linux/net/ipv4/icmp.c	790
linux/net/ipv4/icmp.c	791
linux/net/ipv4/icmp.c	792
linux/net/ipv4/icmp.c	793
linux/net/ipv4/icmp.c	794
linux/net/ipv4/icmp.c	795
linux/net/ipv4/icmp.c	796
linux/net/ipv4/icmp.c	797

linux/net/ipv4/igmp.c	798
linux/net/ipv4/igmp.c	799
linux/net/ipv4/igmp.c	800
linux/net/ipv4/igmp.c	801
linux/net/ipv4/igmp.c	802
linux/net/ipv4/igmp.c	803
linux/net/ipv4/igmp.c	804
linux/net/ipv4/igmp.c	805
linux/net/ipv4/igmp.c	806
linux/net/ipv4/igmp.c	807
linux/net/ipv4/igmp.c	808
linux/net/ipv4/igmp.c	809
linux/net/ipv4/igmp.c	810
linux/net/ipv4/igmp.c	811
linux/net/ipv4/igmp.c	812
linux/net/ipv4/igmp.c	813
linux/net/ipv4/igmp.c	814
linux/net/ipv4/igmp.c	815
linux/net/ipv4/igmp.c	816
linux/net/ipv4/igmp.c	817
linux/net/ipv4/ipmr.c	818
linux/net/ipv4/igmp.c	819
linux/net/ipv4/igmp.c	820
linux/net/ipv4/igmp.c	821
linux/net/ipv4/ipmr.c	822
linux/net/ipv4/ipmr.c	823
linux/net/ipv4/ipmr.c	824
linux/net/ipv4/ipmr.c	825
linux/net/ipv4/ipmr.c	826
linux/net/ipv4/ipmr.c	827
linux/net/ipv4/ipmr.c	828
linux/net/ipv4/ipmr.c	829
linux/net/ipv4/ipmr.c	830
linux/net/ipv4/ipmr.c	831
linux/net/ipv4/ipmr.c	832
linux/net/ipv4/ipmr.c	833
linux/net/ipv4/ipmr.c	834
linux/net/ipv4/ipmr.c	835
linux/net/ipv4/ipmr.c	836
linux/net/ipv4/ipmr.c	837
linux/net/ipv4/ipmr.c	838
linux/net/ipv4/ipmr.c	839
linux/net/ipv4/ipmr.c	840
linux/net/ipv4/ipmr.c	841
linux/net/ipv4/ipmr.c	842
linux/net/ipv4/ipmr.c	843
linux/net/ipv4/ipmr.c	844
linux/net/ipv4/ipmr.c	845
linux/net/ipv4/ipmr.c	846
linux/net/ipv4/ipmr.c	847

linux/net/ipv4/ipmr.c	848
linux/net/ipv4/ipmr.c	849
linux/net/ipv4/ipmr.c	850
linux/net/ipv4/ipmr.c	851
linux/net/ipv4/ipmr.c	852
linux/net/ipv4/ipmr.c	853
linux/net/ipv4/ipmr.c	854
linux/net/ipv4/ipmr.c	855
linux/net/ipv4/ipmr.c	856
linux/net/ipv4/ipmr.c	857
linux/net/ipv4/ipmr.c	858
linux/net/ipv4/ipmr.c	859
linux/net/ipv4/ipmr.c	860
linux/net/ipv4/ipmr.c	861
linux/net/ipv4/ipmr.c	862
linux/net/ipv4/ipmr.c	863
linux/net/ipv4/ipmr.c	864
linux/net/ipv4/ipmr.c	865
linux/net/ipv4/ipmr.c	866
linux/net/ipv4/ipmr.c	867
linux/net/ipv4/ipmr.c	868
linux/net/ipv4/ipmr.c	869
linux/net/ipv4/ip_nat_dumb.c	870
linux/net/ipv4/ip_nat_dumb.c	871
linux/net/ipv4/ip_nat_dumb.c	872
linux/net/ipv4/ip_nat_dumb.c	873
linux/net/ipv4/ipip.c	874
linux/net/ipv4/ipip.c	875
linux/net/ipv4/ipip.c	876
linux/net/ipv4/ipip.c	877
linux/net/ipv4/ipip.c	878
linux/net/ipv4/ipip.c	879
linux/net/ipv4/ipip.c	880
linux/net/ipv4/ipip.c	881
linux/net/ipv4/ipip.c	882
linux/net/ipv4/ipip.c	883
linux/net/ipv4/ipip.c	884
linux/net/ipv4/ipip.c	885
linux/net/ipv4/ipip.c	886
linux/net/ipv4/ipip.c	887
linux/net/ipv4/ipip.c	888
linux/net/ipv4/ipip.c	889
linux/net/ipv4/ipip.c	890
linux/net/ipv4/ipip.c	891
linux/net/ipv4/ipip.c	892
linux/net/ipv4/ipip.c	893
linux/net/ipv4/ipip.c	894
linux/net/ipv4/ipip.c	895
linux/net/ipv4/ipip.c	896
linux/net/ipv4/ipip.c	897

linux/net/ipv4/ipip.c	898
linux/net/ipv4/ipip.c	899
linux/net/ipv4/ipip.c	900
linux/net/ipv4/ipip.c	901
linux/net/ipv4/ipip.c	902
linux/net/ipv4/ipip.c	903
linux/net/ipv4/ip_gre.c	904
linux/net/ipv4/ip_gre.c	905
linux/net/ipv4/ip_gre.c	906
linux/net/ipv4/ip_gre.c	907
linux/net/ipv4/ip_gre.c	908
linux/net/ipv4/ip_gre.c	909
linux/net/ipv4/ip_gre.c	910
linux/net/ipv4/ip_gre.c	911
linux/net/ipv4/ip_gre.c	912
linux/net/ipv4/ip_gre.c	913
linux/net/ipv4/ip_gre.c	914
linux/net/ipv4/ip_gre.c	915
linux/net/ipv4/ip_gre.c	916
linux/net/ipv4/ip_gre.c	917
linux/net/ipv4/ip_gre.c	918
linux/net/ipv4/ip_gre.c	919
linux/net/ipv4/ip_gre.c	920
linux/net/ipv4/ip_gre.c	921
linux/net/ipv4/ip_gre.c	922
linux/net/ipv4/ip_gre.c	923
linux/net/ipv4/ip_gre.c	924
linux/net/ipv4/ip_gre.c	925
linux/net/ipv4/ip_gre.c	926
linux/net/ipv4/ip_gre.c	927
linux/net/ipv4/ip_gre.c	928
linux/net/ipv4/ip_gre.c	929
linux/net/ipv4/ip_gre.c	930
linux/net/ipv4/ip_gre.c	931
linux/net/ipv4/ip_gre.c	932
linux/net/ipv4/ip_gre.c	933
linux/net/ipv4/ip_gre.c	934
linux/net/ipv4/ip_gre.c	935
linux/net/ipv4/ip_gre.c	936
linux/net/ipv4/ip_gre.c	937
linux/net/ipv4/ip_gre.c	938
linux/net/ipv4/ip_gre.c	939
linux/net/ipv4/ip_gre.c	940
linux/net/ipv4/ip_gre.c	941
linux/net/ipv4/ip_gre.c	942
linux/include/linux/ipv6.h	943
linux/include/net/ipv6.h	944
linux/include/net/ipv6.h	945
linux/include/net/ipv6.h	946
linux/include/net/if_inet6.h	947

linux/include/net/if_inet6.h	948
linux/include/net/if_inet6.h	949
linux/include/net/if_inet6.h	950
linux/include/net/if_inet6.h	951
linux/include/linux/in6.h	952
linux/include/linux/in6.h	953
linux/include/linux/in6.h	954
linux/net/ipv6/addrconf.c	955
linux/net/ipv6/addrconf.c	956
linux/net/ipv6/addrconf.c	957
linux/net/ipv6/addrconf.c	958
linux/net/ipv6/addrconf.c	959
linux/net/ipv6/addrconf.c	960
linux/net/ipv6/addrconf.c	961
linux/net/ipv6/addrconf.c	962
linux/net/ipv6/addrconf.c	963
linux/net/ipv6/addrconf.c	964
linux/net/ipv6/addrconf.c	965
linux/net/ipv6/addrconf.c	966
linux/net/ipv6/addrconf.c	967
linux/net/ipv6/addrconf.c	968
linux/net/ipv6/addrconf.c	969
linux/net/ipv6/addrconf.c	970
linux/net/ipv6/ndisc.c	971
linux/net/ipv6/ndisc.c	972
linux/net/ipv6/ndisc.c	973
linux/net/ipv6/ndisc.c	974
linux/net/ipv6/ndisc.c	975
linux/net/ipv6/ndisc.c	976
linux/net/ipv6/ndisc.c	977
linux/net/ipv6/ndisc.c	978
linux/net/ipv6/ip6_input.c	979
linux/net/ipv6/ip6_input.c	980
linux/net/ipv6/ip6_input.c	981
linux/net/ipv6/ip6_output.c	982
linux/net/ipv6/ip6_output.c	983
linux/net/ipv6/ip6_output.c	984
linux/net/ipv6/ip6_output.c	985
linux/net/ipv6/ip6_output.c	986
linux/net/ipv6/ip6_output.c	987
linux/net/ipv6/ip6_output.c	988
linux/net/ipv6/ip6_output.c	989
linux/net/ipv6/ip6_output.c	990
linux/net/ipv6/ip6_output.c	991
linux/net/ipv6/ip6_output.c	992
linux/net/ipv6/ip6_output.c	993
linux/net/ipv6/ipv6_sockglue.c	994
linux/net/ipv6/ipv6_sockglue.c	995
linux/net/ipv6/ipv6_sockglue.c	996
linux/net/ipv6/ipv6_sockglue.c	997

linux/net/ipv6/ipv6_sockglue.c	998
linux/net/ipv6/ipv6_sockglue.c	999
linux/net/ipv6/ipv6_sockglue.c	1000
linux/net/ipv6/ipv6_flowlabel.c	1001
linux/net/ipv6/ipv6_flowlabel.c	1002
linux/net/ipv6/ipv6_flowlabel.c	1003
linux/net/ipv6/exthdrs.c	1004
linux/net/ipv6/exthdrs.c	1005
linux/net/ipv6/sit.c	1006
linux/net/ipv6/sit.c	1007
linux-2.4.4/net/ipv4/af_inet.c	1010
linux-2.4.4/net/ipv4/udp.c	1011
linux-2.4.4/net/ipv4/tcp_ipv4.c	1011
linux-2.4.4/net/ipv4/tcp_ipv4.c	1012
linux-2.4.4/include/net/sock.h	1013
linux-2.4.4/net/ipv4/udp.c	1015
linux-2.4.4/net/ipv4/udp.c	1016
linux-2.4.4/net/ipv4/udp.c	1017
linux-2.4.4/net/ipv4/udp.c	1018
linux-2.4.4/net/ipv4/udp.c	1019
linux-2.4.4/include/net/sock.h	1029
linux-2.4.4/include/net/sock.h	1030
linux-2.4.4/include/net/sock.h	1031
linux-2.4.4/include/net/sock.h	1032
linux-2.4.4/include/net/sock.h	1033
linux-2.4.4/include/net/tcp.h	1034
linux-2.4.4/net/ipv4/tcp.c	1035
linux-2.4.4/include/net/tcp.h	1036
linux-2.4.4/net/ipv4/tcp.c	1037
linux-2.4.4/net/ipv4/tcp.c	1038
linux-2.4.4/net/ipv4/tcp.c	1039
linux-2.4.4/net/ipv4/tcp.c	1040
linux-2.4.4/net/ipv4/tcp.c	1041
linux-2.4.4/net/ipv4/tcp_output.c	1042
linux-2.4.4/include/net/tcp.h	1043
linux-2.4.4/net/ipv4/tcp_output.c	1044
linux-2.4.4/net/ipv4/tcp_output.c	1045
linux-2.4.4/net/ipv4/tcp_output.c	1046
linux-2.4.4/net/ipv4/tcp_ipv4.c	1047
linux-2.4.4/net/ipv4/tcp_ipv4.c	1048
linux-2.4.4/net/ipv4/tcp_ipv4.c	1049
linux-2.4.4/include/net/tcp.h	1050
linux-2.4.4/net/ipv4/tcp_input.c	1051
linux-2.4.4/net/ipv4/tcp_input.c	1052
linux-2.4.4/net/ipv4/tcp_input.c	1053
linux-2.4.4/net/ipv4/tcp_input.c	1054
linux-2.4.4/net/ipv4/tcp_input.c	1055
linux-2.4.4/net/ipv4/tcp_input.c	1056
linux-2.4.4/include/net/tcp.h	1057
linux-2.4.4/net/ipv4/tcp_input.c	1058

linux-2.4.4/net/ipv4/tcp_input.c	1059
linux-2.4.4/net/ipv4/tcp_input.c	1060
linux-2.4.4/net/ipv4/tcp_input.c	1061
linux-2.4.4/net/ipv4/tcp_input.c	1062
linux-2.4.4/net/ipv4/tcp_input.c	1063
linux-2.4.4/net/ipv4/tcp_input.c	1064
linux-2.4.4/net/ipv4/tcp_timer.c	1065
linux-2.4.4/net/ipv4/tcp_input.c	1066
linux-2.4.4/net/ipv4/tcp_input.c	1067
linux-2.4.4/net/ipv4/tcp_input.c	1067
linux-2.4.4/net/ipv4/tcp_input.c	1068
linux-2.4.4/net/ipv4/tcp_input.c	1069
linux-2.4.4/include/net/tcp.h	1070
linux-2.4.4/net/ipv4/tcp_input.c	1071
linux-2.4.4/include/net/tcp.h	1072
linux-2.4.4/net/ipv4/tcp_input.c	1073
linux-2.4.4/net/ipv4/tcp_input.c	1074
linux-2.4.4/net/ipv4/tcp_input.c	1075
linux-2.4.4/include/net/tcp.h	1076
linux-2.4.4/net/ipv4/tcp_input.c	1077
linux/include/net/ip_fib.h	1081
linux/include/net/ip_fib.h	1082
linux/include/net/ip_fib.h	1083
linux/include/net/ip_fib.h	1084
linux/include/net/ip_fib.h	1085
linux/net/ipv4/fib_frontend.c	1086
linux/net/ipv4/fib_frontend.c	1087
linux/net/ipv4/fib_frontend.c	1088
linux/net/ipv4/fib_frontend.c	1089
linux/net/ipv4/fib_frontend.c	1090
linux/net/ipv4/fib_frontend.c	1091
linux/net/ipv4/fib_frontend.c	1092
linux/net/ipv4/fib_frontend.c	1093
linux/net/ipv4/fib_frontend.c	1094
linux/net/ipv4/fib_frontend.c	1095
linux/net/ipv4/fib_frontend.c	1096
linux/net/ipv4/fib_frontend.c	1097
linux/net/ipv4/fib_frontend.c	1098
linux/net/ipv4/fib_frontend.c	1099
linux/net/ipv4/fib_frontend.c	1100
linux/net/ipv4/fib_frontend.c	1101
linux/net/ipv4/fib_frontend.c	1102
linux/net/ipv4/fib_frontend.c	1103
linux/net/ipv4/fib_frontend.c	1104
linux/net/ipv4/fib_frontend.c	1105
linux/net/ipv4/fib_semantics.c	1106
linux/net/ipv4/fib_semantics.c	1107
linux/net/ipv4/fib_semantics.c	1108
linux/net/ipv4/fib_semantics.c	1109
linux/net/ipv4/fib_semantics.c	1110

linux/net/ipv4/fib_semantics.c	1111
linux/net/ipv4/fib_semantics.c	1112
linux/net/ipv4/fib_semantics.c	1113
linux/net/ipv4/fib_semantics.c	1114
linux/net/ipv4/fib_semantics.c	1115
linux/net/ipv4/fib_semantics.c	1116
linux/net/ipv4/fib_semantics.c	1117
linux/net/ipv4/fib_semantics.c	1118
linux/net/ipv4/fib_semantics.c	1119
linux/net/ipv4/fib_semantics.c	1120
linux/net/ipv4/fib_semantics.c	1121
linux/net/ipv4/fib_semantics.c	1122
linux/net/ipv4/fib_semantics.c	1123
linux/net/ipv4/fib_semantics.c	1124
linux/net/ipv4/fib_semantics.c	1125
linux/net/ipv4/fib_semantics.c	1126
linux/net/ipv4/fib_semantics.c	1127
linux/net/ipv4/fib_semantics.c	1128
linux/net/ipv4/fib_semantics.c	1129
linux/net/ipv4/fib_semantics.c	1130
linux/net/ipv4/fib_semantics.c	1131
linux/net/ipv4/fib_semantics.c	1132
linux/net/ipv4/fib_semantics.c	1133
linux/net/ipv4/fib_semantics.c	1134
linux/net/ipv4/fib_semantics.c	1135
linux/net/ipv4/fib_rules.c	1136
linux/net/ipv4/fib_rules.c	1137
linux/net/ipv4/fib_rules.c	1138
linux/net/ipv4/fib_rules.c	1139
linux/net/ipv4/fib_rules.c	1140
linux/net/ipv4/fib_rules.c	1141
linux/net/ipv4/fib_rules.c	1142
linux/net/ipv4/fib_rules.c	1143
linux/net/ipv4/fib_rules.c	1144
linux/net/ipv4/fib_rules.c	1145
linux/net/ipv4/fib_rules.c	1146
linux/net/ipv4/fib_rules.c	1147
linux/net/ipv4/fib_rules.c	1148
linux/net/ipv4/fib_hash.c	1149
linux/net/ipv4/fib_hash.c	1150
linux/net/ipv4/fib_hash.c	1151
linux/net/ipv4/fib_hash.c	1152
linux/net/ipv4/fib_hash.c	1153
linux/net/ipv4/fib_hash.c	1154
linux/net/ipv4/fib_hash.c	1155
linux/net/ipv4/fib_hash.c	1156
linux/net/ipv4/fib_hash.c	1157
linux/net/ipv4/fib_hash.c	1158
linux/net/ipv4/fib_hash.c	1159
linux/net/ipv4/fib_hash.c	1160

linux/net/ipv4/fib_hash.c	1161
linux/net/ipv4/fib_hash.c	1162
linux/net/ipv4/fib_hash.c	1163
linux/net/ipv4/fib_hash.c	1164
linux/net/ipv4/fib_hash.c	1165
linux/net/ipv4/fib_hash.c	1166
linux/net/ipv4/fib_hash.c	1167
linux/net/ipv4/fib_hash.c	1168
linux/net/ipv4/fib_hash.c	1169
linux/net/ipv4/fib_hash.c	1170
linux/net/ipv4/fib_hash.c	1171
linux/net/ipv4/fib_hash.c	1172
linux/net/ipv4/fib_hash.c	1173
linux/net/ipv4/fib_hash.c	1174
linux/include/net/route.h	1177
linux/include/net/route.h	1178
linux/include/net/route.h	1179
linux/net/ipv4/route.c	1180
linux/net/ipv4/route.c	1181
linux/net/core/rtnetlink.c	1182
linux/include/net/route.h	1183
linux/include/net/route.h	1184
linux/net/ipv4/route.c	1185
linux/net/ipv4/route.c	1186
linux/net/ipv4/route.c	1187
linux/net/ipv4/route.c	1188
linux/net/ipv4/route.c	1189
linux/net/ipv4/route.c	1190
linux/net/ipv4/route.c	1191
linux/net/ipv4/route.c	1192
linux/net/ipv4/route.c	1193
linux/net/ipv4/route.c	1194
linux/net/sched/cls_api.c	1206
linux/net/sched/cls_api.c	1207
linux/net/sched/cls_api.c	1208
linux/net/sched/cls_api.c	1209
linux/net/sched/cls_api.c	1210
linux/net/sched/cls_api.c	1211
linux/net/sched/cls_api.c	1212
linux/net/sched/cls_api.c	1213
linux/net/sched/cls_api.c	1214
linux/net/sched/cls_api.c	1215
linux/net/sched/cls_api.c	1216
linux/net/sched/cls_api.c	1217
linux/net/sched/cls_api.c	1218
linux/net/sched/cls_api.c	1219
linux/net/sched/cls_api.c	1220
linux/net/sched/cls_api.c	1221
linux/net/sched/cls_u32.c	1222
linux/net/sched/cls_u32.c	1223

linux/net/sched/cls_u32.c	1224
linux/net/sched/cls_u32.c	1225
linux/net/sched/cls_u32.c	1226
linux/net/sched/cls_u32.c	1227
linux/net/sched/cls_u32.c	1228
linux/net/sched/cls_u32.c	1229
linux/net/sched/cls_u32.c	1230
linux/net/sched/cls_u32.c	1231
linux/net/sched/cls_u32.c	1232
linux/net/sched/cls_u32.c	1233
linux/net/sched/cls_u32.c	1234
linux/net/sched/cls_u32.c	1235
linux/net/sched/cls_u32.c	1236
linux/net/sched/cls_rsvp.h	1237
linux/net/sched/cls_rsvp.h	1238
linux/net/sched/cls_rsvp.h	1239
linux/net/sched/cls_rsvp.h	1240
linux/net/sched/cls_rsvp.h	1241
linux/net/sched/cls_rsvp.h	1242
linux/net/sched/cls_rsvp.h	1243
linux/net/sched/cls_rsvp.h	1244
linux/net/sched/cls_rsvp.h	1245
linux/net/sched/cls_rsvp.h	1246
linux/net/sched/cls_rsvp.h	1247
linux/net/sched/cls_rsvp.h	1248
linux/net/sched/cls_rsvp.h	1249
linux/net/sched/cls_rsvp.h	1250
linux/net/sched/cls_rsvp.h	1251
linux/net/sched/cls_rsvp.h	1252
linux/net/sched/cls_rsvp.c	1253
linux/net/sched/cls_route.c	1254
linux/net/sched/cls_route.c	1255
linux/net/sched/cls_route.c	1256
linux/net/sched/cls_route.c	1257
linux/net/sched/cls_route.c	1258
linux/net/sched/cls_route.c	1259
linux/net/sched/cls_route.c	1260
linux/net/sched/cls_route.c	1261
linux/net/sched/cls_fw.c	1262
linux/net/sched/cls_fw.c	1263
linux/net/sched/cls_fw.c	1264
linux/net/sched/cls_fw.c	1265
linux/net/sched/cls_fw.c	1266
linux/net/sched/cls_fw.c	1267
linux/net/sched/cls_fw.c	1268
linux/net/sched/cls_fw.c	1269
linux/net/sched/cls_fw.c	1270
linux/net/sched/cls_tindex.c	1271
linux/net/sched/cls_tindex.c	1272
linux/net/sched/cls_tindex.c	1273

linux/net/sched/cls_tcindex.c	1274
linux/net/sched/cls_tcindex.c	1275
linux/net/sched/cls_tcindex.c	1276
linux/net/sched/cls_tcindex.c	1277
linux/net/sched/cls_tcindex.c	1278
linux/net/sched/cls_tcindex.c	1279
linux/net/sched/cls_tcindex.c	1280
linux/net/sched/sch_api.c	1281
linux/net/sched/sch_api.c	1282
linux/net/sched/sch_api.c	1283
linux/net/sched/sch_api.c	1284
linux/net/sched/sch_api.c	1285
linux/net/sched/sch_api.c	1286
linux/net/sched/sch_api.c	1287
linux/net/sched/sch_api.c	1288
linux/net/sched/sch_api.c	1289
linux/net/sched/sch_api.c	1290
linux/net/sched/sch_api.c	1291
linux/net/sched/sch_api.c	1292
linux/net/sched/sch_api.c	1293
linux/net/sched/sch_api.c	1294
linux/net/sched/sch_api.c	1295
linux/net/sched/sch_api.c	1296
linux/net/sched/sch_api.c	1297
linux/net/sched/sch_api.c	1298
linux/net/sched/sch_api.c	1299
linux/net/sched/sch_api.c	1300
linux/net/sched/sch_api.c	1301
linux/net/sched/sch_api.c	1302
linux/net/sched/sch_api.c	1303
linux/net/sched/sch_api.c	1304
linux/net/sched/sch_api.c	1305
linux/net/sched/sch_api.c	1306
linux/net/sched/sch_api.c	1307
linux/net/sched/sch_api.c	1308
linux/net/sched/sch_api.c	1309
linux/net/sched/sch_api.c	1310
linux/net/sched/sch_api.c	1311
linux/net/sched/sch_api.c	1312
linux/net/sched/sch_api.c	1313
linux/net/sched/sch_api.c	1314
linux/net/sched/sch_api.c	1315
linux/net/sched/sch_api.c	1316
linux/include/net/pkt_sched.h	1317
linux/include/net/pkt_sched.h	1318
linux/include/net/pkt_sched.h	1319
linux/include/net/pkt_sched.h	1320
linux/include/net/pkt_sched.h	1321
linux/include/net/pkt_sched.h	1322
linux/net/sched/sch_api.c	1323

linux/net/sched/sch_api.c	1324
linux/net/sched/sch_api.c	1325
linux/net/sched/sch_api.c	1326
linux/net/sched/sch_api.c	1327
linux/net/sched/sch_generic.c	1328
linux/net/sched/sch_generic.c	1329
linux/net/sched/sch_generic.c	1330
linux/net/sched/sch_generic.c	1331
linux/net/sched/sch_generic.c	1332
linux/net/sched/sch_generic.c	1333
linux/net/sched/sch_generic.c	1334
linux/net/sched/sch_generic.c	1335
linux/net/sched/sch_generic.c	1336
linux/net/sched/sch_generic.c	1337
linux/net/sched/sch_generic.c	1338
linux/net/sched/sch_cbq.c	1339
linux/net/sched/sch_cbq.c	1340
linux/net/sched/sch_cbq.c	1341
linux/net/sched/sch_cbq.c	1342
linux/net/sched/sch_cbq.c	1343
linux/net/sched/sch_cbq.c	1344
linux/net/sched/sch_cbq.c	1345
linux/net/sched/sch_cbq.c	1346
linux/net/sched/sch_cbq.c	1347
linux/net/sched/sch_cbq.c	1348
linux/net/sched/sch_cbq.c	1349
linux/net/sched/sch_cbq.c	1350
linux/net/sched/sch_cbq.c	1351
linux/net/sched/sch_cbq.c	1352
linux/net/sched/sch_cbq.c	1353
linux/net/sched/sch_cbq.c	1354
linux/net/sched/sch_cbq.c	1355
linux/net/sched/sch_cbq.c	1356
linux/net/sched/sch_cbq.c	1357
linux/net/sched/sch_cbq.c	1358
linux/net/sched/sch_cbq.c	1359
linux/net/sched/sch_cbq.c	1360
linux/net/sched/sch_cbq.c	1361
linux/net/sched/sch_cbq.c	1362
linux/net/sched/sch_cbq.c	1363
linux/net/sched/sch_cbq.c	1364
linux/net/sched/sch_cbq.c	1365
linux/net/sched/sch_cbq.c	1366
linux/net/sched/sch_cbq.c	1367
linux/net/sched/sch_cbq.c	1368
linux/net/sched/sch_cbq.c	1369
linux/net/sched/sch_cbq.c	1370
linux/net/sched/sch_cbq.c	1371
linux/net/sched/sch_cbq.c	1372
linux/net/sched/sch_cbq.c	1373

linux/net/sched/sch_cbq.c	1374
linux/net/sched/sch_cbq.c	1375
linux/net/sched/sch_cbq.c	1376
linux/net/sched/sch_cbq.c	1377
linux/net/sched/sch_cbq.c	1378
linux/net/sched/sch_cbq.c	1379
linux/net/sched/sch_cbq.c	1380
linux/net/sched/sch_cbq.c	1381
linux/net/sched/sch_cbq.c	1382
linux/net/sched/sch_cbq.c	1383
linux/net/sched/sch_cbq.c	1384
linux/net/sched/sch_cbq.c	1385
linux/net/sched/sch_cbq.c	1386
linux/net/sched/sch_cbq.c	1387
linux/net/sched/sch_cbq.c	1388
linux/net/sched/sch_csz.c	1389
linux/net/sched/sch_csz.c	1390
linux/net/sched/sch_csz.c	1391
linux/net/sched/sch_csz.c	1392
linux/net/sched/sch_csz.c	1393
linux/net/sched/sch_csz.c	1394
linux/net/sched/sch_csz.c	1395
linux/net/sched/sch_fifo.c	1396
linux/net/sched/sch_fifo.c	1397
linux/net/sched/sch_fifo.c	1398
linux/net/sched/sch_fifo.c	1399
linux/net/sched/sch_fifo.c	1400
linux/net/sched/sch_fifo.c	1401
linux/net/sched/sch_fifo.c	1402
linux/net/sched/sch_fifo.c	1403
linux/net/sched/sch_fifo.c	1404
linux/net/sched/sch_fifo.c	1405
linux/net/sched/estimator.c	1406
linux/net/sched/estimator.c	1407
linux/net/sched/estimator.c	1408
linux/net/sched/estimator.c	1409
linux/net/sched/estimator.c	1410
linux/net/sched/police.c	1411
linux/net/sched/police.c	1412
linux/net/sched/police.c	1413
linux/net/sched/police.c	1414
linux/net/sched/police.c	1415
linux/net/sched/police.c	1416
linux/net/sched/police.c	1417
linux/net/sched/sch_ingress.c	1418
linux/net/sched/sch_ingress.c	1419
linux/net/sched/sch_ingress.c	1420
linux/net/sched/sch_ingress.c	1421
linux/net/sched/sch_ingress.c	1422
linux/net/sched/sch_ingress.c	1423

linux/net/sched/sch_ingress.c	1424
linux/net/sched/sch_ingress.c	1425
linux/net/sched/sch_dsmark.c	1426
linux/net/sched/sch_dsmark.c	1427
linux/net/sched/sch_dsmark.c	1428
linux/net/sched/sch_dsmark.c	1429
linux/net/sched/sch_dsmark.c	1430
linux/net/sched/sch_dsmark.c	1431
linux/net/sched/sch_dsmark.c	1432
linux/net/sched/sch_dsmark.c	1433
linux/net/sched/sch_dsmark.c	1434
linux/net/sched/sch_dsmark.c	1435
linux/net/sched/sch_dsmark.c	1436
linux/net/sched/sch_dsmark.c	1437
linux/net/sched/sch_dsmark.c	1438
linux/net/sched/sch_dsmark.c	1439
linux/net/sched/sch_dsmark.c	1440
linux/net/sched/sch_dsmark.c	1441
linux/net/sched/sch_dsmark.c	1442
linux/net/sched/sch_red.c	1443
linux/net/sched/sch_red.c	1444
linux/net/sched/sch_red.c	1445
linux/net/sched/sch_red.c	1446
linux/net/sched/sch_red.c	1447
linux/net/sched/sch_red.c	1448
linux/net/sched/sch_red.c	1449
linux/net/sched/sch_red.c	1450
linux/net/sched/sch_red.c	1451
linux/net/sched/sch_red.c	1452
linux/net/sched/sch_red.c	1453
linux/net/sched/sch_red.c	1454
linux/net/sched/sch_red.c	1455
linux/net/sched/sch_red.c	1456
linux/net/sched/sch_red.c	1457
linux/net/sched/sch_tbf.c	1458
linux/net/sched/sch_tbf.c	1459
linux/net/sched/sch_tbf.c	1460
linux/net/sched/sch_tbf.c	1461
linux/net/sched/sch_tbf.c	1462
linux/net/sched/sch_tbf.c	1463
linux/net/sched/sch_tbf.c	1464
linux/net/sched/sch_tbf.c	1465
linux/net/sched/sch_tbf.c	1466
linux/net/sched/sch_tbf.c	1467
linux/net/sched/sch_tbf.c	1468
linux/net/sched/sch_tbf.c	1469
linux/net/sched/sch_ingress.c	1470
linux/net/sched/sch_ingress.c	1471
linux/net/sched/sch_ingress.c	1472
linux/net/sched/sch_prio.c	1473

linux/net/sched/sch_prio.c	1474
linux/net/sched/sch_prio.c	1475
linux/net/sched/sch_prio.c	1476
linux/net/sched/sch_prio.c	1477
linux/net/sched/sch_prio.c	1478
linux/net/sched/sch_sfq.c	1479
linux/net/sched/sch_sfq.c	1480
linux/net/sched/sch_prio.c	1481
linux/net/sched/sch_sfq.c	1482
linux/net/sched/sch_sfq.c	1483
linux/net/sched/sch_sfq.c	1484
linux/net/sched/sch_sfq.c	1485
linux/net/sched/sch_sfq.c	1486
linux/net/sched/sch_sfq.c	1487
linux/net/sched/sch_sfq.c	1488
linux/net/sched/sch_sfq.c	1489
linux/net/sched/sch_sfq.c	1490
linux/net/sched/sch_sfq.c	1491
linux/net/sched/sch_teql.c	1492
linux/net/sched/sch_teql.c	1493
linux/net/sched/sch_teql.c	1494
linux/net/sched/sch_teql.c	1495
linux/net/sched/sch_teql.c	1496
linux/net/sched/sch_teql.c	1497
linux/net/sched/sch_teql.c	1498
linux/net/sched/sch_teql.c	1499
linux/net/sched/sch_teql.c	1500
linux/net/sched/sch_teql.c	1501
linux/net/sched/sch_teql.c	1502
linux/net/sched/sch_teql.c	1503
linux/net/sched/sch_teql.c	1504
linux/net/sched/sch_teql.c	1505
linux/net/core/profile.c	1506
linux/net/core/profile.c	1507
linux/net/core/profile.c	1508
linux/net/core/profile.c	1509
linux/net/core/profile.c	1510
linux/net/sched/sch_atm.c	1511
linux/net/sched/sch_atm.c	1512
linux/net/sched/sch_atm.c	1513
linux/net/sched/sch_atm.c	1514
linux/net/sched/sch_atm.c	1515
linux/net/sched/sch_atm.c	1516
linux/net/sched/sch_atm.c	1517
linux/net/sched/sch_atm.c	1518
linux/net/sched/sch_atm.c	1519
linux/net/sched/sch_atm.c	1520
linux/net/sched/sch_atm.c	1521
linux/net/sched/sch_atm.c	1522
linux/net/sched/sch_atm.c	1523

linux/net/sched/sch_atm.c	1524
linux/net/sched/sch_atm.c	1525
linux/net/sched/sch_atm.c	1526
linux/net/sched/sch_atm.c	1527
linux/net/sched/sch_atm.c	1528
linux/net/sched/sch_atm.c	1529
linux/net/sched/sch_atm.c	1530
linux/net/sched/sch_atm.c	1531
linux/net/sched/sch_atm.c	1532
linux/net/sched/sch_atm.c	1533
tc/tc.c	1534

List of Figures

```
/* WONDERFUL. NO COMMENTS. --ANK */
```

Figure 1: Comments, what comments: `linux/net/khttpd/waiheaders.c`

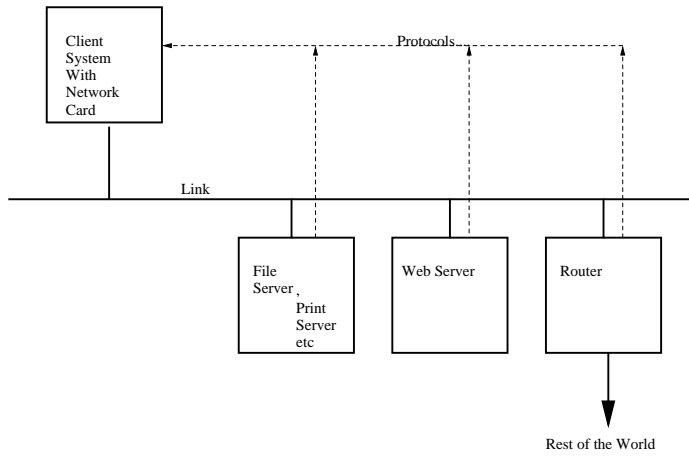


Figure 2: LAN

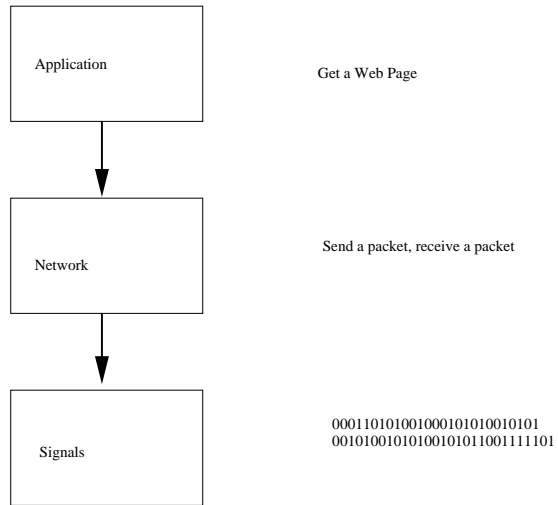


Figure 3: Layers of Abstraction

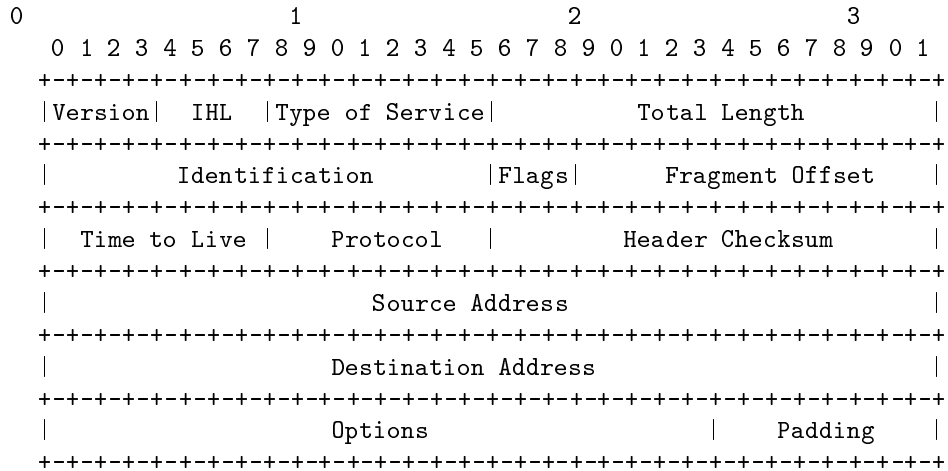


Figure 4: The RFC style IP Header Definition

```

116 struct iphdr {
117     #if defined(_LITTLE_ENDIAN_BITFIELD)
118         __u8    ihl:4,
119                version:4;
120     #elif defined (_BIG_ENDIAN_BITFIELD)
121         __u8    version:4,
122                ihl:4;
123     #else
124     #error "Please fix <asm/byteorder.h>"
125     #endif
126         __u8    tos;
127         __u16   tot_len;
128         __u16   id;
129         __u16   frag_off;
130         __u8    ttl;
131         __u8    protocol;
132         __u16   check;
133         __u32   saddr;
134         __u32   daddr;
135         /*The options start here. */
136 };

```

Figure 5: The C style IP Header Definition

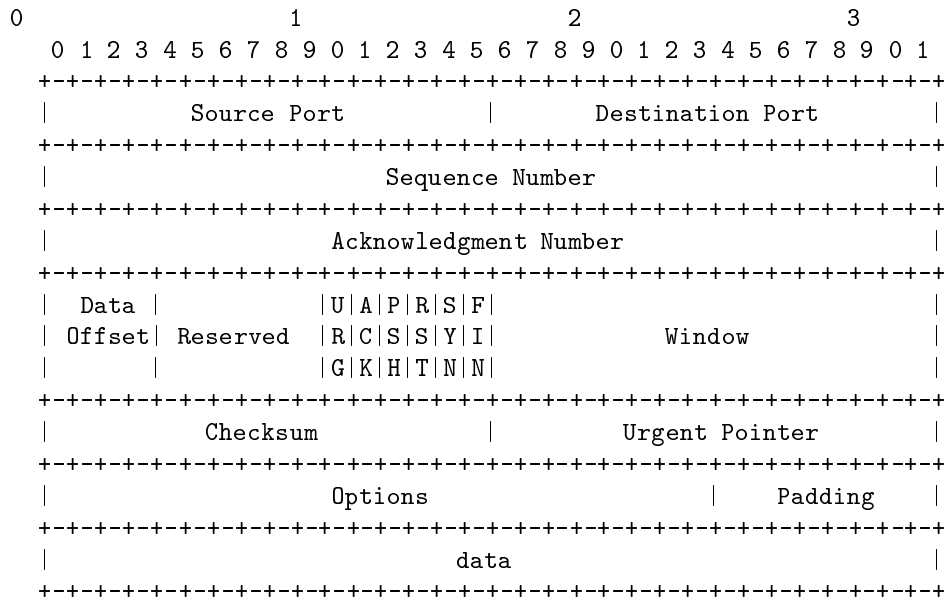


Figure 6: The RFC style IP Header Definition

```

23 struct tcphdr {
24     _u16  source;
25     _u16  dest;
26     _u32  seq;
27     _u32  ack_seq;
28     #if defined(_LITTLE_ENDIAN_BITFIELD)
29         _u16  res1:4,
30             doff:4,
31             fin:1,
32             syn:1,
33             rst:1,
34             psh:1,
35             ack:1,
36             urg:1,
37             ece:1,
38             cwr:1;
39     #elif defined(_BIG_ENDIAN_BITFIELD)
40         _u16  doff:4,
41             res1:4,
42             cwr:1,
43             ece:1,
44             urg:1,
45             ack:1,
46             psh:1,
47             rst:1,
48             syn:1,
49             fin:1;
50     #else
51     #error "Adjust your <asm/byteorder.h> defines"
52     #endif
53     _u16  window;
54     _u16  check;
55     _u16  urg_ptr;
56 };

```

Figure 7: The TCP Header Definition: `linux/include/linux/tcp.h`

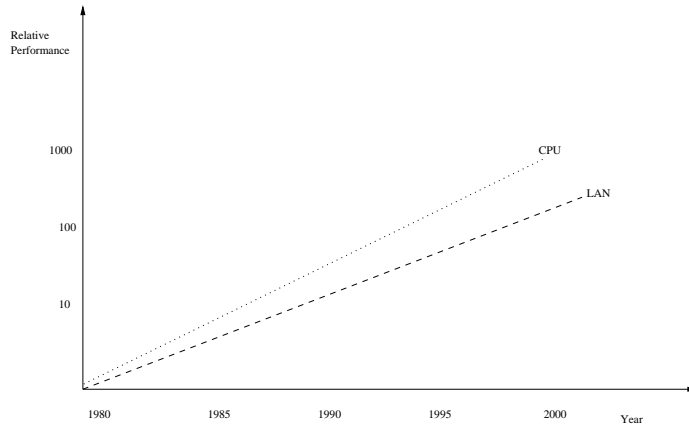


Figure 8: trends

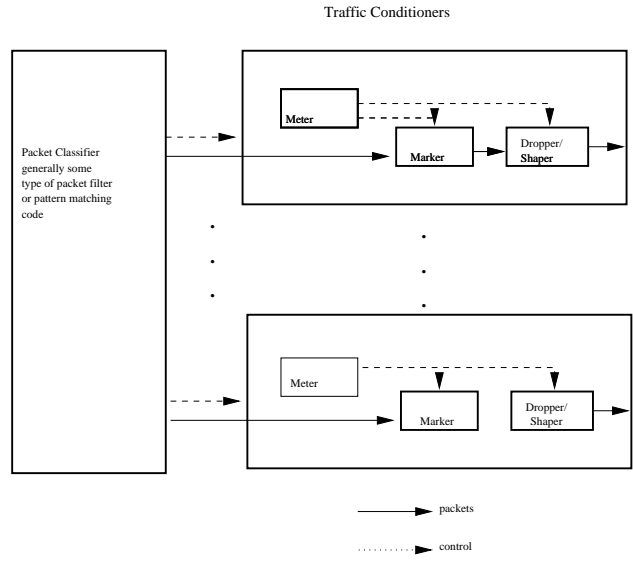


Figure 9: Differentiated Services Metering and Marking


```

6  /*
7  * This allows for 1024 file descriptors: if NR_OPEN is ever grown
8  * beyond that you'll have to change this too. But 1024 fd's seem to be
9  * enough even for such "real" unices like OSF/1, so hopefully this is
10 * one limit that doesn't have to be changed [again].
11 *
12 * Note that POSIX wants the FD_CLEAR(fd,fdsetp) defines to be in
13 * <sys/time.h> (and thus <linux/time.h>) - but this is a more logical
14 * place for them. Solved by having dummy defines in <sys/time.h>.
15 */
16
17 /*
18 * Those macros may have been defined in <gnu/types.h>. But we always
19 * use the ones here.
20 */
21 #undef __NFDBITS
22 #define __NFDBITS      (8 * sizeof(unsigned long))
23
24 #undef __FD_SETSIZE
25 #define __FD_SETSIZE 1024
26
27 #undef __FDSET_LONGS
28 #define __FDSET_LONGS (__FD_SETSIZE/ __NFDBITS)
29
30 #undef __FDELT
31 #define __FDELT(d)    ((d) / __NFDBITS)
32
33 #undef __FDMASK
34 #define __FDMASK(d)  (1UL << ((d) % __NFDBITS))
35
36 typedef struct {
37     unsigned long fds_bits [ __FDSET_LONGS];
38 } __kernel_fd_set ;
39
40 /* Type of a signal handler. */
41 typedef void (* __kernel_sighandler_t )(int);
42
43 /* Type of a SYSV IPC key. */
44 typedef int __kernel_key_t ;

```

Figure 10: POSIX C Type Definitions: `linux/include/linux/posix_types.h`

```

4 typedef unsigned short umode_t;
5
6 /*
7  * __xx is ok: it doesn't pollute the POSIX namespace. Use these in the
8  * header files exported to user space
9  */
10
11 typedef __signed__ char __s8;
12 typedef unsigned char __u8;
13
14 typedef __signed__ short __s16;
15 typedef unsigned short __u16;
16
17 typedef __signed__ int __s32;
18 typedef unsigned int __u32;
19
20 #if defined(__GNUC__) && !defined(__STRICT_ANSI__)
21 typedef __signed__ long long __s64;
22 typedef unsigned long long __u64;
23 #endif
24
25 /*
26  * These aren't exported outside the kernel to avoid name space clashes
27  */
28 #ifdef __KERNEL__
29
30 typedef signed char s8;
31 typedef unsigned char u8;
32
33 typedef signed short s16;
34 typedef unsigned short u16;
35
36 typedef signed int s32;
37 typedef unsigned int u32;
38
39 typedef signed long long s64;
40 typedef unsigned long long u64;
41
42 #define BITS_PER_LONG 32
43
44 /* Dma addresses are 32-bits wide. */
45
46 typedef u32 dma_addr_t;

```

Figure 11: i386 ASM header interface C Type Definitions:
 linux/include/asm/i386/types.h

```

13 typedef __kernel_fd_set      fd_set ;
14 typedef __kernel_dev_t      dev_t ;
15 typedef __kernel_ino_t      ino_t ;
16 typedef __kernel_mode_t     mode_t;
17 typedef __kernel_nlink_t    nlink_t;
18 typedef __kernel_off_t      off_t ;
19 typedef __kernel_pid_t      pid_t ;
20 typedef __kernel_daddr_t    daddr_t;
21 typedef __kernel_key_t      key_t ;
22 typedef __kernel_suseconds_t suseconds_t;
23
24 #ifdef __KERNEL__
25 typedef __kernel_uid32_t     uid_t ;
26 typedef __kernel_gid32_t     gid_t ;
27 typedef __kernel_uid16_t     uid16_t;
28 typedef __kernel_gid16_t     gid16_t;
29
30 #ifdef CONFIG_UID16
31 /* This is defined by include/asm-{arch}/posix_types.h */
32 typedef __kernel_old_uid_t   old_uid_t ;
33 typedef __kernel_old_gid_t   old_gid_t ;
34 #endif /* CONFIG_UID16 */
35
36 /* libc5 includes this file to define uid_t, thus uid_t can never change
37  * when it is included by non-kernel code
38  */
39 #else
40 typedef __kernel_uid_t       uid_t ;
41 typedef __kernel_gid_t       gid_t ;
42 #endif /* __KERNEL__ */
43
44 #if defined(__GNUC__) && !defined(_STRICT_ANSI_)
45 typedef __kernel_loff_t      loff_t ;
46 #endif
47
48 /*
49  * The following typedefs are also protected by individual ifdefs for
50  * historical reasons:
51  */
52 #ifndef _SIZE_T
53 #define _SIZE_T
54 typedef __kernel_size_t      size_t ;
55 #endif
56
57 #ifndef _SSIZE_T
58 #define _SSIZE_T
59 typedef __kernel_ssize_t     ssize_t ;
60 #endif

```

Figure 12: Kernel C Type Definitions: `linux/include/linux/types.h`

```

62 #ifndef _PTRDIFF_T
63 #define _PTRDIFF_T
64 typedef __kernel_ptrdiff_t    ptrdiff_t ;
65 #endif
66
67 #ifndef _TIME_T
68 #define _TIME_T
69 typedef __kernel_time_t      time_t;
70 #endif
71
72 #ifndef _CLOCK_T
73 #define _CLOCK_T
74 typedef __kernel_clock_t     clock_t;
75 #endif
76
77 #ifndef _CADDR_T
78 #define _CADDR_T
79 typedef __kernel_caddr_t     caddr_t;
80 #endif
81
82 /* bsd */
83 typedef unsigned char        u_char;
84 typedef unsigned short       u_short;
85 typedef unsigned int         u_int ;
86 typedef unsigned long        u_long;
87
88 /* sysv */
89 typedef unsigned char        unchar;
90 typedef unsigned short       ushort;
91 typedef unsigned int         uint ;
92 typedef unsigned long        ulong;

```

Figure 13: Kernel C Type Definitions: `linux/include/linux/types.h`

```

94 #ifndef __BIT_TYPES_DEFINED__
95 #define __BIT_TYPES_DEFINED__
96
97 typedef      __u8      u_int8_t ;
98 typedef      __s8      int8_t ;
99 typedef      __u16     u_int16_t ;
100 typedef      __s16     int16_t ;
101 typedef      __u32     u_int32_t ;
102 typedef      __s32     int32_t ;
103
104 #endif /* !(__BIT_TYPES_DEFINED__) */
105
106 typedef      __u8      uint8_t ;
107 typedef      __u16     uint16_t ;
108 typedef      __u32     uint32_t ;
109
110 #if defined(__GNUC__) && !defined(__STRICT_ANSI__)
111 typedef      __u64     uint64_t ;
112 typedef      __u64     u_int64_t ;
113 typedef      __s64     int64_t ;
114 #endif

```

Figure 14: Kernel C Type Definitions: `linux/include/linux/types.h`

Context Switch Between User Space and Kernel, and between Interrupt and Kernel)

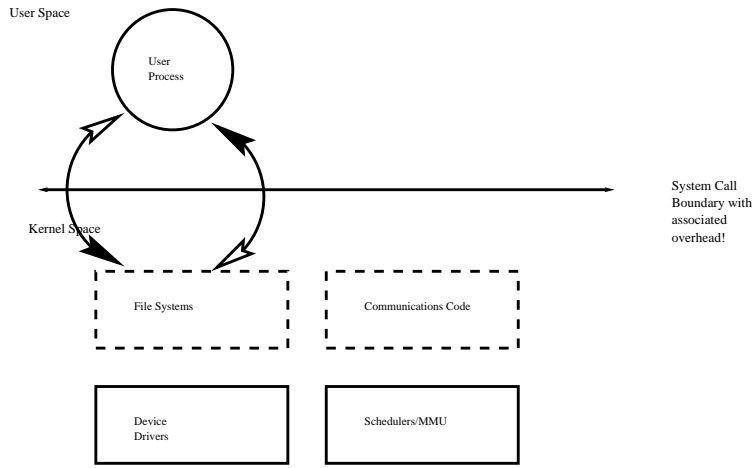


Figure 15: System Calls

```

490     */
491 needs_resched:
492     {
493         unsigned long flags;
494
495         /*
496          * Avoid taking the runqueue lock in cases where
497          * no preemption-check is necessary:
498          */
499         if ((prev == idle_task(smp_processor_id())) ||
500             (policy & SCHED_YIELD))
501             goto out_unlock;
502
503         spin_lock_irqsave(&runqueue_lock, flags);
504         if ((prev->state == TASK_RUNNING) && !prev->has_cpu)
505             reschedule_idle(prev);
506         spin_unlock_irqrestore(&runqueue_lock, flags);
507         goto out_unlock;
508     }
509 #else
510     prev->policy &= ~SCHED_YIELD;
511 #endif /* CONFIG_SMP */
512 }
513
514 void schedule_tail(struct task_struct *prev)
515 {
516     _schedule_tail(prev);
517 }
518
519 /*
520  * 'schedule()' is the scheduler function. It's a very simple and nice
521  * scheduler: it's not perfect, but certainly works for most things.
522  */

```

Figure 16: The Scheduler i: linux/kernel/sched.c

```
523 * The goto is "interesting".
524 *
525 * NOTE!! Task 0 is the 'idle' task, which gets called when no other
526 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
527 * information in task[0] is never used.
528 */
529 asmlinkage void schedule(void)
530 {
531     struct schedule_data * sched_data;
532     struct task_struct *prev, *next, *p;
533     struct list_head *tmp;
534     int this_cpu, c;
535
536     if (!current->active_mm) BUG();
537 need_resched_back:
```

Figure 17: The Scheduler ii: `linux/kernel/sched.c`


```

538     prev = current;
539     this_cpu = prev->processor;
540
541     if ( in_interrupt ()
542         goto scheduling_in_interrupt ;
543
544     release_kernel_lock (prev, this_cpu);
545
546     /* Do "administrative" work here while we don't hold any locks */
547     if ( softirq_active (this_cpu) & softirq_mask(this_cpu))
548         goto handle_softirq;
549 handle_softirq_back:
550
551     /*
552     * 'sched_data' is protected by the fact that we can run
553     * only one process per CPU.
554     */
555     sched_data = & aligned_data[this_cpu].schedule_data;
556
557     spin_lock_irq (&runqueue_lock);
558
559     /* move an exhausted RR process to be last.. */
560     if (prev->policy == SCHED_RR)
561         goto move_rr_last;
562 move_rr_back:
563
564     switch (prev->state) {
565     case TASK_INTERRUPTIBLE:
566         if (signal_pending(prev)) {
567             prev->state = TASK_RUNNING;
568             break;
569         }
570     default:
571         del_from_runqueue(prev);

```

Figure 18: The Scheduler iii: linux/kernel/sched.c

```

572         case TASK_RUNNING;
573     }
574     prev->need_resched = 0;
575
576     /*
577     * this is the scheduler proper:
578     */
579
580 repeat_schedule:
581     /*
582     * Default process to select..
583     */
584     next = idle_task(this_cpu);
585     c = -1000;
586     if (prev->state == TASK_RUNNING)
587         goto still_running;
588
589 still_running_back :
590     list_for_each (tmp, &runqueue_head) {
591         p = list_entry (tmp, struct task_struct, run_list);
592         if (can_schedule(p, this_cpu)) {
593             int weight = goodness(p, this_cpu, prev->active_mm);
594             if (weight > c)
595                 c = weight, next = p;
596         }
597     }
598
599     /* Do we need to re-calculate counters? */
600     if (!c)
601         goto recalculate;
602
603     /*
604     * from this point on nothing can prevent us from
605     * switching to the next task, save this fact in
606     * sched_data.
607     */
608     sched_data->curr = next;
609 #ifdef CONFIG_SMP
610     next->has_cpu = 1;
611     next->processor = this_cpu;
612 #endif
613     spin_unlock_irq(&runqueue_lock);

```

Figure 19: The Scheduler iv: linux/kernel/sched.c

```

614     if (prev == next)
615         goto same_process;
616
617 #ifdef CONFIG_SMP
618     /*
619      * maintain the per-process 'last schedule' value.
620      * (this has to be recalculated even if we reschedule to
621      * the same process) Currently this is only used on SMP,
622      * and it's approximate, so we do not have to maintain
623      * it while holding the runqueue spinlock.
624      */
625     sched_data->last_schedule = get_cycles();
626
627     /*
628      * We drop the scheduler lock early (it's a global spinlock),
629      * thus we have to lock the previous process from getting
630      * rescheduled during switch_to().
631      */
632
633 #endif /* CONFIG_SMP */
634
635     kstat.context_switch++;
636     /*
637      * there are 3 processes which are affected by a context switch:
638      *
639      * prev == ... ==> (last => next)
640      *
641      * It's the 'much more previous' 'prev' that is on next's stack,
642      * but prev is set to (the just run) 'last' process by switch_to().
643      * This might sound slightly confusing but makes tons of sense.
644      */
645     prepare_to_switch();
646     {
647         struct mm_struct *mm = next->mm;
648         struct mm_struct *oldmm = prev->active_mm;
649         if (!mm) {
650             if (next->active_mm) BUG();
651             next->active_mm = oldmm;

```

Figure 20: The Scheduler iv: linux/kernel/sched.c

```

652         atomic_inc(&oldmm->mm_count);
653         enter_lazy_tlb (oldmm, next, this_cpu);
654     } else {
655         if (next->active_mm != mm) BUG();
656         switch_mm(oldmm, mm, next, this_cpu);
657     }
658
659     if (!prev->mm) {
660         prev->active_mm = NULL;
661         mmdrop(oldmm);
662     }
663 }
664
665 /*
666  * This just switches the register state and the
667  * stack.
668  */
669 switch_to(prev, next, prev);
670 _schedule_tail (prev);
671
672 same_process:
673     reacquire_kernel_lock (current);
674     if (current->need_resched)
675         goto need_resched_back;
676
677     return;
678
679 recalculate :
680     {
681         struct task_struct *p;
682         spin_unlock_irq (&runqueue_lock);
683         read_lock (&tasklist_lock);
684         for_each_task (p)
685             p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice)
686             ;
687         read_unlock (&tasklist_lock);
688         spin_lock_irq (&runqueue_lock);
689     }
690     goto repeat_schedule;
691
692 still_running :
693     c = goodness(prev, this_cpu, prev->active_mm);
694     next = prev;
695     goto still_running_back ;
696
697 handle_softirq :
698     do_softirq ();
699     goto handle_softirq_back ;
700
701 move_rr_last :

```

Figure 21: The Scheduler v: linux/kernel/sched.c

```

446  */
447  static inline void __schedule_tail (struct task_struct *prev)
448  {
449  #ifdef CONFIG_SMP
450      int policy;
451
452      /*
453       * prev->policy can be written from here only before prev'
454       * can be scheduled (before setting prev->has_cpu to zero).
455       * Of course it must also be read before allowing prev
456       * to be rescheduled, but since the write depends on the read
457       * to complete, wmb() is enough. (the spin_lock() acquired
458       * before setting has_cpu is not enough because the spin_lock()
459       * common code semantics allows code outside the critical section
460       * to enter inside the critical section)
461       */
462      policy = prev->policy;
463      prev->policy = policy & ~SCHED_YIELD;
464      wmb();
465
466      /*
467       * fast path falls through. We have to clear has_cpu before
468       * checking prev->state to avoid a wakeup race - thus we
469       * also have to protect against the task exiting early.
470       */
471      task_lock(prev);
472      prev->has_cpu = 0;
473      mb();
474      if (prev->state == TASK_RUNNING)
475          goto needs_resched;
476
477  out_unlock:
478      task_unlock(prev);      /* Synchronise here with release_task() if prev is
479                               TASK_ZOMBIE */
480      return;
481
482      /*
483       * Slow path - we 'push' the previous process and
484       * reschedule_idle() will attempt to find a new

```

Figure 22: The Scheduler Tail: linux/kernel/sched.c

```

346     success = 1;
347 out:
348     spin_unlock_irqrestore(&runqueue_lock, flags);
349     return success;
350 }
351
352 inline int wake_up_process(struct task_struct * p)
353 {
354     return try_to_wake_up(p, 0);
355 }
356
357 static void process_timeout(unsigned long _data)
358 {
359     struct task_struct * p = (struct task_struct *) _data;
360
361     wake_up_process(p);
362 }
363
364 /**
365  * schedule_timeout - sleep until timeout
366  * @timeout: timeout value in jiffies
367  *
368  * Make the current task sleep until @timeout jiffies have
369  * elapsed. The routine will return immediately unless

```

Figure 23: The Wakeup Common Code : linux/kernel/sched.c

```
803     return timeout;
804 }
805
806 void sleep_on(wait_queue_head_t *q)
807 {
808     SLEEP_ON_VAR
809     current->state = TASK_UNINTERRUPTIBLE;
810
811
```

Figure 24: The Sleepon Code : `linux/kernel/sched.c`

```

86 #define SAVE_ALL \
87     cld; \
88     pushl %es; \
89     pushl %ds; \
90     pushl %eax; \
91     pushl %ebp; \
92     pushl %edi; \
93     pushl %esi; \
94     pushl %edx; \
95     pushl %ecx; \
96     pushl %ebx; \
97     movl $(_KERNEL_DS),%edx; \
98     movl %edx,%ds; \
99     movl %edx,%es;
100
101 #define RESTORE_ALL \
102     popl %ebx; \
103     popl %ecx; \
104     popl %edx; \
105     popl %esi; \
106     popl %edi; \
107     popl %ebp; \
108     popl %eax; \
109 1:  popl %ds; \
110 2:  popl %es; \
111     addl $4,%esp; \
112 3:  iret ; \
113 .section .fixup,"ax"; \
114 4:  movl $0,(%esp); \
115     jmp 1b; \
116 5:  movl $0,(%esp); \
117     jmp 2b; \
118 6:  pushl %ss; \
119     popl %ds; \
120     pushl %ss; \
121     popl %es; \
122     pushl $11; \
123     call do_exit; \
124 .previous; \
125 .section __ex_table,"a"; \
126     .align 4; \
127     .long 1b,4b; \
128     .long 2b,5b; \
129     .long 3b,6b; \
130 .previous

```

Figure 25: The Linux Kernel System Call Entry Code -save and restore overhead: `linux/arch/i386/kernel/entry.S`


```

136 ENTRY(lcall7)
137     pushfl                # We get a different stack layout with call gates,
138     pushl %eax           # which has to be cleaned up later..
139     SAVE_ALL
140     movl EIP(%esp),%eax  # due to call gates, this is eflags , not eip ..
141     movl CS(%esp),%edx   # this is eip ..
142     movl EFLAGS(%esp),%ecx # and this is cs..
143     movl %eax,EFLAGS(%esp) #
144     movl %edx,EIP(%esp) # Now we move them to their "normal" places
145     movl %ecx,CS(%esp)  #
146     movl %esp,%ebx
147     pushl %ebx
148     andl $-8192,%ebx    # GET_CURRENT
149     movl exec_domain(%ebx),%edx # Get the execution domain
150     movl 4(%edx),%edx   # Get the lcall7 handler for the domain
151     pushl $0x7
152     call *%edx
153     addl $4, %esp
154     popl %eax
155     jmp ret_from_sys_call

```

Figure 26: The Linux Kernel System Call Entry Code - example of entry: `linux/arch/i386/kernel/entry.S`

```

498 static void __init smp_init(void)
499 {
500     /* Get other processors into their bootup holding patterns. */
501     smp_boot_cpus();
502     smp_threads_ready=1;
503     smp_commence();
504 }
505
506 #endif
507
508 /*
509  *   Activate the first processor.
510  */
511
512 asmlinkage void __init start_kernel(void)
513 {
514     char * command_line;
515     unsigned long mempages;
516     extern char saved_command_line[];
517
518     /*
519      * Interrupts are still disabled. Do necessary setups, then
520      * enable them
521      */
521     lock_kernel();
522     printk(linux_banner);

```

Figure 27: The Linux Kernel Startup Code i: `linux/init/main.c`

```

523     setup_arch(&command_line);
524     printk("Kernel command line: %s\n", saved_command_line);
525     parse_options(command_line);
526     trap_init ();
527     init_IRQ();
528     sched_init ();
529     time_init ();
530     softirq_init ();
531
532     /*
533     * HACK ALERT! This is early. We're enabling the console before
534     * we've done PCI setups etc, and console_init() must be aware of
535     * this. But we do want output early, in case something goes wrong.
536     */
537     console_init ();
538 #ifdef CONFIG_MODULES
539     init_modules();
540 #endif
541     if ( prof_shift ) {
542         unsigned int size;
543         /* only text is profiled */
544         prof_len = (unsigned long) &etext - (unsigned long) &stext;
545         prof_len >>= prof_shift;
546
547         size = prof_len * sizeof(unsigned int) + PAGE_SIZE-1;
548         prof_buffer = (unsigned int *) alloc_bootmem(size);
549     }
550
551     kmem_cache_init();
552     sti ();
553     calibrate_delay ();
554 #ifdef CONFIG_BLK_DEV_INITRD
555     if ( initrd_start && !initrd_below_start_ok &&
556         initrd_start < min_low_pfn << PAGE_SHIFT) {
557         printk(KERN_CRIT "initrd overwritten (0x%08lx-<<0x%08lx)->"
558             "disabling it.\n", initrd_start ,min_low_pfn << PAGE_SHIFT);
559         initrd_start = 0;
560     }
561 #endif
562     mem_init();

```

Figure 28: The Linux Kernel Startup Code ii: linux/init/main.c

```

563     kmem_cache_sizes_init ();
564 #ifdef CONFIG_PROC_FS
565     proc_root_init ();
566 #endif
567     mempages = num_physpages;
568
569     fork_init (mempages);
570     proc_caches_init ();
571     vfs_caches_init (mempages);
572     buffer_init (mempages);
573     page_cache_init (mempages);
574 #if defined(CONFIG_ARCH_S390)
575     ccwcache_init ();
576 #endif
577     signals_init ();
578     bdev_init ();
579     inode_init (mempages);
580 #if defined(CONFIG_SYSVIPC)
581     ipc_init ();
582 #endif
583 #if defined(CONFIG_QUOTA)
584     dquot_init_hash ();
585 #endif
586     check_bugs();
587     printk("POSIX_conformance_testing_by_UNIFIX\n");
588
589     /*
590     *   We count on the initial thread going ok
591     *   Like idlers init is an unlocked kernel thread, which will
592     *   make syscalls (and thus be locked).

```

Figure 29: The Linux Kernel Startup Code iii: `linux/init/main.c`

```

100     if (major == 0) {
101         write_lock(&chrdevs_lock);
102         for (major = MAX_CHRDEV-1; major > 0; major--) {
103             if (chrdevs[major].fops == NULL) {
104                 chrdevs[major].name = name;
105                 chrdevs[major].fops = fops;
106                 write_unlock(&chrdevs_lock);
107                 return major;
108             }
109         }
110         write_unlock(&chrdevs_lock);
111         return -EBUSY;
112     }
113     if (major >= MAX_CHRDEV)
114         return -EINVAL;
115     write_lock(&chrdevs_lock);
116     if (chrdevs[major].fops && chrdevs[major].fops != fops) {
117         write_unlock(&chrdevs_lock);
118         return -EBUSY;
119     }
120     chrdevs[major].name = name;
121     chrdevs[major].fops = fops;
122     write_unlock(&chrdevs_lock);
123     return 0;
124 }
125
126 int unregister_chrdev(unsigned int major, const char * name)
127 {
128     if (major >= MAX_CHRDEV)
129         return -EINVAL;
130     write_lock(&chrdevs_lock);
131     if (!chrdevs[major].fops || strcmp(chrdevs[major].name, name)) {
132         write_unlock(&chrdevs_lock);
133         return -EINVAL;
134     }
135     chrdevs[major].name = NULL;
136     chrdevs[major].fops = NULL;
137     write_unlock(&chrdevs_lock);
138     return 0;
139 }
140
141 /*

```

Figure 30: Linux Character Device Registration: `linux/fs/devices.c`

```

478     len = sprintf(p, "\nBlock devices:\n");
479     for (i = 0; i < MAX_BLKDEV; i++) {
480         if (blkdevs[i].bdops) {
481             len += sprintf(p+len, "%3d_%s\n", i, blkdevs[i].name);
482         }
483     }
484     return len;
485 }
486
487
488 /*
489  Return the function table of a device.
490  Load the driver if needed.
491 */
492 const struct block_device_operations * get_blkfops(unsigned int major)
493 {
494     const struct block_device_operations *ret = NULL;
495
496     /* major 0 is used for non-device mounts */
497     if (major && major < MAX_BLKDEV) {
498 #ifdef CONFIG_KMOD
499         if (!blkdevs[major].bdops) {
500             char name[20];
501             sprintf(name, "block-major-%d", major);
502             request_module(name);
503         }
504 #endif
505         ret = blkdevs[major].bdops;
506     }
507     return ret;
508 }
509
510 int register_blkdev(unsigned int major, const char * name, struct block_device_operations
    *bdops)

```

Figure 31: Linux Block Device Registration : `linux/fs/block_dev.c`

```

2151         return -EEXIST;
2152         memcpy(dev->name, ifr->ifr_newname, IFNAMSIZ);
2153         dev->name[IFNAMSIZ-1] = 0;
2154         notifier_call_chain (&netdev_chain, NETDEV_CHANGENAME,
                               dev);
2155         return 0;
2156
2157     /*
2158     *   Unknown or private ioctl
2159     */
2160
2161     default:
2162         if ((cmd >= SIOCDEVPRIVATE &&
2163             cmd <= SIOCDEVPRIVATE + 15) ||
2164             cmd == SIOCETHTOOL) {
2165             if (dev->do_ioctl) {
2166                 if (!netif_device_present (dev))
2167                     return -ENODEV;
2168                 return dev->do_ioctl(dev, ifr, cmd);
2169             }
2170             return -EOPNOTSUPP;
2171         }
2172
2173     #ifdef WIRELESS_EXT
2174         if (cmd >= SIOCIWFIRST && cmd <= SIOCIWLAST) {
2175             if (dev->do_ioctl) {
2176                 if (!netif_device_present (dev))
2177                     return -ENODEV;
2178                 return dev->do_ioctl(dev, ifr, cmd);
2179             }
2180             return -EOPNOTSUPP;
2181         }
2182     #endif /* WIRELESS_EXT */
2183
2184     }
2185     return -EINVAL;
2186 }
2187
2188 /*
2189 *   This function handles all "interface"-type I/O control requests. The
2190 *   actual
2191 *   'doing' part of this is dev_ifsioc above.

```

Figure 32: Linux Network Device Registration - i: `linux/net/core/dev.c`

```

2192
2193 /**
2194  *   dev_ioctl      -      network device ioctl
2195  *   @cmd: command to issue
2196  *   @arg: pointer to a struct ifreq in user space
2197  *
2198  *   Issue ioctl functions to devices. This is normally called by the
2199  *   user space syscall interfaces but can sometimes be useful for
2200  *   other purposes. The return value is the return from the syscall if
2201  *   positive or a negative errno code on error.
2202  */
2203
2204 int dev_ioctl(unsigned int cmd, void *arg)
2205 {
2206     struct ifreq ifr;
2207     int ret;
2208     char *colon;
2209
2210     /* One special case: SIOCGIFCONF takes ifconf argument
2211        and requires shared lock, because it sleeps writing
2212        to user space.

```

Figure 33: Linux Network Device Registration - ii: linux/net/core/dev.c


```

2214
2215     if (cmd == SIOCGIFCONF) {
2216         rtnl_shlock();
2217         ret = dev_ifconf((char *) arg);
2218         rtnl_shunlock();
2219         return ret;
2220     }
2221     if (cmd == SIOCGIFNAME) {
2222         return dev_ifname((struct ifreq *)arg);
2223     }
2224
2225     if (copy_from_user(&ifr, arg, sizeof(struct ifreq)))
2226         return -EFAULT;
2227
2228     ifr.ifr_name[IFNAMSIZ-1] = 0;
2229
2230     colon = strchr(ifr.ifr_name, ':');
2231     if (colon)
2232         *colon = 0;
2233
2234     /*
2235      *   See which interface the caller is talking about.
2236      */
2237
2238     switch(cmd)
2239     {
2240         /*
2241          *   These ioctl calls:
2242          *   - can be done by all.
2243          *   - atomic and do not require locking.
2244          *   - return a value
2245          */
2246
2247         case SIOCGIFFLAGS:
2248         case SIOCGIFMETRIC:
2249         case SIOCGIFMTU:
2250         case SIOCGIFHWADDR:
2251         case SIOCGIFSLAVE:
2252         case SIOCGIFMAP:
2253         case SIOCGIFINDEX:
2254         case SIOCGIFTXQLEN:
2255             dev_load(ifr.ifr_name);
2256             read_lock(&dev_base_lock);
2257             ret = dev_ifsioc(&ifr, cmd);
2258             read_unlock(&dev_base_lock);
2259             if (!ret) {
2260                 if (colon)
2261                     *colon = ':' ;

```

Figure 34: Linux Network Device Registration - iii: `linux/net/core/dev.c`

```
2264     }
2265     return ret;
2266
2267     /*
2268     *   These ioctl calls:
2269     *   - require superuser power.
2270     *   - require strict serialization.
2271     *   - do not return a value
2272     */
2273
2274     case SIOCSIFFLAGS:
2275     case SIOCSIFMETRIC:
2276     case SIOCSIFMTU:
2277     case SIOCSIFMAP:
2278     case SIOCSIFHWADDR:
2279     case SIOCSIFSLAVE:
2280     case SIOCADDMULTI:
2281     case SIOCDELMULTI:
2282     case SIOCSIFHWBROADCAST:
2283     case SIOCSIFTXQLEN:
2284     case SIOCSIFNAME:
```

Figure 35: Linux Network Device Registration - iv: `linux/net/core/dev.c`

```

2285     case SIOCETHTOOL:
2286         if (!capable(CAP_NET_ADMIN))
2287             return -EPERM;
2288         dev_load(ifr.ifr_name);
2289         dev_probe_lock();
2290         rtnl_lock();
2291         ret = dev_ifsioc(&ifr, cmd);
2292         rtnl_unlock();
2293         dev_probe_unlock();
2294         return ret;
2295
2296     case SIOCGIFMEM:
2297         /* Get the per device memory space. We can add this but
2298            currently
2299            do not support it */
2300     case SIOCSIFMEM:
2301         /* Set the per device memory buffer space. Not applicable
2302            in our case */
2303     case SIOCSIFLINK:
2304         return -EINVAL;
2305
2306     /*
2307     *   Unknown or private ioctl.
2308     */
2309
2310     default:
2311         if (cmd >= SIOCDEVPRIVATE &&
2312             cmd <= SIOCDEVPRIVATE + 15) {
2313             dev_load(ifr.ifr_name);
2314             dev_probe_lock();
2315             rtnl_lock();
2316             ret = dev_ifsioc(&ifr, cmd);
2317             rtnl_unlock();
2318             dev_probe_unlock();

```

Figure 36: Linux Network Device Registration - v: linux/net/core/dev.c

```

2317         if (!ret && copy_to_user(arg, &ifr, sizeof(struct ifreq)))
2318             return -EFAULT;
2319         return ret;
2320     }
2321 #ifdef WIRELESS_EXT
2322     /* Take care of Wireless Extensions */
2323     if (cmd >= SIOCIWFIRST && cmd <= SIOCIWLAST) {
2324         /* If command is set a parameter', or
2325          * get the encoding parameters', check if
2326          * the user has the right to do it */
2327         if (IW_IS_SET(cmd) || (cmd == SIOCGIWENCOD)) {
2328             if (!capable(CAP_NET_ADMIN))
2329                 return -EPERM;
2330         }
2331         dev_load(ifr.ifr_name);
2332         rtnl_lock();
2333         ret = dev_ifsioc(&ifr, cmd);
2334         rtnl_unlock();
2335         if (!ret && IW_IS_GET(cmd) &&
2336             copy_to_user(arg, &ifr, sizeof(struct ifreq)))
2337             return -EFAULT;
2338         return ret;
2339     }
2340 #endif /* WIRELESS_EXT */
2341     return -EINVAL;
2342 }
2343 }
2344
2345
2346 /**
2347  * dev_new_index - allocate an ifindex
2348  *
2349  * Returns a suitable unique value for a new device interface
2350  * number. The caller must hold the rtnl semaphore or the
2351  * dev_base_lock to be sure it remains unique.
2352  */
2353
2354 int dev_new_index(void)
2355 {
2356     static int ifindex;
2357     for (;;) {

```

Figure 37: Linux Network Device Registration - 2: linux/net/core/dev.c

```

2358         if (++ifindex <= 0)
2359             ifindex=1;
2360         if (_dev_get_by_index(ifindex) == NULL)
2361             return ifindex;
2362     }
2363 }
2364
2365 static int dev_boot_phase = 1;
2366
2367 /**
2368  *   register_netdevice   - register a network device
2369  *   @dev: device to register
2370  *
2371  *   Take a completed network device structure and add it to the kernel
2372  *   interfaces. A %NETDEV_REGISTER message is sent to the netdev notifier
2373  *   chain. 0 is returned on success. A negative errno code is returned
2374  *   on a failure to set up the device, or if the name is a duplicate.
2375  *
2376  *   Callers must hold the rtnl semaphore. See the comment at the
2377  *   end of Space.c for details about the locking. You may want
2378  *   register_netdev() instead of this.
2379  *
2380  *   BUGS:
2381  *   The locking appears insufficient to guarantee two parallel registers
2382  *   will not get the same name.
2383  */
2384
2385 int register_netdevice (struct net_device *dev)
2386 {
2387     struct net_device *d, **dp;

```

Figure 38: Linux Network Device Registration - 2: linux/net/core/dev.c

```

2389     int ret;
2390 #endif
2391
2392     spin_lock_init (&dev->queue_lock);
2393     spin_lock_init (&dev->xmit_lock);
2394     dev->xmit_lock_owner = -1;
2395 #ifdef CONFIG_NET_FASTROUTE
2396     dev->fastpath_lock=RW_LOCK_UNLOCKED;
2397 #endif
2398
2399     if (dev_boot_phase) {
2400 #ifdef CONFIG_NET_DIVERT
2401         ret = alloc_divert_blk (dev);
2402         if (ret)
2403             return ret;
2404 #endif /* CONFIG_NET_DIVERT */
2405
2406         /* This is NOT bug, but I am not sure, that all the
2407            devices, initialized before netdev module is started
2408            are sane.
2409
2410            Now they are chained to device boot list

```

Figure 39: Linux Network Device Registration - 2: linux/net/core/dev.c

Design Choice: Monolithic Kernel, Micro-kernel, or hybrid (Linux with modules!)

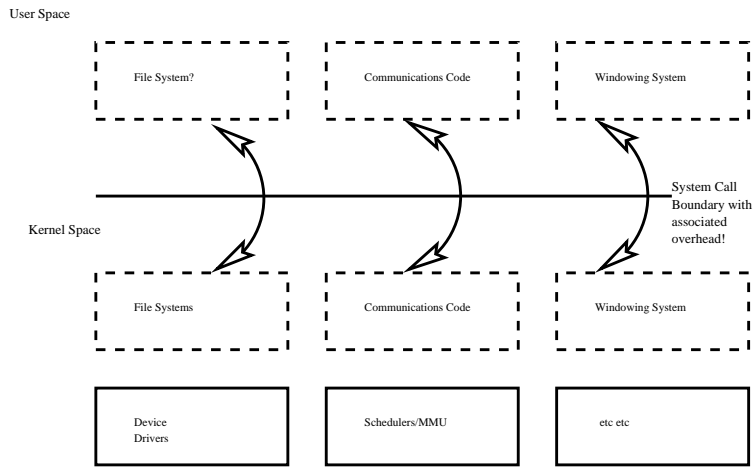


Figure 40: Monolithic, Micro-kernel or Hybrid Kernel?

```

474 } __attribute__((__aligned__(SMP_CACHE_BYTES)));
475
476
477 extern struct softnet_data softnet_data[NR_CPUS];
478
479 #define HAVE_NETIF_QUEUE
480
481 static inline void __netif_schedule(struct net_device *dev)
482 {
483     if (!test_and_set_bit(__LINK_STATE_SCHED, &dev->state)) {
484         unsigned long flags;
485         int cpu = smp_processor_id();
486
487         local_irq_save(flags);
488         dev->next_sched = softnet_data[cpu].output_queue;
489         softnet_data[cpu].output_queue = dev;
490         __cpu_raise_softirq(cpu, NET_TX_SOFTIRQ);
491         local_irq_restore(flags);
492     }
493 }
494
495 static inline void netif_schedule(struct net_device *dev)
496 {
497     if (!test_bit(__LINK_STATE_XOFF, &dev->state))
498         __netif_schedule(dev);
499 }
500
501 static inline void netif_start_queue(struct net_device *dev)
502 {
503     clear_bit(__LINK_STATE_XOFF, &dev->state);
504 }
505
506 static inline void netif_wake_queue(struct net_device *dev)
507 {
508     if (test_and_clear_bit(__LINK_STATE_XOFF, &dev->state))
509         __netif_schedule(dev);
510 }
511
512 static inline void netif_stop_queue(struct net_device *dev)
513 {
514     set_bit(__LINK_STATE_XOFF, &dev->state);
515 }
516
517 static inline int netif_queue_stopped(struct net_device *dev)
518 {
519     return test_bit(__LINK_STATE_XOFF, &dev->state);
520 }
521
522 static inline int netif_running(struct net_device *dev)

```

Figure 41: SMP netdevice scheduling: `linux/include/linux/netdevice.h`


```
20 struct timer_list {
21     struct list_head list;
22     unsigned long expires;
23     unsigned long data;
24     void (*function)(unsigned long);
25 };
```

Figure 42: Linux Timer Structure: `linux/include/linux/timer.h`

```

38  /*
39  * mod_timer is a more efficient way to update the expire field of an
40  * active timer (if the timer is inactive it will be activated)
41  * mod_timer(a,b) is equivalent to del_timer(a); a->expires = b; add_timer(a).
42  * If the timer is known to be not pending (ie, in the handler), mod_timer
43  * is less efficient than a->expires = b; add_timer(a).
44  */
45  int mod_timer(struct timer_list *timer, unsigned long expires);
46
47  extern void it_real_fn (unsigned long);
48
49  static inline void init_timer(struct timer_list * timer)
50  {
51      timer->list.next = timer->list.prev = NULL;
52  }
53
54  static inline int timer_pending (const struct timer_list * timer)
55  {
56      return timer->list.next != NULL;
57  }

```

Figure 43: Updating the Linux Timer Structure:
linux/include/linux/timer.h

```

176 void add_timer(struct timer_list *timer)
177 {
178     unsigned long flags;
179
180     spin_lock_irqsave (&timerlist_lock, flags);
181     if (timer_pending(timer))
182         goto bug;
183     internal_add_timer(timer);
184     spin_unlock_irqrestore (&timerlist_lock, flags);
185     return;
186 bug:
187     spin_unlock_irqrestore (&timerlist_lock, flags);
188     printk("bug: kernel timer added twice at %p.\n",
189           _builtin_return_address(0));
190 }

```

Figure 44: Linux Timer Calls - add_timer: linux/kernel/timer.c

```

120 static unsigned long timer_jiffies ;
121
122 static inline void internal_add_timer(struct timer_list *timer)
123 {
124     /*
125      * must be cli-ed when calling this
126      */
127     unsigned long expires = timer->expires;
128     unsigned long idx = expires - timer_jiffies ;
129     struct list_head * vec;
130
131     if (idx < TVR_SIZE) {
132         int i = expires & TVR_MASK;
133         vec = tv1.vec + i;
134     } else if (idx < 1 << (TVR_BITS + TVN_BITS)) {
135         int i = (expires >> TVR_BITS) & TVN_MASK;
136         vec = tv2.vec + i;
137     } else if (idx < 1 << (TVR_BITS + 2 * TVN_BITS)) {
138         int i = (expires >> (TVR_BITS + TVN_BITS)) & TVN_MASK;
139         vec = tv3.vec + i;
140     } else if (idx < 1 << (TVR_BITS + 3 * TVN_BITS)) {
141         int i = (expires >> (TVR_BITS + 2 * TVN_BITS)) & TVN_MASK;
142         vec = tv4.vec + i;
143     } else if ((signed long) idx < 0) {
144         /* can happen if you add a timer with expires == jiffies,
145          * or you set a timer to go off in the past
146          */
147         vec = tv1.vec + tv1.index;
148     } else if (idx <= 0xffffffffUL) {
149         int i = (expires >> (TVR_BITS + 3 * TVN_BITS)) & TVN_MASK;
150         vec = tv5.vec + i;
151     } else {
152         /* Can only get here on architectures with 64-bit jiffies */
153         INIT_LIST_HEAD(&timer->list);
154         return;
155     }
156     /*
157      * Timers are FIFO!
158      */
159     list_add(&timer->list, vec->prev);
160 }

```

Figure 45: Linux Timer Calls - internal_add_timer: linux/kernel/timer.c

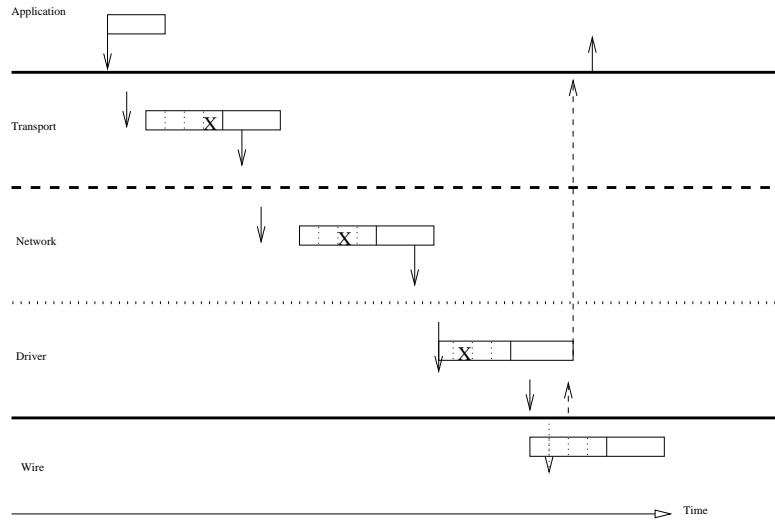


Figure 46: The overall transmission process: from Application to Wire

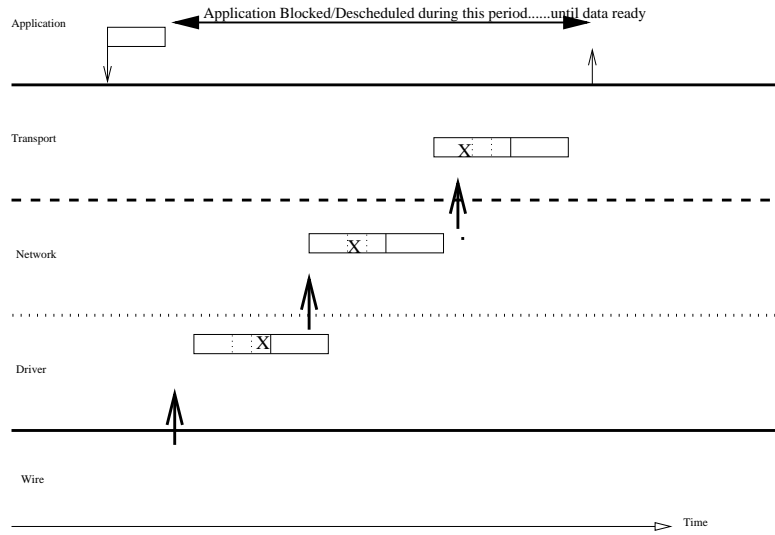


Figure 47: The overall reception process: from Wire to Application

```

236         /* RFC1323: The window in SYN & SYN/ACK segments
237          * is never scaled.
238          */
239         th->window = htons(tp->rcv_wnd);
240     } else {
241         th->window = htons(tcp_select_window(sk));
242     }
243     th->check = 0;
244     th->urg_ptr = 0;
245
246     if (tp->urg_mode &&
247         between(tp->snd_up, tcb->seq+1, tcb->seq+0xFFFF)) {
248         th->urg_ptr = htons(tp->snd_up-tcb->seq);
249         th->urg = 1;
250     }
251
252     if (tcb->flags & TCPCB_FLAG_SYN) {
253         tcp_syn_build_options((__u32*)(th + 1),
254                               tcp_advertise_mss(sk),
255                               (sysctl_flags & SYSCTL_FLAG_TSTAMPS
256                                ),

```

Figure 48: Filling in Some Key TCP fields: `linux/net/ipv4/tcp_output.c`

```

367         if (opt && opt->is_strictroute && rt->rt_dst != rt->rt_gateway)
368             goto no_route;
369
370
371         /* OK, we know where to send it, allocate and build IP header. */
372         iph = (struct iphdr *) skb_push(skb, sizeof(struct iphdr) + (opt ? opt->optlen
373             : 0));
374         *((_u16 *)iph) = htons((4 << 12) | (5 << 8) | (sk->protinfo.af_inet.tos & 0xff));
375         iph->tot_len = htons(skb->len);
376         iph->frag_off = 0;
377         iph->ttl = sk->protinfo.af_inet.ttl;
378         iph->protocol = sk->protocol;

```

Figure 49: Filling in Some Key IP bits: `linux/net/ipv4/ip_output.c`


```

1028         spin_unlock_bh(&dev->xmit_lock);
1029         if ( net_ratelimit ()
1030             printk(KERN_DEBUG "Virtual device %s asks to queue
                packet!\n", dev->name);
1031         kfree_skb(skb);
1032         return -ENETDOWN;
1033     } else {
1034         /* Recursion is detected! It is possible, unfortunately */
1035         if ( net_ratelimit ()
1036             printk(KERN_DEBUG "Dead loop on virtual device %s,
                fix it urgently!\n", dev->name);
1037     }
1038 }
1039 spin_unlock_bh(&dev->queue_lock);
1040
1041 kfree_skb(skb);

```

Figure 50: Network Device Driver Rx i: `linux/net/core/dev.c`

```

1044
1045
1046 /*=====
1047 Receiver routines
1048 =====*/
1049
1050 int netdev_max_backlog = 300;
1051 /* These numbers are selected based on intuition and some
1052 * experimentation, if you have more scientific way of doing this
1053 * please go ahead and fix things.
1054 */
1055 int no_cong_thresh = 10;
1056 int no_cong = 20;
1057 int lo_cong = 100;
1058 int mod_cong = 290;
1059
1060 struct netif_rx_stats netdev_rx_stat[NR_CPUS];
1061
1062
1063 #ifdef CONFIG_NET_HW_FLOWCONTROL
1064 atomic_t netdev_dropping = ATOMIC_INIT(0);
1065 static unsigned long netdev_fc_mask = 1;
1066 unsigned long netdev_fc_xoff = 0;
1067 spinlock_t netdev_fc_lock = SPIN_LOCK_UNLOCKED;
1068
1069 static struct
1070 {
1071     void (*stimul)(struct net_device *);
1072     struct net_device *dev;
1073 } netdev_fc_slots [32];
1074
1075 int netdev_register_fc (struct net_device *dev, void (*stimul)(struct net_device *dev))
1076 {
1077     int bit = 0;
1078     unsigned long flags;

```

Figure 51: Network Device Driver Rx ii: `linux/net/core/dev.c`

```

1135         softnet_data[cpu].cng_level = NET_RX_CN_HIGH;
1136 #ifdef RAND_LIE
1137         rd = net_random();
1138         rq = rd % netdev_max_backlog;
1139         if (rq < avg_blog) /* unlucky bastard */
1140             softnet_data[cpu].cng_level = NET_RX_DROP;
1141 #endif
1142     } else if (avg_blog > lo_cong) {
1143         softnet_data[cpu].cng_level = NET_RX_CN_MOD;
1144 #ifdef RAND_LIE
1145         rd = net_random();
1146         rq = rd % netdev_max_backlog;
1147         if (rq < avg_blog) /* unlucky bastard */
1148             softnet_data[cpu].cng_level = NET_RX_CN_HIGH;
1149 #endif
1150     } else if (avg_blog > no_cong)
1151         softnet_data[cpu].cng_level = NET_RX_CN_LOW;
1152     else /* no congestion */
1153         softnet_data[cpu].cng_level = NET_RX_SUCCESS;
1154
1155     softnet_data[cpu].avg_blog = avg_blog;
1156 }
1157
1158 #ifdef OFFLINE_SAMPLE
1159 static void sample_queue(unsigned long dummy)
1160 {
1161     /* 10 ms 0r 1ms -- i dont care -- JHS */
1162     int next_tick = 1;
1163     int cpu = smp_processor_id();
1164
1165     get_sample_stats(cpu);
1166     next_tick += jiffies;
1167     mod_timer(&samp_timer, next_tick);
1168 }
1169 #endif

```

Figure 52: Network Device Driver Rx i: `linux/net/core/dev.c`

```

1172 /**
1173  *   netif_rx      -   post buffer to the network code
1174  *   @skb: buffer to post
1175  *
1176  *   This function receives a packet from a device driver and queues it for
1177  *   the upper (protocol) levels to process. It always succeeds. The buffer
1178  *   may be dropped during processing for congestion control or by the
1179  *   protocol layers.
1180  *
1181  *   return values:
1182  *   NET_RX_SUCCESS (no congestion)
1183  *   NET_RX_CN_LOW  (low congestion)
1184  *   NET_RX_CN_MOD  (moderate congestion)
1185  *   NET_RX_CN_HIGH (high congestion)
1186  *   NET_RX_DROP   (packet was dropped)
1187  *
1188  *
1189  */
1190
1191 int netif_rx(struct sk_buff *skb)
1192 {
1193     int this_cpu = smp_processor_id();
1194     struct softnet_data *queue;
1195     unsigned long flags;
1196
1197     if (skb->stamp.tv_sec == 0)
1198         get_fast_time(&skb->stamp);

```

Figure 53: Network Device Driver Rx ii: `linux/net/core/dev.c`

```
1083     iov.iov_base = kaddr + offset;
1084     iov.iov_len = size;
1085
1086     old_fs = get_fs();
1087     set_fs(KERNEL_DS);
1088     res = sock_sendmsg(sock, &msg, size);
1089     set_fs(old_fs);
1090
```

Figure 54: Socket Receive wake up process: `linux/net/core/sock.c`

```

488     }
489     if (!ipc.opt)
490         ipc.opt = sk->protoinfo.af_inet.opt;
491
492     ufh.saddr = ipc.addr;
493     ipc.addr = daddr = ufh.daddr;
494
495     if (ipc.opt && ipc.opt->srr) {
496         if (!daddr)
497             return -EINVAL;
498         daddr = ipc.opt->faddr;
499         connected = 0;
500     }
501     tos = RT_TOS(sk->protoinfo.af_inet.tos);
502     if (sk->localroute || (msg->msg_flags&MSG_DONTROUTE) ||
503         (ipc.opt && ipc.opt->is_strictroute)) {
504         tos |= RTO_ONLINK;
505         connected = 0;
506     }
507
508     if (MULTICAST(daddr)) {
509         if (!ipc.oif)
510             ipc.oif = sk->protoinfo.af_inet.mc_index;
511         if (!ufh.saddr)
512             ufh.saddr = sk->protoinfo.af_inet.mc_addr;
513         connected = 0;
514     }
515
516     if (connected)
517         rt = (struct rtable*)sk_dst_check(sk, 0);
518
519     if (rt == NULL) {
520         err = ip_route_output(&rt, daddr, ufh.saddr, tos, ipc.oif);
521         if (err)
522             goto out;
523
524         err = -EACCES;
525         if (rt->rt_flags&RTCF_BROADCAST && !sk->broadcast)
526             goto out;
527         if (connected)
528             sk_dst_set(sk, dst_clone(&rt->u.dst));

```

Figure 55: UDP Output i: linux/net/ipv4/udp.c

```

530         if (msg->msg_flags&MSG_CONFIRM)
531             goto do_confirm;
532     back_from_confirm:
533
534         ufh.saddr = rt->rt_src;
535         if (!ipc.addr)
536             ufh.daddr = ipc.addr = rt->rt_dst;
537         ufh.uh.len = htons(ulen);
538         ufh.uh.check = 0;
539         ufh.iov = msg->msg_iov;
540         ufh.wcheck = 0;
541
542         /* RFC1122: OK. Provides the checksumming facility (MUST) as per */
543         /* 4.1.3.4. It's configurable by the application via setsockopt() */
544         /* (MAY) and it defaults to on (MUST). */
545
546         err = ip_build_xmit(sk,
547                             (sk->no_check == UDP_CSUM_NOXMIT ?
548                              udp_getfrag_nosum :
549                              udp_getfrag),
550                             &ufh, ulen, &ipc, rt, msg->msg_flags);
551
552     out:
553         ip_rt_put(rt);
554         if (free)
555             kfree(ipc.opt);
556         if (!err) {
557             UDP_INC_STATS_USER(UdpOutDatagrams);
558             return len;
559         }
560         return err;
561     do_confirm:
562         dst_confirm(&rt->u.dst);
563
564

```

Figure 56: UDP Output ii: linux/net/ipv4/udp.c

```

566         goto back_from_confirm;
567     err = 0;
568     goto out;
569 }
570
571 /*
572  *   IOCTL requests applicable to the UDP protocol
573  */
574
575 int udp_ioctl(struct sock *sk, int cmd, unsigned long arg)
576 {
577     switch(cmd)
578     {
579         case SIOCOUTQ:
580             {
581                 int amount = atomic_read(&sk->wmem_alloc);
582                 return put_user(amount, (int *)arg);
583             }
584
585         case SIOCINQ:
586             {

```

Figure 57: UDP Output iii: linux/net/ipv4/udp.c


```

683     err = copied;
684
685 out_free:
686     skb_free_datagram(sk, skb);
687 out:
688     return err;
689
690 csum_copy_err:
691     UDP_INC_STATS_BH(UdpInErrors);
692
693     /* Clear queue. */
694     if (flags & MSG_PEEK) {
695         int clear = 0;
696         spin_lock_irq(&sk->receive_queue.lock);
697         if (skb == skb_peek(&sk->receive_queue)) {
698             __skb_unlink(skb, &sk->receive_queue);
699             clear = 1;
700         }
701         spin_unlock_irq(&sk->receive_queue.lock);
702         if (clear)
703             kfree_skb(skb);
704     }
705
706     skb_free_datagram(sk, skb);
707
708     return -EAGAIN;
709 }
710
711 int udp_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
712 {
713     struct sockaddr_in *usin = (struct sockaddr_in *) uaddr;
714     struct rtable *rt;
715     int err;
716
717
718     if (addr_len < sizeof(*usin))
719         return -EINVAL;
720
721     if (usin->sin_family != AF_INET)
722         return -EAFNOSUPPORT;

```

Figure 58: UDP Input i: `linux/net/ipv4/udp.c`

```

725     err = ip_route_connect(&rt, usin->sin_addr.s_addr, sk->saddr,
726                          sk->proto.af_inet.tos | sk->localroute, sk->
727                          bound_dev_if);
728     if (err)
729         return err;
730     if ((rt->rt_flags & RTCF_BROADCAST) && !sk->broadcast) {
731         ip_rt_put(rt);
732         return -EACCES;
733     }
734     if (!sk->saddr)
735         sk->saddr = rt->rt_src;      /* Update source address */
736     if (!sk->rcv_saddr)
737         sk->rcv_saddr = rt->rt_src;
738     sk->daddr = rt->rt_dst;
739     sk->dport = usin->sin_port;
740     sk->state = TCP_ESTABLISHED;
741     sk->proto.af_inet.id = jiffies;
742
743     sk_dst_set(sk, &rt->u.dst);
744     return(0);
745 }
746
747 int udp_disconnect(struct sock *sk, int flags)
748 {
749     /*

```

Figure 59: UDP Input iii: linux/net/ipv4/udp.c

```

859     NETDEBUG(printf(KERN_DEBUG "udp_v4_hw_csum_failure.\n"));
860     skb->ip_summed = CHECKSUM_NONE;
861 }
862 if (skb->ip_summed != CHECKSUM_UNNECESSARY)
863     skb->csum = csum_tcpudp_nofold(saddr, daddr, ulen, IPPROTO_UDP, 0)
864     ;
865 /* Probably, we should checksum udp header (it should be in cache
866 * in any case) and data in tiny packets (< rx copybreak).
867 */
868 return 0;
869 }
870
871 /*
872 * All we need to do is get the socket, and then do a checksum.
873 */
874 int udp_rcv(struct sk_buff *skb)
875 {
876     struct sock *sk;
877     struct udphdr *uh;
878     unsigned short ulen;
879     struct rtable *rt = (struct rtable*)skb->dst;
880     u32 saddr = skb->nh.iph->saddr;

```

Figure 60: UDP Receive Multicast: `linux/net/ipv4/udp.c`

```
1
2 PING bells.cs.ucl.ac.uk (128.16.5.31) from 128.16.6.226 : 56(84) bytes of
   data.
3
4 64 bytes from bells.cs.ucl.ac.uk (128.16.5.31): icmp_seq=0 ttl=255 time
   =819 usec
```

Figure 61: Use of ping

```
1
2 10:38:11.705142 0:c0:4f:d3:db:c3 0:20:af:ab:e1:6e ip 98: ovavu.cs.ucl.ac.
   uk > brahms.cs.ucl.ac.uk: icmp: echo request
3
4 10:38:11.705270 0:20:af:ab:e1:6e 0:c0:4f:d3:db:c3 ip 98: brahms.cs.ucl.ac
   .uk > ovavu.cs.ucl.ac.uk: icmp: echo reply
```

Figure 62: Wire view of ping

```

1  execve("/bin/ping", ["ping", "-c", "1", "bells"], [/* 24 vars */]) = 0
2  open("/lib/libresolv.so.2", O_RDONLY) = 3
3  getpid() = 4510
4  getuid() = 0
5  socket(PF_INET, SOCK_RAW, IPPROTO_ICMP) = 3
6  setuid(0) = 0
7  gettimeofday({960371553, 351449}, NULL) = 0
8  open("/etc/resolv.conf", O_RDONLY) = 4
9  read(4, "search_ucs.ucl.ac.uk\nnameserver_1"... , 4096) = 43
10 socket(PF_UNIX, SOCK_STREAM, 0) = 4
11 connect(4, {sin_family=AF_UNIX, path="/var/run/nscd_socket"... , 110) = -1 ECONNREFUSED (Connection refused)
12 open("/etc/nsswitch.conf", O_RDONLY) = 4
13 open("/etc/host.conf", O_RDONLY) = 4
14 open("/etc/hosts", O_RDONLY) = 4
15 open("/lib/libnss_nisplus.so.2", O_RDONLY) = 4
16 uname({sys="Linux", node="ovavu.cs.ucl.ac.uk", ...}) = 0
17 open("/var/nis/NIS_COLD_START", O_RDONLY) = -1 ENOENT (No such file or directory)
18 socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 4
19 connect(4, {sin_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("128.16.5.31")...}, 16) = 0
20 send(4, "V\1\0\1\0\0\0\0\0\0\0\5bells\2cs\3ucl\2ac\2uk\0"... , 36, 0) = 36
21 time(NULL) = 960371553
22 poll([{fd=4, events=POLLIN, revents=POLLIN}], 1, 5000) = 1
23 recvfrom(4, "V\205\200\0\1\0\1\0\4\0\6\5bells\2cs\3ucl\2ac\2uk\0"... , 1024, 0, {sin_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("128.16.5.31")...}, [16]) = 262
24 close(4) = 0
25 socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 4
26 connect(4, {sin_family=AF_INET, sin_port=htons(1025), sin_addr=inet_addr("128.16.5.31")...}, 16) = 0
27 getsockname(4, {sin_family=AF_INET, sin_port=htons(1031), sin_addr=inet_addr("128.16.6.226")...}, [16]) = 0
28 close(4) = 0
29 bind(3, {sin_family=AF_INET, sin_port=htons(0), sin_addr=inet_addr("128.16.6.226")...}, 16) = 0
30 setsockopt(3, IPPROTO_RAW, [-6202], 4) = 0
31 rt_sigaction(SIGINT, {0x804a670, [], SA_INTERRUPT|0x4000000}, NULL, 8) = 0
32 rt_sigaction(SIGALRM, {0x8049e74, [], SA_INTERRUPT|0x4000000}, NULL, 8) = 0
33 gettimeofday({960371553, 403056}, NULL) = 0
34 gettimeofday({960371553, 403316}, NULL) = 0
35 sendmsg(3, {msg_name(16)={sin_family=AF_INET, sin_port=htons(0), sin_addr=inet_addr("128.16.5.31")...}, msg_iov(1)=[{"\10\0Uo\236\21\0\0a\33>9t'\6\0\10\t\n\v\fr\16\17\20\21"... , 64}], msg_controllen=0, msg_flags=0}, 0) = 64
36 rt_sigaction(SIGALRM, {0x804a670, [], SA_INTERRUPT|0x4000000}, NULL, 8) = 0
37 setitimer(ITIMER_REAL, {it_interval={0, 0}, it_value={10, 0}}, NULL) = 0
38 time(NULL) = 960371553
39 recvfrom(3, "E\0\0TR\250\0\0\377\1\337\200\20\5\37\200\20\6\342\0"... , 192, 0, {sin_family=AF_INET, sin_port=htons(4288), sin_addr=inet_addr("128.16.5.31")...}, [16]) = 84
40 gettimeofday({960371553, 407464}, NULL) = 0
41 uname({sys="Linux", node="ovavu.cs.ucl.ac.uk", ...}) = 0
42 socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 4
43 connect(4, {sin_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("128.16.5.31")...}, 16) = 0
44 send(4, "V(\1\0\0\1\0\0\0\0\0\0\00231\0015\00216\003128\7in-add"... , 42, 0) = 42
45 time(NULL) = 960371553

```

```
1
2 #include <sys/types.h>
3 #include <sys/socket.h>
4
5 int socket(int domain, int type, int protocol);
```

Figure 64: socket()

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/socket.h>
4 #include <netinet/in.h>
5 #include <netdb.h>
6
7
8 #include <errno.h>
9 extern int errno;
10
11 int child_pid;    /* Handy child handle */
12
13 main(argc, argv)
14 char *argv [];
15 {
16     long host = 0;
17     short port;
18
19     if (argc < 3) {
20         printf("What host(i.j.k.l)/port?\n");
21         exit(-1);
22     }
23
24     host = inet_addr(argv[1]);
25     port = (short)atoi(argv[2]);
26
27     if ((child_pid = fork()) == 0)
28         client(host, port); /* Child Code */
29     else
30         server(port); /* Parent */
31 }
32

```

Figure 65: Socket Programming 1 - main: `pix/sock.c`


```

34 client (host , port)
35 long host ;
36 short port ;
37 {
38
39     char buff [80];
40     int len ;
41     int asock ;
42     struct sockaddr_in where, there ;
43     int ret ;
44
45     /* Wait for child to set up listen */
46     sleep(1);
47
48     asock = socket(AF_INET, SOCK_STREAM, 0);
49     check(asock, "socket");
50
51     where.sin_family = AF_INET;
52     where.sin_addr.s_addr = 0;
53     where.sin_port = 0;
54     ret = bind(asock, &where, sizeof(struct sockaddr_in));
55     check(ret, "bind");
56
57     there.sin_family = AF_INET;
58     there.sin_addr.s_addr = host;
59     there.sin_port = htons(port);
60     ret = connect(asock, &there, sizeof(struct sockaddr_in));
61     check(ret, "connect");
62
63     while((len = read(0, buff, 1)) > 0)
64         write(asock, buff, 1); /* Write to pipe */
65
66     fprintf(stderr, "\nChild_gone\n");
67     shutdown(asock, 1);
68     close(asock);
69     exit(0);
70 }
71

```

Figure 66: Socket Programming 2 - client: pix/sock.c

```

72 server(port)
73 short port;
74 {
75     char buff [80];
76     int len;
77     int status;
78     int asock, ns; /* Socket descriptor thingy */
79     struct sockaddr_in where, there;
80     int thing = sizeof(struct sockaddr_in);
81
82
83
84     asock = socket(AF_INET, SOCK_STREAM, 0);
85     check(asock, "socket");
86
87     where.sin_family = AF_INET;
88     where.sin_addr.s_addr = INADDR_ANY; /* System Knows */
89     where.sin_port = htons(port);
90
91     status = bind(asock, &where, sizeof(struct sockaddr_in));
92     check(status, "bind");
93     status = listen(asock, 1);
94     check(status, "listen");
95
96     ns = accept(asock, &there, &thing);
97     check(ns, "accept");
98
99     while ((len = read(ns, buff, 1)) > 0)
100         write(2, buff, len); /* Write to terminal */
101
102     shutdown(asock, 0);
103     close(asock);
104
105     fprintf(stderr, "\nParent_gone\n");
106     wait(status); /* Wait for child to finish */
107     exit(0);
108 }
109
110 check(status, sysname)
111 int status;
112 char *sysname;
113 {
114     if (status < 0) {
115         perror(sysname);
116         exit(-1);
117     }
118 }

```

Figure 67: Socket Programming 3 - server: `pix/sock.c`

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/time.h>
4 #include <stdlib.h>
5
6
7 struct timeval timeout,then,now;
8
9 main(argc, argv)
10 int    argc;
11 char  *argv [];
12 {
13     int  selret ;
14
15     if (argc != 3) {
16         printf("usage: timer <#seconds> <#microseconds>");
17         exit (-1);
18     }
19
20     timeout.tv_sec = atol(argv[1]);
21     timeout.tv_usec = atol(argv[2]);
22
23     gettimeofday(&then,(struct timezone *)0);
24     if ((selret =select (0, NULL, NULL, NULL, &timeout)) < 0) {
25         check(selret , "select");
26     }
27     gettimeofday(&now,(struct timezone *)0);
28     printf("Delay = %d microseconds\n", (now.tv_sec - then.tv_sec)*1000000 +
29         now.tv_usec - then.tv_usec);
30     exit (0);
31 }

```

Figure 68: Select: `pix/poll.c`

```
1 #include <sys/poll.h>
2
3 int poll(struct pollfd *ufds, unsigned int nfd, int timeout);
4
```

Figure 69: poll()

```
1
2     struct pollfd {
3         int fd;          /* file descriptor */
4         short events;    /* requested events */
5         short revents;   /* returned events */
6     };
```

Figure 70: pollfd

```
1 setnb()
2 {
3     int    on = 1;
4     int    iret ;
5
6     if (( iret =ioctl(0, FIONBIO, (char *) &on)) < 0)
7         check(iret , " ioctl_FIONBIO_error");
8 }
```

Figure 71: NBIO: `pix/nbio.c`

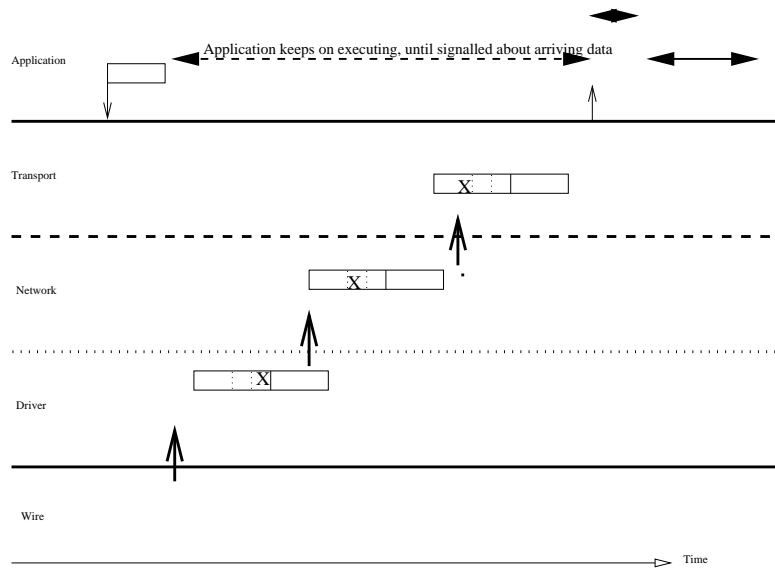


Figure 72: Asynchronous Reception

```

1 #include <signal.h>
2 #include <fcntl.h>
3
4 #define BUFFSIZE 4096
5
6 int sigflag ; /* gotta be global! */
7
8 int sigio_handler ()
9 {
10     sigflag = 1;          /* just set flag and return */
11 }
12
13 main()
14 {
15     int n, fret;
16     char buff[BUFFSIZE];
17
18     signal(SIGIO, sigio_handler);
19     fret = fcntl(0, F_SETOWN, getpid());
20     check(fret, "F_SETOWN_error");
21
22     fret = fcntl(0, F_SETFL, FASYNC);
23     check(fret, "F_SETFL_FASYNC_error");
24
25     /* This turns asynch into a loop which blocks in sigpause... */
26
27     for (;;) {
28         sigblock(sigmask(SIGIO));
29         while ( sigflag == 0)
30             sigpause(0);          /* wait for a signal */
31
32         /*
33          * We're here if (sigflag != 0). Also, we know that the
34          * SIGIO signal is currently blocked.
35          */
36
37         if ( (n = read(0, buff, BUFFSIZE)) > 0) {
38             if (write(1, buff, n) != n)
39                 check(-1, "write_error");
40         } else if (n < 0)
41             check(n, "read_error");
42         else if (n == 0)
43             exit(0);          /* EOF */
44
45         sigflag = 0;          /* turn off our flag */
46         sigsetmask(0);      /* and reenabale signals */
47     }
48 }

```

Figure 73: SIGIO: pix/asynch.c


```

111 /* Structure describing an Internet (IP) socket address. */
112 #define _SOCK_SIZE_ 16          /* sizeof(struct sockaddr) */
113 struct sockaddr_in {
114     sa_family_t    sin_family;    /* Address family      */
115     unsigned short sin_port;      /* Port number        */
116     struct in_addr sin_addr;      /* Internet address   */
117
118     /* Pad to size of struct sockaddr. */
119     unsigned char  __pad[_SOCK_SIZE_ - sizeof(short int) -
120                       sizeof(unsigned short int) - sizeof(struct in_addr)];
121 };
122 #define sin_zero    __pad        /* for BSD UNIX comp. -FvK */

```

Figure 74: IP user view of address structure: `linux/include/linux/in.h`

```

30 struct in6_addr
31 {
32     union
33     {
34         __u8          u6_addr8[16];
35         __u16         u6_addr16[8];
36         __u32         u6_addr32[4];
37     } in6_u;
38 #define s6_addr          in6_u.u6_addr8
39 #define s6_addr16       in6_u.u6_addr16
40 #define s6_addr32       in6_u.u6_addr32
41 };
42
43 struct sockaddr_in6 {
44     unsigned short int  sin6_family; /* AF_INET6 */
45     __u16               sin6_port; /* Transport layer port # */
46     __u32               sin6_flowinfo; /* IPv6 flow information */
47     struct in6_addr      sin6_addr; /* IPv6 address */
48     __u32               sin6_scope_id; /* scope id (new in RFC2553) */
49 };

```

Figure 75: IP user view of the IPv6 address structure:
linux/include/linux/in6.h

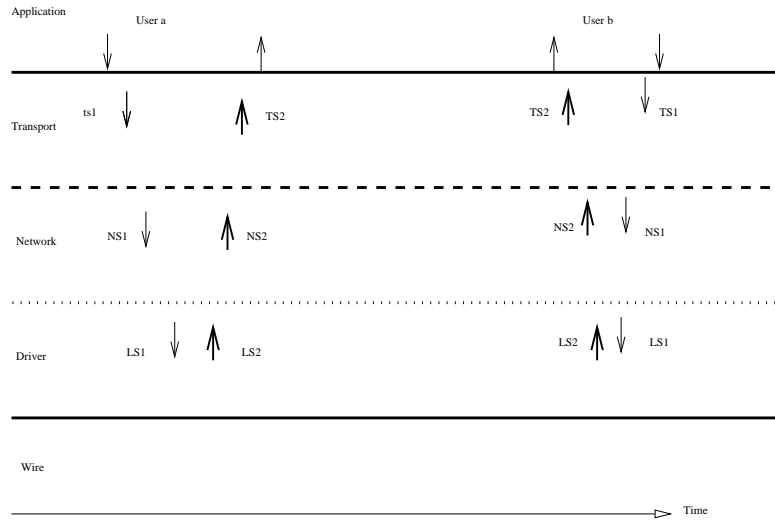


Figure 76: Producer Consumer Chain

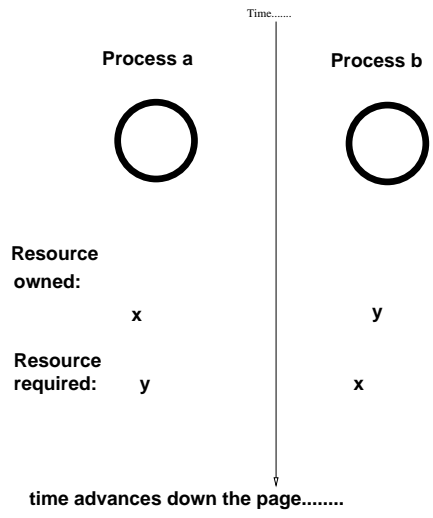


Figure 77: Deadlock

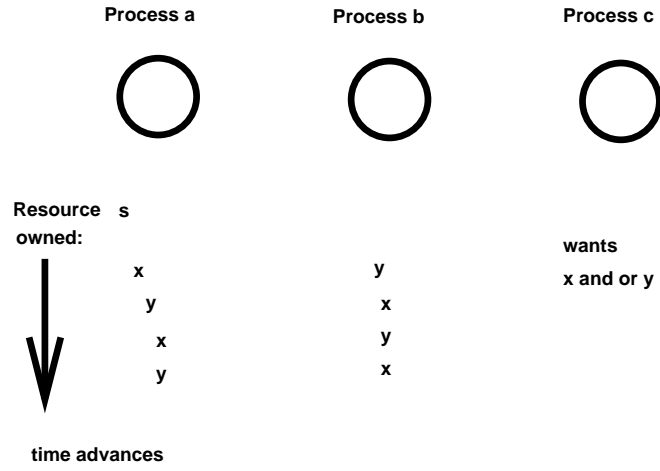


Figure 78: Livelock

```

1  /*
2  * tqqueue.h --- task queue handling for Linux.
3  *
4  * Mostly based on a proposed bottom-half replacement code written by
5  * Kai Petzke, wpp@marie.physik.tu-berlin.de.
6  *
7  * Modified for use in the Linux kernel by Theodore Ts'o,
8  * tytso@mit.edu. Any bugs are my fault, not Kai's.
9  *
10 * The original comment follows below.
11 */
12
13 #ifndef _LINUX_TQUEUE_H
14 #define _LINUX_TQUEUE_H
15
16 #include <linux/spinlock.h>
17 #include <linux/list.h>
18 #include <asm/bitops.h>
19 #include <asm/system.h>
20
21 /*
22 * New proposed "bottom half" handlers:
23 * (C) 1994 Kai Petzke, wpp@marie.physik.tu-berlin.de
24 *
25 * Advantages:
26 * - Bottom halves are implemented as a linked list. You can have as many
27 *   of them, as you want.
28 * - No more scanning of a bit field is required upon call of a bottom half.
29 * - Support for chained bottom half lists. The run_task_queue() function can be
30 *   used as a bottom half handler. This is for example useful for bottom
31 *   halves, which want to be delayed until the next clock tick.
32 *
33 * Notes:
34 * - Bottom halves are called in the reverse order that they were linked into
35 *   the list.
36 */

```

Figure 79: Task Queue structure - copyright statement:
linux/include/linux/tqueue.h

```

38 struct tq_struct {
39     struct list_head list;          /* linked list of active bh's */
40     unsigned long sync;            /* must be initialized to zero */
41     void (*routine)(void *);       /* function to call */
42     void *data;                    /* argument to function */
43 };
44
45 /*
46  * Emit code to initialise a tq_struct's routine and data pointers
47  */
48 #define PREPARE_TQUEUE(_tq, _routine, _data)          \
49     do {                                             \
50         (_tq)->routine = _routine;                 \

```

Figure 80: Task Queue (TQ) structures: `linux/include/linux/tqueue.h`

```

52     } while (0)
53
54  /*
55  * Emit code to initialise all of a tq_struct
56  */
57  #define INIT_TQUEUE(_tq, _routine, _data)          \
58      do {                                          \
59          INIT_LIST_HEAD(&(_tq)->list);           \
60          (_tq)->sync = 0;                          \
61          PREPARE_TQUEUE((_tq), (_routine), (_data)); \
62      } while (0)
63
64  typedef struct list_head task_queue;
65
66  #define DECLARE_TASK_QUEUE(q) LIST_HEAD(q)
67  #define TQ_ACTIVE(q)          (!list_empty(&q))
68
69  extern task_queue tq_timer, tq_immediate, tq_disk;
70
71  /*
72  * To implement your own list of active bottom halves, use the following
73  * two definitions:

```

Figure 81: TQ mapping to deferred processing/bh:
linux/include/linux/tqueue.h


```

75  * DECLARE_TASK_QUEUE(my_tqueue);
76  * struct tq_struct my_task = {
77  *     routine: (void (*)(void *)) my_routine,
78  *     data: &my_data
79  * };
80  *
81  * To activate a bottom half on a list , use:
82  *
83  *     queue_task(&my_task, &my_tqueue);
84  *
85  * To later run the queued tasks use
86  *
87  *     run_task_queue(&my_tqueue);
88  *
89  * This allows you to do deferred processing . For example, you could
90  * have a task queue called tq_timer, which is executed within the timer
91  * interrupt .
92  */
93
94 extern spinlock_t tqueue_lock;
95
96 /*
97  * Queue a task on a tq. Return non-zero if it was successfully
98  * added.
99  */
100 static inline int queue_task(struct tq_struct *bh_pointer, task_queue *bh_list)
101 {
102     int ret = 0;
103     if (! test_and_set_bit (0,&bh_pointer->sync)) {
104         unsigned long flags ;

```

Figure 82: TQ inline functions, queue, and run: linux/include/linux/tqueue.h

```
1  /*
2  *    linux/kernel/softirq.c
3  *
4  *    Copyright (C) 1992 Linus Torvalds
5  *
6  *    Fixed a disable_bh()/enable_bh() race (was causing a console lockup)
7  *    due bh_mask_count not atomic handling. Copyright (C) 1998 Andrea Arcangeli
8  *
9  *    Rewritten. Old one was good in 2.2, but in 2.3 it was immoral. --ANK (990903)
10 *
11 */
12 #include <linux/config.h>
13 #include <linux/mm.h>
14 #include <linux/kernel_stat.h>
15 #include <linux/interrupt.h>
16 #include <linux/smp_lock.h>
17 #include <linux/init.h>
18 #include <linux/tqueue.h>
```

Figure 83: Soft IRQs - main copyright etc: linux/kernel/softirq.c

```

20  /*
21  - No shared variables, all the data are CPU local.
22  - If a softirq needs serialization, let it serialize itself
23    by its own spinlocks.
24  - Even if softirq is serialized, only local cpu is marked for
25    execution. Hence, we get something sort of weak cpu binding.
26    Though it is still not clear, will it result in better locality
27    or will not.
28  - These softirqs are not masked by global cli() and start_bh_atomic()
29    (by clear reasons). Hence, old parts of code still using global locks
30    MUST NOT use softirqs, but insert interfacing routines acquiring
31    global locks. F.e. look at BHs implementation.
32
33  Examples:
34  - NET RX softirq. It is multithreaded and does not require
35    any global serialization.
36  - NET TX softirq. It kicks software netdevice queues, hence
37    it is logically serialized per device, but this serialization
38    is invisible to common code.
39  - Tasklets: serialized wrt itself.
40  - Bottom halves: globally serialized, grr...
41  */

```

Figure 84: Soft IRQs - main comment: linux/kernel/softirq.c

```

48 static struct softirq_action softirq_vec [32] __cacheline_aligned ;
49
50 asmlinkage void do_softirq()
51 {
52     int cpu = smp_processor_id();
53     __u32 active, mask;
54
55     if (in_interrupt())
56         return;
57
58     local_bh_disable();
59
60     local_irq_disable();
61     mask = softirq_mask(cpu);
62     active = softirq_active(cpu) & mask;
63
64     if (active) {
65         struct softirq_action *h;
66
67 restart:
68         /* Reset active bitmask before enabling irqs */
69         softirq_active(cpu) &= ~active;
70
71         local_irq_enable();
72
73         h = softirq_vec;
74         mask &= ~active;
75
76         do {
77             if (active & 1)
78                 h->action(h);
79             h++;
80             active >>= 1;
81         } while (active);

```

Figure 85: Soft IRQs - handler i: linux/kernel/softirq.c

```
83         local_irq_disable ();
84
85         active = softirq_active (cpu);
86         if ((active &= mask) != 0)
87             goto retry;
88     }
89
90     local_bh_enable();
91
92     /* Leave with locally disabled hard irqs. It is critical to close
93      * window for infinite recursion, while we help local bh count,
94      * it protected us. Now we are defenceless.
95      */
96     return;
97
98 retry:
99     goto restart;
100 }
```

Figure 86: Soft IRQs - handler ii: `linux/kernel/softirq.c`

```

103 static spinlock_t softirq_mask_lock = SPIN_LOCK_UNLOCKED;
104
105 void open_softirq(int nr, void (*action)(struct softirq_action *), void *data)
106 {
107     unsigned long flags;
108     int i;
109
110     spin_lock_irqsave(&softirq_mask_lock, flags);
111     softirq_vec[nr].data = data;
112     softirq_vec[nr].action = action;
113
114     for (i=0; i<NR_CPUS; i++)
115         softirq_mask(i) |= (1<<nr);
116     spin_unlock_irqrestore(&softirq_mask_lock, flags);
117 }

```

Figure 87: Soft IRQs - setup: `linux/kernel/softirq.c`

```

120 /* Tasklets */
121
122 struct tasklet_head tasklet_vec[NR_CPUS] __cacheline_aligned;
123
124 static void tasklet_action (struct softirq_action *a)
125 {
126     int cpu = smp_processor_id();
127     struct tasklet_struct * list ;
128
129     local_irq_disable ();
130     list = tasklet_vec[cpu].list ;
131     tasklet_vec[cpu].list = NULL;
132     local_irq_enable ();
133
134     while ( list != NULL) {
135         struct tasklet_struct *t = list ;
136
137         list = list ->next;
138
139         if ( tasklet_trylock (t) ) {
140             if (atomic_read(&t->count) == 0) {
141                 clear_bit (TASKLET_STATE_SCHED, &t->state);
142
143                 t->func(t->data);
144                 /*
145                  * talklet_trylock() uses test_and_set_bit that
146                  * imply
147                  * an mb when it returns zero, thus we need the
148                  * explicit
149                  * mb only here: while closing the critical section.
150                  */
151                 #ifdef CONFIG_SMP
152                 smp_mb__before_clear_bit();
153                 #endif
154                 tasklet_unlock (t);
155                 continue;
156             }
157             tasklet_unlock (t);
158         }
159         local_irq_disable ();
160         t->next = tasklet_vec[cpu].list ;
161         tasklet_vec[cpu].list = t;
162         __cpu_raise_softirq (cpu, TASKLET_SOFTIRQ);
163         local_irq_enable ();
164     }
165 }

```

Figure 88: Soft IRQs - tasklet handler: linux/kernel/softirq.c

```
167 struct tasklet_head tasklet_hi_vec [NR_CPUS] __cacheline_aligned;
168
169 static void tasklet_hi_action (struct softirq_action *a)
```

Figure 89: Soft IRQs - tasklet hi priority handler: `linux/kernel/softirq.c`


```

203 void tasklet_init (struct tasklet_struct *t,
204                   void (*func)(unsigned long), unsigned long data)
205 {
206     t->func = func;
207     t->data = data;
208     t->state = 0;
209     atomic_set(&t->count, 0);
210 }
211
212 void tasklet_kill (struct tasklet_struct *t)
213 {
214     if (in_interrupt ())
215         printk("Attempt to kill tasklet from interrupt\n");
216
217     while ( test_and_set_bit (TASKLET_STATE_SCHED, &t->state)) {
218         current->state = TASK_RUNNING;
219         do {
220             current->policy |= SCHED_YIELD;
221             schedule();
222         } while ( test_bit (TASKLET_STATE_SCHED, &t->state));
223     }
224     tasklet_unlock_wait (t);
225     clear_bit (TASKLET_STATE_SCHED, &t->state);
226 }

```

Figure 90: Soft IRQs - tasklet initialisation and destruction:
linux/kernel/softirq.c

```

230 /* Old style BHs */
231
232 static void (*bh_base[32])(void);
233 struct tasklet_struct bh_task_vec[32];
234
235 /* BHs are serialized by spinlock global_bh_lock.
236
237    It is still possible to make synchronize_bh() as
238    spin_unlock_wait(&global_bh_lock). This operation is not used
239    by kernel now, so that this lock is not made private only
240    due to wait_on_irq().
241
242    It can be removed only after auditing all the BHs.
243 */
244 spinlock_t global_bh_lock = SPIN_LOCK_UNLOCKED;
245
246 static void bh_action(unsigned long nr)
247 {
248     int cpu = smp_processor_id();
249
250     if (!spin_trylock(&global_bh_lock))
251         goto resched;
252
253     if (!hardirq_trylock(cpu))
254         goto resched_unlock;
255
256     if (bh_base[nr])
257         bh_base[nr]();
258
259     hardirq_endlock(cpu);
260     spin_unlock(&global_bh_lock);
261     return;
262
263 resched_unlock:
264     spin_unlock(&global_bh_lock);
265 resched:
266     mark_bh(nr);
267 }

```

Figure 91: Soft IRQs - handler for legacy bottom half code: `linux/kernel/softirq.c`

```
269 void init_bh(int nr, void (*routine)(void))
270 {
271     bh_base[nr] = routine;
272     mb();
273 }
274
275 void remove_bh(int nr)
276 {
277     tasklet_kill (bh_task_vec+nr);
278     bh_base[nr] = NULL;
279 }
```

Figure 92: Soft IRQs - handler for legacy bottom half code:
`linux/kernel/softirq.c`

```

281 void __init softirq_init ()
282 {
283     int i;
284
285     for (i=0; i<32; i++)
286         tasklet_init (bh_task_vec+i, bh_action, i);
287
288     open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
289     open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
290 }
291
292 void __run_task_queue(task_queue *list)
293 {
294     struct list_head head, *next;
295     unsigned long flags;
296
297     spin_lock_irqsave (&queue_lock, flags);
298     list_add (&head, list);
299     list_del_init (list);
300     spin_unlock_irqrestore (&queue_lock, flags);
301
302     next = head.next;
303     while (next != &head) {
304         void (*f) (void *);
305         struct tq_struct *p;
306         void *data;
307
308         p = list_entry (next, struct tq_struct, list);
309         next = next->next;
310         f = p->routine;
311         data = p->data;
312         wmb();
313         p->sync = 0;
314         if (f)
315             f(data);
316     }
317 }

```

Figure 93: Soft IRQs - initialise softirq and run tasklets:
linux/kernel/softirq.c

```

217 #define NETDEV_BOOT_SETUP_MAX 8
218
219
220 /*
221  * The DEVICE structure.
222  * Actually, this whole structure is a big mistake. It mixes I/O
223  * data with strictly "high-level" data, and it has to know about
224  * almost every data structure used in the INET module.
225  *
226  * FIXME: cleanup struct net_device such that network protocol info
227  * moves out.
228  */
229
230 struct net_device
231 {
232
233     /*
234     * This is the first field of the "visible" part of this structure
235     * (i.e. as seen by users in the "Space.c" file). It is the name
236     * the interface.
237     */
238     char                name[IFNAMSIZ];
239
240     /*
241     * I/O specific fields
242     * FIXME: Merge these and struct ifmap into one
243     */
244     unsigned long       rmem_end;    /* shmem "recv" end */
245     unsigned long       rmem_start;  /* shmem "recv" start */
246     unsigned long       mem_end;     /* shared mem end */

```

Figure 94: Network/OS Device Level Abstraction 0:
linux/include/linux/netdevice.h

```

248 unsigned long      base_addr;    /* device I/O address */
249 unsigned int      irq;          /* device IRQ number */
250
251 /*
252  *      Some hardware also needs these fields, but they are not
253  *      part of the usual set specified in Space.c.
254  */
255
256 unsigned char      if_port;      /* Selectable AUI, TP,..*/
257 unsigned char      dma;          /* DMA channel */
258
259 unsigned long      state;
260
261 struct net_device  *next;
262
263 /* The device initialization function. Called only once. */
264 int                (*init)(struct net_device *dev);
265
266 /* ----- Fields preinitialized in Space.c finish here ----- */
267
268 struct net_device  *next_sched;
269
270 /* Interface index. Unique device identifier */
271 int                ifindex;

```

Figure 95: Network/OS Device Level Abstraction 1:
linux/include/linux/netdevice.h

```

272         int                iflink ;
273
274
275     struct net_device_stats * (* get_stats )(struct net_device *dev);
276     struct iw_statistics *   (* get_wireless_stats )(struct net_device *dev);
277
278     /*
279     * This marks the end of the "visible" part of the structure. All
280     * fields hereafter are internal to the system, and may change at
281     * will (read: may be cleaned up at will).
282     */
283
284     /* These may be needed for future network-power-down code. */
285     unsigned long          trans_start; /* Time (in jiffies) of last Tx */
286     unsigned long          last_rx;     /* Time of last Rx */
287
288     unsigned short         flags; /* interface flags (a la BSD) */
289     unsigned short         gflags;
290     unsigned               mtu; /* interface MTU value */
291     unsigned short         type; /* interface hardware type */
292     unsigned short         hard_header_len; /* hardware hdr length */
293     void                   *priv; /* pointer to private data */
294
295     struct net_device       *master; /* Pointer to master device of a group,
296                                     * which this device is member of.
297                                     */
298
299     /* Interface address info. */
300     unsigned char          broadcast[MAX_ADDR_LEN]; /* hw bcast add */
301     unsigned char          pad; /* make dev_addr aligned to 8 bytes
302                                */
303     unsigned char          dev_addr[MAX_ADDR_LEN]; /* hw address */
304     unsigned char          addr_len; /* hardware address length */
305
306     struct dev_mc_list      *mc_list; /* Multicast mac addresses */
307     int                    mc_count; /* Number of installed mcasts */
308     int                    promiscuity;
309     int                    allmulti;
310
311     int                    watchdog_timeo;
312     struct timer_list       watchdog_timer;

```

Figure 96: Network/OS Device Level Abstraction 2:
linux/include/linux/netdevice.h

```

313     /* Protocol specific pointers */
314
315     void          *atalk_ptr;    /* AppleTalk link */
316     void          *ip_ptr;      /* IPv4 specific data */
317     void          *dn_ptr;      /* DECnet specific data */
318     void          *ip6_ptr;     /* IPv6 specific data */
319     void          *ec_ptr;      /* Econet specific data */
320
321     struct Qdisc  *qdisc;
322     struct Qdisc  *qdisc_sleeping;
323     struct Qdisc  *qdisc_list;
324     struct Qdisc  *qdisc_ingress;
325     unsigned long tx_queue_len; /* Max frames per queue allowed */
326
327     /* hard_start_xmit synchronizer */
328     spinlock_t    xmit_lock;
329     /* cpu id of processor entered to hard_start_xmit or -1,
330      * if nobody entered there.
331      */
332     int           xmit_lock_owner;
333     /* device queue lock */
334     spinlock_t    queue_lock;
335     /* Number of references to this device */
336     atomic_t      refcnt;
337     /* The flag marking that device is unregistered, but held by a user */
338     int           deadbeaf;
339
340     /* Net device features */
341     int           features;
342     #define NETIF_F_SG          1    /* Scatter/gather IO. */
343     #define NETIF_F_IP_CSUM     2    /* Can checksum only TCP/UDP over IPv4. */
344     #define NETIF_F_NO_CSUM     4    /* Does not require checksum. F.e. loopback
345      * */
346     #define NETIF_F_HW_CSUM     8    /* Can checksum all the packets. */
347     #define NETIF_F_DYNALLOC    16   /* Self-destructable device. */
348     #define NETIF_F_HIGHDMA     32   /* Can DMA to high memory. */

```

Figure 97: Network/OS Device Level Abstraction 3:
linux/include/linux/netdevice.h


```

348 #define NETIF_F_FRAGLIST 1      /* Scatter/gather IO. */
349
350 /* Called after device is detached from network. */
351 void (*uninit)(struct net_device *dev);
352 /* Called after last user reference disappears. */
353 void (*destructor)(struct net_device *dev);
354
355 /* Pointers to interface service routines. */
356 int (*open)(struct net_device *dev);
357 int (*stop)(struct net_device *dev);
358 int (*hard_start_xmit)(struct sk_buff *skb,
359                       struct net_device *dev);
360 int (*hard_header)(struct sk_buff *skb,
361                   struct net_device *dev,
362                   unsigned short type,
363                   void *daddr,
364                   void *saddr,
365                   unsigned len);
366 int (*rebuild_header)(struct sk_buff *skb);
367 #define HAVE_MULTICAST
368 void (*set_multicast_list)(struct net_device *dev);
369 #define HAVE_SET_MAC_ADDR
370 int (*set_mac_address)(struct net_device *dev,
371                       void *addr);
372 #define HAVE_PRIVATE_IOCTL
373 int (*do_ioctl)(struct net_device *dev,
374               struct ifreq *ifr, int cmd);
375 #define HAVE_SET_CONFIG
376 int (*set_config)(struct net_device *dev,
377                 struct ifmap *map);
378 #define HAVE_HEADER_CACHE
379 int (*hard_header_cache)(struct neighbour *neigh,
380                          struct hh_cache *hh);
381 void (*header_cache_update)(struct hh_cache *hh,
382                             struct net_device *dev,
383                             unsigned char * haddr);
384 #define HAVE_CHANGE_MTU
385 int (*change_mtu)(struct net_device *dev, int new_mtu);
386
387 #define HAVE_TX_TIMEOUT
388 void (*tx_timeout)(struct net_device *dev);
389
390 int (*hard_header_parse)(struct sk_buff *skb,
391                          unsigned char *haddr);
392 int (*neigh_setup)(struct net_device *dev, struct neigh_parms
393                  *);
394 int (*accept_fastpath)(struct net_device *, struct dst_entry*);
395
396 /* open/release and usage marking */
397 struct module *owner;

```

Figure 98: Network/OS Device Level Abstraction 4:
linux/include/linux/netdevice.h

```

1028         spin_unlock_bh(&dev->xmit_lock);
1029         if ( net_ratelimit ()
1030             printk(KERN_DEBUG "Virtual device %s asks to queue
                packet!\n", dev->name);
1031         kfree_skb(skb);
1032         return -ENETDOWN;
1033     } else {
1034         /* Recursion is detected! It is possible, unfortunately */
1035         if ( net_ratelimit ()
1036             printk(KERN_DEBUG "Dead loop on virtual device %s,
                fix it urgently!\n", dev->name);
1037     }
1038 }
1039 spin_unlock_bh(&dev->queue_lock);
1040
1041 kfree_skb(skb);

```

Figure 99: Network Device Driver Rx: `linux/net/core/dev.c`

```

163 #define RX_STATUS (ioaddr + 0x06)
164 #define RX_CMD RX_STATUS
165 #define TX_STATUS (ioaddr + 0x07)
166 #define TX_CMD TX_STATUS
167 #define GP_LOW (ioaddr + 0x08)
168 #define GP_HIGH (ioaddr + 0x09)
169 #define RX_BUF_CLR (ioaddr + 0x0A)
170 #define RX_LOW (ioaddr + 0x0A)
171 #define RX_HIGH (ioaddr + 0x0B)
172 #define SAPROM (ioaddr + 0x0C)
173 #define AX_STATUS (ioaddr + 0x0E)
174 #define AX_CMD AX_STATUS
175 #define DATAPORT (ioaddr + 0x0F)
176 #define TX_RDY 0x08 /* In TX_STATUS */
177
178 #define EL1_DATAPTR 0x08
179 #define EL1_RXPTR 0x0A
180 #define EL1_SAPROM 0x0C
181 #define EL1_DATAPORT 0x0f
182
183 /*
184 * Writes to the ax command register.
185 */
186
187 #define AX_OFF 0x00 /* Irq off, buffer access on */
188 #define AX_SYS 0x40 /* Load the buffer */
189 #define AX_XMIT 0x44 /* Transmit a packet */
190 #define AX_RX 0x48 /* Receive a packet */
191 #define AX_LOOP 0x0C /* Loopback mode */
192 #define AX_RESET 0x80

```

Figure 100: Ethernet Driver 0 - data: linux/drivers/net/3c501.c

```

194  /*
195  *      Normal receive mode written to RX_STATUS. We must intr on short packets
196  *      to avoid bogus rx lockups.
197  */
198
199  #define RX_NORM 0xA8      /* 0x68 == all addr, 0xA8 only to me. */
200  #define RX_PROM 0x68     /* Senior Prom, uhmm promiscuous mode. */
201  #define RX_MULT 0xE8     /* Accept multicast packets. */
202  #define TX_NORM 0x0A     /* Interrupt on everything that might hang the chip
203                          */
204
205  /*
206  *      TX_STATUS register.
207  */
208  #define TX_COLLISION 0x02
209  #define TX_16COLLISIONS 0x04
210  #define TX_READY 0x08
211
212  #define RX_RUNT 0x08
213  #define RX_MISSED 0x01   /* Missed a packet due to 3c501 braindamage. */
214  #define RX_GOOD 0x30    /* Good packet 0x20, or simple overflow 0x10. */

```

Figure 101: Ethernet Driver 1 - data: linux/drivers/net/3c501.c

```

274     /*
275     *      Reserve I/O resource for exclusive use by this driver
276     */
277
278     if (!request_region(ioaddr, EL1_IO_EXTENT, dev->name))
279         return -ENODEV;
280
281     /*
282     *      Read the station address PROM data from the special port.
283     */
284
285     for (i = 0; i < 6; i++)
286     {
287         outw(i, ioaddr + EL1_DATAPTR);
288         station_addr[i] = inb(ioaddr + EL1_SAPROM);
289     }
290     /*
291     *      Check the first three octets of the S.A. for 3Com's prefix, or
292     *      for the Sager NP943 prefix.
293     */
294
295     if (station_addr[0] == 0x02 && station_addr[1] == 0x60
296         && station_addr[2] == 0x8c)
297     {
298         mname = "3c501";
299     } else if (station_addr[0] == 0x00 && station_addr[1] == 0x80
300         && station_addr[2] == 0xC8)
301     {
302         mname = "NP943";
303     }
304     else {
305         release_region(ioaddr, EL1_IO_EXTENT);
306         return -ENODEV;
307     }

```

Figure 102: Ethernet Driver 2 - init a: linux/drivers/net/3c501.c

```

309     /*
310     *   We auto-IRQ by shutting off the interrupt line and letting it
311     *   float
312     *   high.
313     */
314     if (dev->irq < 2)
315     {
316         autoirq_setup(2);
317         inb(RX_STATUS);    /* Clear pending interrupts. */
318         inb(TX_STATUS);
319         outb(AX_LOOP + 1, AX_CMD);
320
321         outb(0x00, AX_CMD);
322
323         autoirq = autoirq_report(1);
324
325         if (autoirq == 0)
326         {
327             printk("%s probe at %x failed to detect IRQ line.\n",
328                   mname, ioaddr);
329             release_region(ioaddr, EL1_IO_EXTENT);
330             return -EAGAIN;
331         }
332     }
333
334     outb(AX_RESET+AX_LOOP, AX_CMD);    /* Loopback mode. */
335     dev->base_addr = ioaddr;
336     memcpy(dev->dev_addr, station_addr, ETH_ALEN);
337
338     if (dev->mem_start & 0xf)
339         el_debug = dev->mem_start & 0x7;
340     if (autoirq)
341         dev->irq = autoirq;
342
343     printk(KERN_INFO "%s: %s Ethernet Link at %x, using %s IRQ %d.\n", dev->
344            name, mname, dev->base_addr,
345            autoirq ? "auto":"assigned", dev->irq);

```

Figure 103: Ethernet Driver 2 - init b: linux/drivers/net/3c501.c

```

346 #ifdef CONFIG_IP_MULTICAST
347     printk(KERN_WARNING "WARNING: Use of the 3c501 in a multicast kernel is
        _NOT_ recommended.\n");
348 #endif
349
350     if (el_debug)
351         printk("%s", version);
352
353     /*
354      *   Initialize the device structure.
355      */
356
357     dev->priv = kmalloc(sizeof(struct net_local), GFP_KERNEL);
358     if (dev->priv == NULL) {
359         release_region(ioaddr, EL1IO_EXTENT);
360         return -ENOMEM;
361     }
362     memset(dev->priv, 0, sizeof(struct net_local));

```

Figure 104: Ethernet Driver 2 - init c: `linux/drivers/net/3c501.c`

```

364     lp=dev->priv;
365     spin_lock_init (&lp->lock);
366
367     /*
368      *   The EL1-specific entries in the device structure.
369      */
370
371     dev->open = &el_open;
372     dev->hard_start_xmit = &el_start_xmit;
373     dev->tx_timeout = &el_timeout;
374     dev->watchdog_timeo = HZ;
375     dev->stop = &el1_close;
376     dev->get_stats = &el1_get_stats;
377     dev->set_multicast_list = &set_multicast_list;
378
379     /*
380      *   Setup the generic properties
381      */
382
383     ether_setup(dev);
384
385     return 0;
386 }
387
388 /**
389  *   el1_open:
390  *   @dev: device that is being opened
391  *
392  *   When an ifconfig is issued which changes the device flags to include
393  *   IFF_UP this function is called. It is only called when the change
394  *   occurs, not when the interface remains up. #el1_close will be called

```

Figure 105: Ethernet Driver 2 - init d: linux/drivers/net/3c501.c


```
451 /**
452  * el_start_xmit:
453  * @skb: The packet that is queued to be sent
454  * @dev: The 3c501 card we want to throw it down
455  *
456  * Attempt to send a packet to a 3c501 card. There are some interesting
457  * catches here because the 3c501 is an extremely old and therefore
458  * stupid piece of technology.
459  *
460  * If we are handling an interrupt on the other CPU we cannot load a packet
461  * as we may still be attempting to retrieve the last RX packet buffer.
462  *
463  * When a transmit times out we dump the card into control mode and just
464  * start again. It happens enough that it isnt worth logging.
465  *
466  * We avoid holding the spin locks when doing the packet load to the board.
467  * The device is very slow, and its DMA mode is even slower. If we held the
468  * lock while loading 1500 bytes onto the controller we would drop a lot of
469  * serial port characters. This requires we do extra locking, but we have
470  * no real choice.
471  */
```

Figure 106: Ethernet Driver 2 - txstart a: linux/drivers/net/3c501.c

```

473 static int el_start_xmit(struct sk_buff *skb, struct net_device *dev)
474 {
475     struct net_local *lp = (struct net_local *)dev->priv;
476     int ioaddr = dev->base_addr;
477     unsigned long flags;
478
479     /*
480      *   Avoid incoming interrupts between us flipping txing and flipping
481      *   mode as the driver assumes txing is a faithful indicator of card
482      *   state
483      */
484
485     spin_lock_irqsave(&lp->lock, flags);
486
487     /*
488      *   Avoid timer-based retransmission conflicts.
489      */
490
491     netif_stop_queue(dev);

```

Figure 107: Ethernet Driver 2 - txstart a: linux/drivers/net/3c501.c

```

493     do
494     {
495         int gp_start = 0x800 - (ETH_ZLEN < skb->len ? skb->len :
496             ETH_ZLEN);
497         unsigned char *buf = skb->data;
498
499         lp->tx_pkt_start = gp_start;
500         lp->collisions = 0;
501
502         lp->stats.tx_bytes += skb->len;
503
504         /*
505          *   Command mode with status cleared should [in theory]
506          *   mean no more interrupts can be pending on the card.
507          */
508         outb_p(AX_SYS, AX_CMD);
509         inb_p(RX_STATUS);
510         inb_p(TX_STATUS);
511
512         lp->loading = 1;
513         lp->txing = 1;
514
515         /*
516          *   Turn interrupts back on while we spend a pleasant afternoon
517          *   loading bytes into the board
518          */
519
520         spin_unlock_irqrestore(&lp->lock, flags);
521
522         outw(0x00, RX_BUF_CLR); /* Set rx packet area to 0. */
523         outw(gp_start, GP_LOW); /* aim - packet will be loaded into
524             buffer start */
525         outsb(DATAPORT, buf, skb->len); /* load buffer (usual thing each
526             byte increments the pointer) */
527         outw(gp_start, GP_LOW); /* the board reuses the same register
528             */

```

Figure 108: Ethernet Driver 2 - txstart b: linux/drivers/net/3c501.c

```

527         if (lp->loading != 2)
528         {
529             outb(AX_XMIT, AX_CMD);    /* fire ... Trigger xmit. */
530             lp->loading=0;
531             dev->trans_start = jiffies;
532             if (el_debug > 2)
533                 printk("xmitted.\n");
534             dev_kfree_skb (skb);
535             return 0;
536         }
537         /* A receive upset our load, despite our best efforts */
538         if (el_debug>2)
539             printk("%s:burped_during_tx_load.\n", dev->name);
540         spin_lock_irqsave (&lp->lock, flags);
541     }
542     while(1);
543 }
544 }

```

Figure 109: Ethernet Driver 2 - txstart b: linux/drivers/net/3c501.c

```

581
582     /*
583     *     What happened ?
584     */
585
586     axsr = inb(AX_STATUS);
587
588     /*
589     *     Log it
590     */
591
592     if (el_debug > 3)
593         printk(KERN_DEBUG "%s: el_interrupt() aux=%#02x", dev->name,
594                axsr);
595
596     if (lp->loading==1 && !lp->txing)
597         printk(KERN_WARNING "%s: Inconsistent state loading while not in tx
598                \n",
599                dev->name);

```

Figure 110: Ethernet Driver 3 - interrupt al: linux/drivers/net/3c501.c

```

599     if (lp->txing)
600     {
601
602         /*
603         *   Board in transmit mode. May be loading. If we are
604         *   loading we shouldn't have got this.
605         */
606
607         int txsr = inb(TX_STATUS);
608
609         if (lp->loading==1)
610         {
611             if (el_debug > 2)
612             {
613                 printk(KERN_DEBUG "%s: Interrupt while loading",
614                        dev->name);
615                 printk(" txsr=%02x gp=%04x rp=%04x\n", txsr, inw(
616                        GP_LOW), inw(RX_LOW));
617             }
618             lp->loading=2;      /* Force a reload */
619             spin_unlock(&lp->lock);
620             return;
621         }
622
623         if (el_debug > 6)
624             printk(KERN_DEBUG " txsr=%02x gp=%04x rp=%04x", txsr,
625                    inw(GP_LOW), inw(RX_LOW));

```

Figure 111: Ethernet Driver 3 - interrupt a2: linux/drivers/net/3c501.c

```

624         if ((axsr & 0x80) && (txsr & TX_READY) == 0)
625         {
626             /*
627              *      FIXME: is there a logic to whether to keep on trying
628              *      or
629              *      reset immediately ?
630              */
631             if (el_debug > 1)
632                 printk("%s: Unusual interrupt during Tx, txsr=%02x,
633                        axsr=%02x
634                        gp=%03x, rp=%03x.\n", dev->name, txsr, axsr,
635                        inw(ioaddr + EL1_DATAPTR), inw(ioaddr + EL1_RXPTR));
636             lp->txing = 0;
637             netif_wake_queue(dev);
638         }
639         else if (txsr & TX_16COLLISIONS)
640         {
641             /*
642              *      Timed out
643              */
644             if (el_debug)
645                 printk("%s: Transmit failed 16 times, Ethernet jammed?\n", dev->name);
646             outb(AX_SYS, AX_CMD);
647             lp->txing = 0;
648             lp->stats.tx_aborted_errors++;
649             netif_wake_queue(dev);
650         }

```

Figure 112: Ethernet Driver 3 - interrupt b1: linux/drivers/net/3c501.c

```

649     else if (txsr & TX_COLLISION)
650     {
651         /*
652          *   Retrigger xmit.
653          */
654
655         if (el_debug > 6)
656             printk("retransmitting after a collision.\n");
657         /*
658          *   Poor little chip can't reset its own start pointer
659          */
660
661         outb(AX_SYS, AX_CMD);
662         outw(lp->tx_pkt_start, GP_LOW);
663         outb(AX_XMIT, AX_CMD);
664         lp->stats.collisions++;
665         spin_unlock(&lp->lock);
666         return;
667     }
668     else
669     {
670         /*
671          *   It worked.. we will now fall through and receive
672          */
673         lp->stats.tx_packets++;
674         if (el_debug > 6)
675             printk("Tx succeeded_%s\n",
676                 (txsr & TX_RDY) ? " " : "but tx is busy!");
677         /*
678          *   This is safe the interrupt is atomic WRT itself.
679          */
680
681         lp->txing = 0;
682         netif_wake_queue(dev); /* In case more to transmit */
683     }
684 }

```

Figure 113: Ethernet Driver 3 - interrupt b2: linux/drivers/net/3c501.c


```

685     else
686     {
687         /*
688          *   In receive mode.
689          */
690
691         int rxsr = inb(RX_STATUS);
692         if (el_debug > 5)
693             printk("_rxsr=%02x\txsr=%02x\rp=%04x", rxsr, inb(TX_STATUS
694                 ),inw(RX_LOW));
695
696         /*
697          *   Just reading rx_status fixes most errors.
698          */
699         if (rxsr & RX_MISSED)
700             lp->stats.rx_missed_errors++;
701         else if (rxsr & RX_RUNT)
702             /* Handled to avoid board lock-up. */
703             lp->stats.rx_length_errors++;
704             if (el_debug > 5)
705                 printk("_runt.\n");
706         }
707         else if (rxsr & RX_GOOD)
708         {
709             /*
710              *   Receive worked.
711              */
712             el_receive (dev);
713         }
714         else
715         {
716             /*
717              *   Nothing? Something is broken!
718              */
719             if (el_debug > 2)
720                 printk("%s:_No_packet_seen,_rxsr=%02x_**resetting_3c501
721                     ***\n",
722                         dev->name, rxsr);
723             el_reset (dev);
724         }
725         if (el_debug > 3)
726             printk(".\n");
727     }

```

Figure 114: Ethernet Driver 3 - interrupt c: linux/drivers/net/3c501.c

```

726     /*
727     *     Move into receive mode
728     */
729
730     outb(AX_RX, AX_CMD);
731     outw(0x00, RX_BUF_CLR);
732     inb(RX_STATUS);    /* Be certain that interrupts are cleared. */
733     inb(TX_STATUS);
734     spin_unlock(&lp->lock);
735     return;
736 }
737
738
739 /**
740 * el_receive:
741 * @dev: Device to pull the packets from
742 *
743 * We have a good packet. Well, not really "good", just mostly not broken.
744 * We must check everything to see if it is good. In particular we occasionally
745 * get wild packet sizes from the card. If the packet seems sane we PIO it
746 * off the card and queue it for the protocol layers.
747 */

```

Figure 115: Ethernet Driver 3 - interrupt d: linux/drivers/net/3c501.c

```

739 /**
740  * el_receive:
741  * @dev: Device to pull the packets from
742  *
743  * We have a good packet. Well, not really "good", just mostly not broken.
744  * We must check everything to see if it is good. In particular we occasionally
745  * get wild packet sizes from the card. If the packet seems sane we PIO it
746  * off the card and queue it for the protocol layers.
747  */
748
749 static void el_receive(struct net_device *dev)
750 {
751     struct net_local *lp = (struct net_local *)dev->priv;
752     int ioaddr = dev->base_addr;
753     int pkt_len;
754     struct sk_buff *skb;
755
756     pkt_len = inw(RX_LOW);
757
758     if (el_debug > 4)
759         printk("_el_receive_%d.\n", pkt_len);
760
761     if ((pkt_len < 60) || (pkt_len > 1536))
762     {
763         if (el_debug)
764             printk("%s:_bogus_packet,length=%d\n", dev->name, pkt_len);
765         lp->stats.rx_over_errors++;
766         return;
767     }

```

Figure 116: Ethernet Driver 4 - receive: `linux/drivers/net/3c501.c`

```

769     /*
770      *      Command mode so we can empty the buffer
771      */
772
773     outb(AX_SYS, AX_CMD);
774     skb = dev_alloc_skb(pkt_len+2);
775
776     /*
777      *      Start of frame
778      */
779
780     outw(0x00, GP_LOW);
781     if (skb == NULL)
782     {
783         printk("%s: Memory squeeze, dropping packet.\n", dev->name);
784         lp->stats.rx_dropped++;
785         return;
786     }
787     else
788     {
789         skb_reserve(skb,2);      /* Force 16 byte alignment */
790         skb->dev = dev;
791         /*
792          *      The read increments through the bytes. The interrupt
793          *      handler will fix the pointer when it returns to
794          *      receive mode.
795          */
796         insb(DATAPORT, skb_put(skb,pkt_len), pkt_len);
797         skb->protocol=eth_type_trans(skb,dev);
798         netif_rx(skb);
799         dev->last_rx = jiffies;
800         lp->stats.rx_packets++;
801         lp->stats.rx_bytes+=pkt_len;
802     }
803     return;

```

Figure 117: Ethernet Driver 4 - receive: `linux/drivers/net/3c501.c`

```
879 static struct net_device_stats *ell_get_stats (struct net_device *dev)
880 {
881     struct net_local *lp = (struct net_local *)dev->priv;
882     return &lp->stats;
883 }
884
885 /**
886  * set_multicast_list:
887  * @dev: The device to adjust
888  *
889  * Set or clear the multicast filter for this adaptor to use the best-effort
890  * filtering supported. The 3c501 supports only three modes of filtering.
891  * It always receives broadcasts and packets for itself. You can choose to
892  * optionally receive all packets, or all multicast packets on top of this.
893  */
```

Figure 118: Ethernet Driver 4 - stats: linux/drivers/net/3c501.c

```
1
2 Kernel Interface table
3 Iface MTU Met  RX-OK RX-ERR RX-DRP RX-OVR TX-OK TX-ERR TX-DRP
4 TX-OVR Flg
5 eth0 1476 0 7545013 3 0 3 2303995 0 0 0 BRU
6 lo 3924 0 18 0 0 0 18 0 0 0 LRU
```

Figure 119: Netstat -i

```
1
2 [ucachgd@ovavu ucachgd]$ /sbin/ifconfig eth0
3 eth0      Link encap:10Mbps Ethernet HWaddr 00:C0:4F:D3:DB:C3
4           inet addr:128.16.6.226 Bcast:128.16.15.255
5 Mask:255.255.240.0
6           UP BROADCAST RUNNING MULTICAST MTU:1476 Metric:1
7           RX packets:2147483647 errors:7545063 dropped:3 overruns:0
8           TX packets:3 errors:0 dropped:0 overruns:207165461
9           Interrupt:10 Base address:0x300
```

Figure 120: ifconfig

```

891  * It always receives broadcasts and packets for itself . You can choose to
892  * optionally receive all packets, or all multicast packets on top of this.
893  */
894
895  static void set_multicast_list (struct net_device *dev)
896  {
897      int ioaddr = dev->base_addr;
898
899      if (dev->flags&IFF_PROMISC)
900      {
901          outb(RX_PROM, RX_CMD);
902          inb(RX_STATUS);
903      }
904      else if (dev->mc_list || dev->flags&IFF_ALLMULTI)
905      {
906          outb(RX_MULT, RX_CMD); /* Multicast or all multicast is the
907                                 same */
908          inb(RX_STATUS);      /* Clear status. */
909      }
910      else
911      {
912          outb(RX_NORM, RX_CMD);
913          inb(RX_STATUS);
914      }
915  }
916
917  #ifndef MODULE
918  static struct net_device dev_3c501 = {
919      init :      ell_probe,
920      base_addr:  0x280,
921      irq :      5,
922  };
923
924  static int io=0x280;
925  static int irq=5;
926  MODULE_PARM(io, "i");
927  MODULE_PARM(irq, "i");
928
929  /**
930   * init_module:
931   *
932   * When the driver is loaded as a module this function is called. We fake up
933   * a device structure with the base I/O and interrupt set as if it was being
934   * called from Space.c. This minimises the extra code that would otherwise
935   * be required.

```

Figure 121: Ethernet Driver 4 - multicast: linux/drivers/net/3c501.c


```

954  * and then free up the resources we took when the card was found.
955  */
956
957 void cleanup_module(void)
958 {
959     /*
960      *      No need to check MOD_IN_USE, as sys_delete_module() checks.
961      */
962
963     unregister_netdev(&dev_3c501);
964
965     /*
966      *      Free up the private structure, or leak memory :- )
967      */
968
969     kfree(dev_3c501.priv);
970     dev_3c501.priv = NULL; /* gets re-allocated by eli_probe1 */
971
972     /*
973      *      If we don't do this, we can't re-insmod it later.
974      */
975     release_region(dev_3c501.base_addr, EL1_IO_EXTENT);
976 }
977
978 #endif /* MODULE */
979
980 /*
981  * Local variables:
982  * compile-command: "gcc -D__KERNEL__ -Wall -Wstrict-prototypes -O6 -fomit-frame
983  *                  -pointer -m486 -c -o 3c501.o 3c501.c"
984  * kept-new-versions: 5
985  * End:
986  */

```

Figure 122: Ethernet Driver 4 - modules: linux/drivers/net/3c501.c

```

1  /*
2  * NET      An implementation of the SOCKET network access protocol.
3  *
4  * Version:  @(#)socket.c  1.1.93  18/02/95
5  *
6  * Authors:  Orest Zborowski, <obz@Kodak.COM>
7  *           Ross Biro, <bir7@leland.Stanford.Edu>
8  *           Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
9  *
10 * Fixes:
11 *         Anonymous      :      NOTSOCK/BADF cleanup. Error fix in
12 *                           shutdown()
13 *         Alan Cox       :      verify_area() fixes
14 *         Alan Cox       :      Removed DDI
15 *         Jonathan Kamens :      SOCK_DGRAM reconnect bug
16 *         Alan Cox       :      Moved a load of checks to the very
17 *                           top level.
18 *         Alan Cox       :      Move address structures to/from user
19 *                           mode above the protocol layers.
20 *         Rob Janssen    :      Allow 0 length sends.
21 *         Alan Cox       :      Asynchronous I/O support (cribbed from the
22 *                           tty drivers).
23 *         Niibe Yutaka   :      Asynchronous I/O for writes (4.4BSD style)
24 *         Jeff Uphoff    :      Made max number of sockets command-line
25 *                           configurable.
26 *         Matti Aarnio   :      Made the number of sockets dynamic,
27 *                           to be allocated when needed, and mr.
28 *                           Uphoff's max is used as max to be
29 *                           allowed to allocate.
30 *         Linus          :      Argh. removed all the socket allocation
31 *                           altogether: it's in the inode now.
32 *         Alan Cox       :      Made sock_alloc()/sock_release() public
33 *                           for NetROM and future kernel nfsd type
34 *                           stuff.
35 *         Alan Cox       :      sendmsg/recvmmsg basics.
36 *         Tom Dyas       :      Export net symbols.
37 *         Marcin Dalecki :      Fixed problems with CONFIG_NET="n".
38 *         Alan Cox       :      Added thread locking to sys_* calls
39 *                           for sockets. May have errors at the
40 *                           moment.
41 *         Kevin Buhr     :      Fixed the dumb errors in the above.
42 *         Andi Kleen     :      Some small cleanups, optimizations,
43 *                           and fixed a copy_from_user() bug.
44 *         Tigran Aivazian :      sys_send(args) calls sys_sendto(args, NULL
45 *                           , 0)
46 *         Tigran Aivazian :      Made listen(2) backlog sanity checks
47 *                           protocol-independent

```

Figure 123: Socket Interface Code - authors etc: linux/net/socket.c

```
48 *
49 *           This program is free software; you can redistribute it and/or
50 *           modify it under the terms of the GNU General Public License
51 *           as published by the Free Software Foundation; either version
52 *           2 of the License, or (at your option) any later version.
53 *
54 *
55 *           This module is effectively the top level interface to the BSD socket
56 *           paradigm.
57 *
58 */
```

Figure 124: Socket Interface Code - main copyright etc: `linux/net/socket.c`

```

89 static int sock_no_open(struct inode *irrelevant , struct file *dontcare);
90 static loff_t sock_lseek(struct file * file , loff_t offset , int whence);
91 static ssize_t sock_read(struct file * file , char *buf,
92                          size_t size , loff_t *ppos);
93 static ssize_t sock_write(struct file * file , const char *buf,
94                          size_t size , loff_t *ppos);
95 static int sock_mmap(struct file * file , struct vm_area_struct * vma);
96
97
98 static int sock_close(struct inode *inode, struct file * file );
99 static unsigned int sock_poll(struct file * file ,
100                              struct poll_table_struct *wait);
101 static int sock_ioctl(struct inode *inode, struct file * file ,
102                      unsigned int cmd, unsigned long arg);
103 static int sock_fasync(int fd, struct file * filp , int on);
104 static ssize_t sock_readv(struct file * file , const struct iovec *vector,
105                          unsigned long count, loff_t *ppos);
106 static ssize_t sock_writev(struct file * file , const struct iovec *vector,

```

Figure 125: Socket Interface Code - Prototypes: `linux/net/socket.c`

```

109         int offset , size_t size , loff_t *ppos, int more);
110
111
112  /*
113  *   Socket files have a set of 'special' operations as well as the generic
114  *   file ones. These don't appear
115  *   in the operation structures but are done directly via the socketcall()
116  *   multiplexor.
117  */
118  static struct file_operations socket_file_ops = {
119      llseek :      sock_llseek ,
120      read :       sock_read ,
121      write :      sock_write ,
122      poll :       sock_poll ,
123      ioctl :      sock_ioctl ,
124      mmap :       sock_mmap ,
125      open :       sock_no_open, /* special open code to disallow open via /
126      proc */
127      release :    sock_close ,
128      fasync :     sock_fasync ,

```

Figure 126: Socket Interface Code - file operation mappings etc:
linux/net/socket.c

```
128     writev:      sock_writev,  
129     sendpage:   sock_sendpage  
130 };  
131  
132 /*  
133  *   The protocol list. Each protocol is registered in here.  
134  */  
135  
136 static struct net_proto_family *net_families[NPROTO];
```

Figure 127: Socket Interface Code - Protocol Families supported:
linux/net/socket.c

```

138 #ifdef CONFIG_SMP
139 static atomic_t net_family_lockct = ATOMIC_INIT(0);
140 static spinlock_t net_family_lock = SPIN_LOCK_UNLOCKED;
141
142 /* The strategy is: modifications net_family vector are short, do not
143    sleep and veery rare, but read access should be free of any exclusive
144    locks.
145    */
146
147 static void net_family_write_lock(void)
148 {
149     spin_lock(&net_family_lock);
150     while (atomic_read(&net_family_lockct) != 0) {
151         spin_unlock(&net_family_lock);
152
153         current->policy |= SCHED_YIELD;
154         schedule();
155
156         spin_lock(&net_family_lock);
157     }
158 }
159
160 static __inline__ void net_family_write_unlock(void)
161 {
162     spin_unlock(&net_family_lock);
163 }
164
165 static __inline__ void net_family_read_lock(void)
166 {
167     atomic_inc(&net_family_lockct);
168     spin_unlock_wait(&net_family_lock);
169 }
170
171 static __inline__ void net_family_read_unlock(void)
172 {
173     atomic_dec(&net_family_lockct);
174 }
175
176 #else
177 #define net_family_write_lock() do { } while(0)

```

Figure 128: Socket Interface Code - lock code: `linux/net/socket.c`

```

180 #define net_family_read_unlock() do { } while(0)
181 #endif
182
183
184 /*
185  *   Statistics counters of the socket lists
186  */
187
188 static union {
189     int     counter;
190     char    _pad[SMP_CACHE_BYTES];
191 } sockets_in_use[NR_CPUS] __cacheline_aligned = {{0}};
192
193 /*
194  *   Support routines. Move socket addresses back and forth across the kernel/
195     user
196     divide and look after the messy bits.
197  */
198 #define MAX SOCK_ADDR 128          /* 108 for Unix domain -
199                                     16 for IP, 16 for IPX,
200                                     24 for IPv6,
201                                     about 80 for AX.25

```

Figure 129: Socket Interface Code - Global Counters and statistics:
linux/net/socket.c


```

203         the AF_UNIX size (see net/unix/af_unix.c
204         :unix_mkname()).
205     */
206
207 /**
208  * move_addr_to_kernel - copy a socket address into kernel space
209  * @uaddr: Address in user space
210  * @kaddr: Address in kernel space
211  * @ulen: Length in user space
212  *
213  * The address is copied into kernel space. If the provided address is
214  * too long an error code of -EINVAL is returned. If the copy gives
215  * invalid addresses -EFAULT is returned. On a success 0 is returned.
216  */
217
218 int move_addr_to_kernel(void *uaddr, int ulen, void *kaddr)
219 {
220     if (ulen < 0 || ulen > MAX_SOCKET_ADDR)
221         return -EINVAL;
222     if (ulen == 0)
223         return 0;

```

Figure 130: Check buffer addresses passed between user/kernel:
linux/net/socket.c

```

225         return -EFAULT;
226     return 0;
227 }
228
229 /**
230  *   move_addr_to_user      -      copy an address to user space
231  *   @kaddr: kernel space address
232  *   @klen: length of address in kernel
233  *   @uaddr: user space address
234  *   @ulen: pointer to user length field
235  *
236  *   The value pointed to by ulen on entry is the buffer length available.
237  *   This is overwritten with the buffer space used. -EINVAL is returned
238  *   if an overlong buffer is specified or a negative buffer size. -EFAULT
239  *   is returned if either the buffer or the length field are not
240  *   accessible.
241  *   After copying the data up to the limit the user specifies, the true
242  *   length of the data is written over the length limit the user
243  *   specified. Zero is returned for a success.
244  */
245
246 int move_addr_to_user(void *kaddr, int klen, void *uaddr, int *ulen)
247 {
248     int err;
249     int len;
250
251     if ((err=get_user(len, ulen)))
252         return err;
253     if (len>klen)
254         len=klen;
255     if (len<0 || len> MAX_SOCKET_ADDR)
256         return -EINVAL;
257     if (len)
258     {
259         if (copy_to_user(uaddr,kaddr,len))
260             return -EFAULT;
261     }
262     /*
263     *   "fromlen shall refer to the value before truncation.."

```

Figure 131: More legality checks: linux/net/socket.c

```
265     */
266     return __put_user(klen, ulen);
267 }
268
269 #define SOCKFS_MAGIC 0x534F434B
270 static int sockfs_statfs(struct super_block *sb, struct statfs *buf)
271 {
272     buf->f_type = SOCKFS_MAGIC;
273     buf->f_bsize = 1024;
274     buf->f_namelen = 255;
275     return 0;
276 }
```

Figure 132: Socket Glue - socket statfs: `linux/net/socket.c`

```

278 static struct super_operations sockfs_ops = {
279     statfs :      sockfs_statfs ,
280 };
281
282 static struct super_block * sockfs_read_super(struct super_block *sb, void *data, int
      silent )
283 {
284     struct inode *root = new_inode(sb);
285     if (!root)
286         return NULL;
287     root->i_mode = S_IFDIR | S_IRUSR | S_IWUSR;
288     root->i_uid = root->i_gid = 0;
289     root->i_atime = root->i_mtime = root->i_ctime = CURRENT_TIME;
290     sb->s_blocksize = 1024;
291     sb->s_blocksize_bits = 10;
292     sb->s_magic = SOCKFS_MAGIC;
293     sb->s_op      = &sockfs_ops;
294     sb->s_root = d_alloc(NULL, &(const struct qstr) { "socket:", 7, 0 });
295     if (!sb->s_root) {
296         iput(root);
297         return NULL;
298     }
299     sb->s_root->d_sb = sb;

```

Figure 133: Socket Glue - socket fs read: `linux/net/socket.c`

```
301     d_instantiate(sb->s_root, root);
302     return sb;
303 }
304
305 static struct vfsmount *sock_mnt;
306 static DECLARE_FSTYPE(sock_fs_type, "sockfs", sockfs_read_super,
307     FS_NOMOUNT|FS_SINGLE);
308 static int sockfs_delete_dentry (struct dentry *dentry)
309 {
310     return 1;
```

Figure 134: Socket Glue - socket fs mnt, delete, dentry: `linux/net/socket.c`

```
527     return size ;
528 }
529
530
531
532 /*
533  *   Sockets are not seekable.
534  */
```

Figure 135: Socket Glue - seek (and ye shall not find!): `linux/net/socket.c`

```
685
686     unlock_kernel();
687     sock = socki_lookup(inode);
688     err = sock->ops->ioctl(sock, cmd, arg);
689     lock_kernel();
```

Figure 136: Socket Glue - mmap: `linux/net/socket.c`

```
312 static struct dentry_operations sockfs_dentry_operations = {
313     d_delete:      sockfs_delete_dentry ,
314 };
315
316 /*
317  *   Obtains the first available file descriptor and sets it up for use.
318  *
319  *   This functions creates file structure and maps it to fd space
320  *   of current process. On success it returns file descriptor
321  *   and file struct implicitly stored in sock->file.
322  *   Note that another thread may close file descriptor before we return
323  *   from this function. We use the fact that now we do not refer
324  *   to socket after mapping. If one day we will need it, this
325  *   function will inincrement ref. count on file by 1.
326  *
327  *   In any case returned fd MAY BE not valid!
```

Figure 137: Socket Glue - find a spare file descriptor: `linux/net/socket.c`


```

329  *   with shared fd spaces, we cannot solve is inside kernel,
330  *   but we take care of internal coherence yet.
331  */
332
333  static int sock_map_fd(struct socket *sock)
334  {
335      int fd;
336      struct qstr this;
337      char name[32];
338
339      /*
340       *   Find a file descriptor suitable for return to the user.
341       */
342
343      fd = get_unused_fd();
344      if (fd >= 0) {
345          struct file *file = get_empty_filp();
346
347          if (!file) {
348              put_unused_fd(fd);
349              fd = -ENFILE;
350              goto out;
351          }
352
353          sprintf(name, "[%lu]", sock->inode->i_ino);
354          this.name = name;
355          this.len = strlen(name);
356          this.hash = sock->inode->i_ino;
357
358          file->f_dentry = d_alloc(sock->mnt->mnt_sb->s_root, &this);
359          if (!file->f_dentry) {
360              put_filp(file);

```

Figure 138: Socket Glue - get a file descriptor for a sock fd:
linux/net/socket.c

```

361         put_unused_fd(fd);
362         fd = -ENOMEM;
363         goto out;
364     }
365     file->f_dentry->d_op = &sockfs_dentry_operations;
366     d_add(file->f_dentry, sock->inode);
367     file->f_vfsmnt = mntget(sock_mnt);
368
369     sock->file = file;
370     file->f_op = sock->inode->i_fop = &socket_file_ops;
371     file->f_mode = 3;
372     file->f_flags = O_RDWR;
373     file->f_pos = 0;
374     fd_install(fd, file);
375 }

```

Figure 139: Socket Glue - get a file descriptor for a sock fd:
linux/net/socket.c

```
377 out:
378     return fd;
379 }
```

Figure 140: Socket Glue - lookup a file descriptor to a sock fd:
`linux/net/socket.c`

```

382 {
383     return &inode->u.socket_i;
384 }
385
386 /**
387  * sockfd_lookup - Go from a file number to its socket slot
388  * @fd: file handle
389  * @err: pointer to an error code return
390  *
391  * The file handle passed in is locked and the socket it is bound
392  * too is returned. If an error occurs the err pointer is overwritten
393  * with a negative errno code and NULL is returned. The function checks
394  * for both invalid handles and passing a handle which is not a socket.
395  *
396  * On a success the socket object pointer is returned.
397  */
398
399 struct socket *sockfd_lookup(int fd, int *err)
400 {
401     struct file *file ;
402     struct inode *inode;
403     struct socket *sock;
404
405     if (!(file = fget(fd)))
406     {
407         *err = -EBADF;
408         return NULL;
409     }
410
411     inode = file->f_dentry->d_inode;
412     if (!inode->i_sock || !(sock = socki_lookup(inode)))
413     {
414         *err = -ENOTSOCK;
415         fput(file);
416         return NULL;
417     }
418
419     if (sock->file != file) {
420         printk(KERN_ERR "socki_lookup: socket_file_changed!\n");

```

Figure 141: Socket Glue - lookup a file descriptor to a sock fd:
linux/net/socket.c

```

422     }
423     return sock;
424 }
425
426 extern __inline__ void sockfd_put(struct socket *sock)
427 {
428     fput(sock->file);
429 }
430
431 /**
432  *   sock_alloc   -   allocate a socket
433  *
434  *   Allocate a new inode and socket object. The two are bound together
435  *   and initialised. The socket is then returned. If we are out of inodes
436  *   NULL is returned.
437  */
438
439 struct socket * sock_alloc(void)
440 {
441     struct inode * inode;
442     struct socket * sock;
443
444     inode = get_empty_inode();
445     if (!inode)
446         return NULL;
447
448     inode->i_sb = sock_mnt->mnt_sb;
449     sock = socki_lookup(inode);
450
451     inode->i_mode = S_IFSOCK|S_IRWXUGO;
452     inode->i_sock = 1;
453     inode->i_uid = current->fsuid;
454     inode->i_gid = current->fsgid;
455
456     sock->inode = inode;
457     init_waitqueue_head(&sock->wait);
458     sock->fasync_list = NULL;
459     sock->state = SS_UNCONNECTED;
460     sock->flags = 0;
461     sock->ops = NULL;
462     sock->sk = NULL;
463     sock->file = NULL;

```

Figure 142: Socket Glue - allocate a socket *inode*: linux/net/socket.c

```

476 {
477     return -ENXIO;
478 }
479
480 /**
481  * sock_release - close a socket
482  * @sock: socket to close
483  *
484  * The socket is released from the protocol stack if it has a release
485  * callback, and the inode is then released if the socket is bound to
486  * an inode not a file.
487  */
488
489 void sock_release(struct socket *sock)
490 {
491     if (sock->ops)
492         sock->ops->release(sock);
493
494     if (sock->fasync_list)
495         printk(KERN_ERR "sock_release: fasync_list not empty!\n");
496
497     sockets_in_use[smp_processor_id()].counter--;
498     if (!sock->file) {
499         iput(sock->inode);

```

Figure 143: Socket Glue - free a socket *inode*: linux/net/socket.c

```
465     sockets_in_use[smp_processor_id()].counter++;
466     return sock;
467 }
468
469 /*
470  * In theory you can't get an open on this inode, but /proc provides
471  * a back door. Remember to keep it shut otherwise you'll let the
472  * creepy crawlies in.
473  */
```

Figure 144: Socket Glue - block “opening” a socket *inode*: `linux/net/socket.c`

```

1305     }
1306     return err;
1307 }
1308
1309 /*
1310  *   Get a socket option. Because we don't know the option lengths we have
1311  *   to pass a user mode parameter for the protocols to sort out.
1312  */
1313
1314 asmlinkage long sys_getsockopt(int fd, int level, int optname, char *optval, int *optlen)
1315 {
1316     int err;
1317     struct socket *sock;
1318
1319     if ((sock = sockfd_lookup(fd, &err))!=NULL)
1320     {
1321         if ( level == SOL_SOCKET)

```

Figure 145: Socket Glue - shutdown: `linux/net/socket.c`


```

501     }
502     sock->file=NULL;
503 }
504
505 int sock_sendmsg(struct socket *sock, struct msghdr *msg, int size)
506 {
507     int err;
508     struct scm_cookie scm;
509
510     err = scm_send(sock, msg, &scm);
511     if (err >= 0) {
512         err = sock->ops->sendmsg(sock, msg, size, &scm);
513         scm_destroy(&scm);
514     }
515     return err;
516 }
517
518 int sock_recvmsg(struct socket *sock, struct msghdr *msg, int size, int flags)
519 {
520     struct scm_cookie scm;
521
522     memset(&scm, 0, sizeof(scm));
523
524     size = sock->ops->recvmsg(sock, msg, size, flags, &scm);
525     if (size >= 0)

```

Figure 146: Socket Glue - send and receive messages on a socket:
linux/net/socket.c

```

537 {
538     return -ESPIPE;
539 }
540
541 /*
542  *   Read data from a socket. ubuf is a user mode pointer. We make sure the
543  *   user
544  *   area ubuf...ubuf+size-1 is writable before asking the protocol.
545  */
546 static ssize_t sock_read(struct file * file , char *ubuf,
547                          size_t size , loff_t *ppos)
548 {
549     struct socket *sock;
550     struct iovec iov;
551     struct msghdr msg;
552     int flags;
553
554     if ( ppos != &file->f_pos)
555         return -ESPIPE;
556     if ( size==0)          /* Match SYS5 behaviour */
557         return 0;
558
559     sock = socki_lookup( file ->f_dentry->d_inode);
560
561     msg.msg_name=NULL;
562     msg.msg_namelen=0;
563     msg.msg_iov=&iov;
564     msg.msg_iovlen=1;
565     msg.msg_control=NULL;
566     msg.msg_controllen=0;
567     iov.iov_base=ubuf;
568     iov.iov_len=size;
569     flags = !( file ->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;

```

Figure 147: Socket Glue - read: linux/net/socket.c

```

571     return sock_recvmsg(sock, &msg, size, flags);
572 }
573
574
575 /*
576  * Write data to a socket. We verify that the user area ubuf..ubuf+size-1
577  * is readable by the user process.
578  */
579
580 static ssize_t sock_write(struct file *file, const char *ubuf,
581                          size_t size, loff_t *ppos)
582 {
583     struct socket *sock;
584     struct msghdr msg;
585     struct iovec iov;
586
587     if (ppos != &file->f_pos)
588         return -ESPIPE;
589     if (size==0) /* Match SYS5 behaviour */
590         return 0;
591
592     sock = socki_lookup(file->f_dentry->d_inode);
593
594     msg.msg_name=NULL;
595     msg.msg_namelen=0;
596     msg.msg_iov=&iov;
597     msg.msg_iovlen=1;
598     msg.msg_control=NULL;
599     msg.msg_controllen=0;
600     msg.msg_flags=(file->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;
601     if (sock->type == SOCK_SEQPACKET)
602         msg.msg_flags |= MSG_EOR;
603     iov.iov_base=(void *)ubuf;

```

Figure 148: Socket Glue - write: `linux/net/socket.c`

```

605     return sock_sendmsg(sock, &msg, size);
606 }
607
608 ssize_t sock_sendpage(struct file *file, struct page *page,
609                      int offset, size_t size, loff_t *ppos, int more)
610 {
611     struct socket *sock;
612     int flags;
613
614     if (ppos != &file->f_pos)
615         return -ESPIPE;
616
617     sock = socki_lookup(file->f_dentry->d_inode);
618
619     flags = !(file->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;
620     if (more)
621         flags |= MSG_MORE;
622
623     return sock->ops->sendpage(sock, page, offset, size, flags);
624 }
625
626 int sock_readv_writev(int type, struct inode *inode, struct file *file,
627                      const struct iovec *iov, long count, long size)
628 {
629

```

Figure 149: Socket Glue - readv/writev service routine: `linux/net/socket.c`

```

631     struct socket *sock;
632
633     sock = socki_lookup(inode);
634
635     msg.msg_name = NULL;
636     msg.msg_namelen = 0;
637     msg.msg_control = NULL;
638     msg.msg_controllen = 0;
639     msg.msg_iov = (struct iovec *) iov;
640     msg.msg_iovlen = count;
641     msg.msg_flags = (file->f_flags & O_NONBLOCK) ? MSG_DONTWAIT : 0;
642
643     /* read() does a VERIFY_WRITE */
644     if (type == VERIFY_WRITE)
645         return sock_recvmsg(sock, &msg, size, msg.msg_flags);
646
647     if (sock->type == SOCK_SEQPACKET)
648         msg.msg_flags |= MSG_EOR;
649
650     return sock_sendmsg(sock, &msg, size);
651 }

```

Figure 150: Socket Glue - readv and writev: `linux/net/socket.c`

```

1159
1160 /*
1161 *   Get the remote address ('name') of a socket object. Move the obtained
1162 *   name to user space.
1163 */
1164
1165 asmlinkage long sys_getpeername(int fd, struct sockaddr *usockaddr, int *usockaddr_len)
1166 {
1167     struct socket *sock;
1168     char address[MAX_SOCKET_ADDR];
1169     int len, err;
1170
1171     if ((sock = sockfd_lookup(fd, &err)) != NULL)
1172     {
1173         err = sock->ops->getname(sock, (struct sockaddr *)address, &len, 1);
1174         if (!err)
1175             err = move_addr_to_user(address, len, usockaddr, usockaddr_len);
1176         sockfd_put(sock);
1177     }
1178     return err;
1179 }
1180
1181 /*
1182 *   Send a datagram to a given address. We move the address into kernel
1183 *   space and check the user space data area is readable before invoking
1184 *   the protocol.

```

Figure 151: Socket Glue - sendto i: linux/net/socket.c

```

1185  */
1186
1187  asmlinkage long sys_sendto(int fd, void * buff, size_t len, unsigned flags,
1188                          struct sockaddr *addr, int addr_len)
1189  {
1190      struct socket *sock;
1191      char address[MAX_SOCKET_ADDR];
1192      int err;
1193      struct msghdr msg;
1194      struct iovec iov;
1195
1196      sock = sockfd_lookup(fd, &err);
1197      if (!sock)
1198          goto out;
1199      iov.iov_base = buff;
1200      iov.iov_len = len;
1201      msg.msg_name = NULL;
1202      msg.msg_iov = &iov;
1203      msg.msg_iovlen = 1;
1204      msg.msg_control = NULL;
1205      msg.msg_controllen = 0;
1206      msg.msg_namelen = addr_len;
1207      if (addr)
1208      {
1209          err = move_addr_to_kernel(addr, addr_len, address);
1210          if (err < 0)

```

Figure 152: Socket Glue - sendto ii: `linux/net/socket.c`

```

1212         msg.msg_name=address;
1213     }
1214     if (sock->file->f_flags & O_NONBLOCK)
1215         flags |= MSG_DONTWAIT;
1216     msg.msg_flags = flags;
1217     err = sock_sendmsg(sock, &msg, len);
1218
1219 out_put:
1220     sockfd_put(sock);
1221 out:
1222     return err;
1223 }
1224
1225 /*
1226  *   Send a datagram down a socket.
1227  */
1228
1229 asmlinkage long sys_send(int fd, void * buff, size_t len, unsigned flags)

```

Figure 153: Socket Glue - recvfrom: `linux/net/socket.c`


```

1231     return sys_sendto(fd, buff, len, flags, NULL, 0);
1232 }
1233
1234 /*
1235  *   Receive a frame from the socket and optionally record the address of the
1236  *   sender. We verify the buffers are writable and if needed move the
1237  *   sender address from kernel to user space.
1238  */
1239
1240 asmlinkage long sys_recvfrom(int fd, void * ubuf, size_t size, unsigned flags,
1241                             struct sockaddr *addr, int *addr_len)
1242 {
1243     struct socket *sock;
1244     struct iovec iov;
1245     struct msghdr msg;
1246     char address[MAX_SOCKET_ADDR];
1247     int err, err2;
1248
1249     sock = sockfd_lookup(fd, &err);
1250     if (!sock)
1251         goto out;

```

Figure 154: Socket Glue - recvfrom: `linux/net/socket.c`

```
1254     msg.msg_controllen=0;
1255     msg.msg_iovlen=1;
1256     msg.msg_iov=&iov;
1257     iov.iov_len=size;
1258     iov.iov_base=ubuf;
1259     msg.msg_name=address;
1260     msg.msg_namelen=MAX_SOCKET_ADDR;
1261     if (sock->file->f_flags & O_NONBLOCK)
```

Figure 155: Socket Glue - recvfrom: `linux/net/socket.c`

```

653 static ssize_t sock_readv(struct file *file , const struct iovec *vector,
654                          unsigned long count, loff_t *ppos)
655 {
656     size_t tot_len = 0;
657     int i;
658     for (i = 0 ; i < count ; i++)
659         tot_len += vector[i].iov_len;
660     return sock_readv_writev(VERIFY_WRITE, file->f_dentry->d_inode,
661                             file , vector, count, tot_len);
662 }
663
664 static ssize_t sock_writev(struct file *file , const struct iovec *vector,
665                          unsigned long count, loff_t *ppos)
666 {
667     size_t tot_len = 0;
668     int i;
669     for (i = 0 ; i < count ; i++)
670         tot_len += vector[i].iov_len;

```

Figure 156: Socket Glue - ioctl: `linux/net/socket.c`

```
673 }
674
675 /*
676 *   With an ioctl arg may well be a user mode pointer, but we don't know what
        to do
677 *   with it - that's up to the protocol still.
678 */
679
680 int sock_ioctl(struct inode *inode, struct file *file , unsigned int cmd,
681               unsigned long arg)
682 {
683     struct socket *sock;
```

Figure 157: Socket Glue - poll: `linux/net/socket.c`

```

692 }
693
694
695 /* No kernel lock held - perfect */
696 static unsigned int sock_poll(struct file * file , poll_table * wait)
697 {
698     struct socket *sock;
699
700     /*
701      *   We can't return errors to poll, so it's either yes or no.
702      */
703     sock = socki_lookup(file->f_dentry->d_inode);
704     return sock->ops->poll(file, sock, wait);
705 }
706
707 static int sock_mmap(struct file * file , struct vm_area_struct * vma)

```

Figure 158: Socket Glue - close: `linux/net/socket.c`

```

709     struct socket *sock = socki_lookup(file->f_dentry->d_inode);
710
711     return sock->ops->mmap(file, sock, vma);
712 }
713
714 int sock_close(struct inode *inode, struct file *filp)
715 {
716     /*
717      *   It was possible the inode is NULL we were
718      *   closing an unfinished socket.
719      */
720
721     if (!inode)
722     {

```

Figure 159: Socket Glue - asynch events on sockets i: `linux/net/socket.c`

```

724         return 0;
725     }
726     sock_fasync(-1, filp, 0);
727     sock_release(socki_lookup(inode));
728     return 0;
729 }
730
731 /*
732  *   Update the socket async list
733  *
734  *   Fasync_list locking strategy.
735  *
736  *   1. fasync_list is modified only under process context socket lock
737  *      i.e. under semaphore.
738  *   2. fasync_list is used under read_lock(&sk->callback_lock)
739  *      or under socket lock.
740  *   3. fasync_list can be used from softirq context, so that
741  *      modification under socket lock have to be enhanced with
742  *      write_lock_bh(&sk->callback_lock).
743  *
744  *                                     --ANK (990710)
745  */
746 static int sock_fasync(int fd, struct file *filp, int on)
747 {
748     struct fasync_struct *fa, *fna=NULL, **prev;
749     struct socket *sock;

```

Figure 160: Socket Glue - asynch events on sockets, ii: linux/net/socket.c

```

751     if (on)
752     {
753         fna=(struct fasync_struct *)kmalloc(sizeof(struct fasync_struct),
754             GFP_KERNEL);
755         if (fna==NULL)
756             return -ENOMEM;
757     }
758
759     sock = socki_lookup(filp->f_dentry->d_inode);
760
761     if ((sk=sock->sk) == NULL)
762         return -EINVAL;
763
764     lock_sock(sk);
765
766     prev=&(sock->fasync_list);
767
768     for (fa=*prev; fa!=NULL; prev=&fa->fa_next,fa=*prev)
769

```

Figure 161: Socket Glue - asynch events on sockets, iii: `linux/net/socket.c`


```

770         if (fa->fa_file==filp)
771             break;
772
773     if (on)
774     {
775         if (fa!=NULL)
776         {
777             write_lock_bh(&sk->callback_lock);
778             fa->fa_fd=fd;
779             write_unlock_bh(&sk->callback_lock);
780
781             kfree(fna);
782             goto out;
783         }
784         fna->fa_file=filp;

```

Figure 162: Socket Glue - asynch events on sockets else off, ii:
linux/net/socket.c

```

786     fna->magic=FASYNC_MAGIC;
787     fna->fa_next=sock->fasync_list;
788     write_lock_bh(&sk->callback_lock);
789     sock->fasync_list=fna;
790     write_unlock_bh(&sk->callback_lock);
791 }
792 else
793 {
794     if (fa!=NULL)
795     {
796         write_lock_bh(&sk->callback_lock);
797         *prev=fa->fa_next;
798         write_unlock_bh(&sk->callback_lock);
799         kfree(fa);
800     }
801 }
802
803 out:
804     release_sock(sock->sk);
805     return 0;
806 }
807
808 /* This function may be called only under socket lock or callback_lock */
809
810 int sock_wake_async(struct socket *sock, int how, int band)
811 {

```

Figure 163: Socket Glue - finish asynch events on sockets: `linux/net/socket.c`

```

814     switch (how)
815     {
816     case 1:
817
818         if ( test_bit (SOCK_ASYNC_WAITDATA, &sock->flags))
819             break;
820         goto call_kill ;
821     case 2:
822         if (! test_and_clear_bit (SOCK_ASYNC_NOSPACE, &sock->flags))
823             break;
824         /* fall through */
825     case 0:
826         call_kill :
827             _kill_fasync (sock->fasync_list, SIGIO, band);
828             break;
829     case 3:
830         _kill_fasync (sock->fasync_list, SIGURG, band);
831     }
832     return 0;
833 }
834
835
836 int sock_create(int family, int type, int protocol, struct socket **res)
837 {

```

Figure 164: Socket Glue - create a socket i: `linux/net/socket.c`

```
840
841     /*
842     *     Check protocol is in range
843     */
844     if (family < 0 || family >= NPROTO)
845         return -EAFNOSUPPORT;
846
847     /* Compatibility.
848
849     This uglomoron is moved from INET layer to here to avoid
850     deadlock in module load.
851     */
```

Figure 165: Socket Glue - create a socket ii: `linux/net/socket.c`

```

860
861 #if defined(CONFIG_KMOD) && defined(CONFIG_NET)
862     /* Attempt to load a protocol module if the find failed.
863     *
864     * 12/09/1996 Marcin: But! this makes REALLY only sense, if the user
865     * requested real, full-featured networking support upon configuration.
866     * Otherwise module support will break!
867     */
868     if ( net_families [family]==NULL)
869     {
870         char module_name[30];
871         sprintf (module_name,"net-pf-%d",family);
872         request_module(module_name);
873     }
874 #endif
875
876     net_family_read_lock ();
877     if ( net_families [family] == NULL) {
878         i = -EAFNOSUPPORT;
879         goto out;
880     }
881
882 /*
883 * Allocate the socket and allow the family to set things up. if
884 * the protocol is 0, the family is instructed to select an appropriate
885 * default.
886 */

```

Figure 166: Socket Glue - create a socket: linux/net/socket.c

```

889     {
890         printk(KERN_WARNING "socket: no more sockets\n");
891         i = -ENFILE;          /* Not exactly a match, but its the
892                               closest posix thing */
893         goto out;
894     }
895
896     sock->type = type;
897
898     if ((i = net_families[family]->create(sock, protocol)) < 0)
899     {
900         sock_release(sock);
901         goto out;
902     }
903
904     *res = sock;
905
906 out:
907     net_family_read_unlock();
908     return i;
909 }

```

Figure 167: Socket Glue - asm glue to create a socket: `linux/net/socket.c`

```

911 asmlinkage long sys_socket(int family, int type, int protocol)
912 {
913     int retval;
914     struct socket *sock;
915
916     retval = sock_create(family, type, protocol, &sock);
917     if (retval < 0)
918         goto out;
919
920     retval = sock_map_fd(sock);
921     if (retval < 0)
922         goto out_release;
923
924 out:
925     /* It may be already another descriptor 8) Not kernel problem. */
926     return retval;
927
928 out_release:
929     sock_release(sock);
930     return retval;
931 }
932
933 /*
934  *   Create a pair of connected sockets.
935  */
936
937 asmlinkage long sys_socketpair(int family, int type, int protocol, int usockvec[2])

```

Figure 168: Socket Glue - asm glue to create a socket pair: `linux/net/socket.c`

```

939     struct socket *sock1, *sock2;
940     int fd1, fd2, err;
941
942     /*
943     * Obtain the first socket and check if the underlying protocol
944     * supports the socketpair call.
945     */
946
947     err = sock_create(family, type, protocol, &sock1);
948     if (err < 0)
949         goto out;
950
951     err = sock_create(family, type, protocol, &sock2);
952     if (err < 0)
953         goto out_release_1;
954
955     err = sock1->ops->socketpair(sock1, sock2);
956     if (err < 0)
957         goto out_release_both;
958
959     fd1 = fd2 = -1;
960
961     err = sock_map_fd(sock1);
962     if (err < 0)
963         goto out_release_both;
964     fd1 = err;
965
966     err = sock_map_fd(sock2);
967     if (err < 0)
968         goto out_close_1;
969     fd2 = err;
970
971     /* fd1 and fd2 may be already another descriptors.
972     * Not kernel problem.
973     */

```

Figure 169: Socket Glue - ... to create a socket pair: `linux/net/socket.c`


```
977         err = put_user(fd2, &usockvec[1]);
978         if (!err)
979             return 0;
980
981         sys_close(fd2);
982         sys_close(fd1);
983         return err;
984
985     out_close_1:
986         sock_release(sock2);
987         sys_close(fd1);
988         return err;
989
990     out_release_both:
991         sock_release(sock2);
992     out_release_1:
993         sock_release(sock1);
994     out:
995         return err;
996 }
```

Figure 170: Socket Glue - bind a socket: `linux/net/socket.c`

```

1001 *   the protocol's responsibility to handle the local address.
1002 *
1003 *   We move the socket address to kernel space before we call
1004 *   the protocol layer (having also checked the address is ok).
1005 */
1006
1007 asmlinkage long sys_bind(int fd, struct sockaddr *umyaddr, int addrlen)
1008 {
1009     struct socket *sock;
1010     char address[MAX_SOCKET_ADDR];
1011     int err;
1012
1013     if ((sock = sockfd_lookup(fd, &err)) != NULL)
1014     {
1015         if ((err = move_addr_to_kernel(umyaddr, addrlen, address)) >= 0)
1016             err = sock->ops->bind(sock, (struct sockaddr *)address, addrlen);
1017         sockfd_put(sock);
1018     }
1019     return err;

```

Figure 171: Socket Glue - listen on a socket: `linux/net/socket.c`

```
1022
1023 /*
1024 *   Perform a listen. Basically, we allow the protocol to do anything
1025 *   necessary for a listen, and if that works, we mark the socket as
1026 *   ready for listening.
1027 */
1028
1029 asmlinkage long sys_listen (int fd, int backlog)
1030 {
1031     struct socket *sock;
1032     int err;
```

Figure 172: Socket Glue - accept on a socket: `linux/net/socket.c`

```

1034     if ((sock = sockfd_lookup(fd, &err)) != NULL) {
1035         if ((unsigned) backlog > SOMAXCONN)
1036             backlog = SOMAXCONN;
1037         err=sock->ops->listen(sock, backlog);
1038         sockfd_put(sock);
1039     }
1040     return err;
1041 }
1042
1043
1044 /*
1045  * For accept, we attempt to create a new socket, set up the link
1046  * with the client, wakeup the client, then return the new
1047  * connected fd. We collect the address of the connector in kernel
1048  * space and move it to user at the very end. This is unclean because
1049  * we open the socket then return an error.
1050  *
1051  * 1003.1g adds the ability to recvmsg() to query connection pending
1052  * status to recvmsg. We need to add that support in a way thats
1053  * clean when we restructure accept also.

```

Figure 173: Socket Glue - accept on a socket i: linux/net/socket.c

```

1055 asmlinkage long sys_accept(int fd, struct sockaddr *upeer_sockaddr, int *upeer_addrlen)
1056 {
1057     struct socket *sock, *newsock;
1058     int err, len;
1059     char address[MAX_SOCKET_ADDR];
1060
1061     sock = sockfd_lookup(fd, &err);
1062     if (!sock)
1063         goto out;
1064
1065     err = -EMFILE;
1066     if (!(newsock = sock_alloc()))
1067         goto out_put;
1068
1069     newsock->type = sock->type;
1070     newsock->ops = sock->ops;
1071
1072     err = sock->ops->accept(sock, newsock, sock->file->f_flags);
1073     if (err < 0)
1074         goto out_release;
1075
1076     if (upeer_sockaddr) {
1077         if (newsock->ops->getname(newsock, (struct sockaddr *)address, &len
1078             , 2) < 0) {

```

Figure 174: Socket Glue - accept on a socket ii: `linux/net/socket.c`

```

1081     }
1082     err = move_addr_to_user(address, len, upeer_sockaddr, upeer_addrlen);
1083     if (err < 0)
1084         goto out_release;
1085     }
1086
1087     /* File flags are not inherited via accept() unlike another OSes. */
1088
1089     if ((err = sock_map_fd(newsock)) < 0)
1090         goto out_release;
1091
1092 out_put:
1093     sockfd_put(sock);
1094 out:
1095     return err;
1096
1097 out_release:
1098     sock_release(newsock);
1099     goto out_put;
1100 }
1101
1102
1103 /*
1104 *   Attempt to connect to a socket with the server address. The address
1105 *   is in user space so we verify it is OK and move it to kernel space.
1106 *
1107 *   For 1003.1g we need to add clean support for a bind to AF_UNSPEC to
1108 *   break bindings
1109 *
1110 *   NOTE: 1003.1g draft 6.3 is broken with respect to AX.25/NetROM and
1111 *   other SEQPACKET protocols that take time to connect() as it doesn't

```

Figure 175: Socket Glue - connect on a socket: linux/net/socket.c

```

1113  */
1114
1115  asmlinkage long sys_connect(int fd, struct sockaddr *useraddr, int addrlen)
1116  {
1117      struct socket *sock;
1118      char address[MAX_SOCKET_ADDR];
1119      int err;
1120
1121      sock = sockfd_lookup(fd, &err);
1122      if (!sock)
1123          goto out;
1124      err = move_addr_to_kernel(useraddr, addrlen, address);
1125      if (err < 0)
1126          goto out_put;
1127      err = sock->ops->connect(sock, (struct sockaddr *) address, addrlen,
1128                             sock->file->f_flags);
1129  out_put:
1130      sockfd_put(sock);
1131  out:
1132      return err;
1133  }
1134
1135  /*
1136  *   Get the local address ('name') of a socket object. Move the obtained

```

Figure 176: Socket Glue - get the local name (“address”) of a socket:
linux/net/socket.c

```

1138  */
1139
1140  asmlinkage long sys_getsockname(int fd, struct sockaddr *usockaddr, int *usockaddr_len)
1141  {
1142      struct socket *sock;
1143      char address[MAX_SOCKET_ADDR];
1144      int len, err;
1145
1146      sock = sockfd_lookup(fd, &err);
1147      if (!sock)
1148          goto out;
1149      err = sock->ops->getname(sock, (struct sockaddr *)address, &len, 0);
1150      if (err)
1151          goto out_put;
1152      err = move_addr_to_user(address, len, usockaddr, usockaddr_len);
1153
1154  out_put:
1155      sockfd_put(sock);
1156  out:
1157      return err;

```

Figure 177: Socket Glue - get the far end's name ("address") for a socket:
linux/net/socket.c


```

1263     err=sock_recvmsg(sock, &msg, size, flags);
1264
1265     if (err >= 0 && addr != NULL && msg.msg_namelen)
1266     {
1267         err2=move_addr_to_user(address, msg.msg_namelen, addr, addr_len);
1268         if (err2<0)
1269             err=err2;
1270     }
1271     sockfd_put(sock);
1272 out:
1273     return err;
1274 }
1275
1276 /*
1277  *   Receive a datagram from a socket.
1278  */
1279
1280 asmlinkage long sys_recv(int fd, void * ubuf, size_t size, unsigned flags)
1281 {
1282     return sys_recvfrom(fd, ubuf, size, flags, NULL, NULL);

```

Figure 178: Socket Glue - setsockopt: `linux/net/socket.c`

```

1284
1285 /*
1286 *   Set a socket option. Because we don't know the option lengths we have
1287 *   to pass the user mode parameter for the protocols to sort out.
1288 */
1289
1290 asmlinkage long sys_setsockopt(int fd, int level, int optname, char *optval, int optlen)
1291 {
1292     int err;
1293     struct socket *sock;
1294
1295     if (optlen < 0)
1296         return -EINVAL;
1297
1298     if ((sock = sockfd_lookup(fd, &err))!=NULL)
1299     {
1300         if (level == SOL_SOCKET)
1301             err=sock_setsockopt(sock, level, optname, optval, optlen);
1302         else
1303             err=sock->ops->setsockopt(sock, level, optname, optval, optlen);

```

Figure 179: Socket Glue - getsockopt: `linux/net/socket.c`

```

1323         else
1324             err=sock->ops->getsockopt(sock, level, optname, optval, optlen);
1325             sockfd_put(sock);
1326     }
1327     return err;
1328 }
1329
1330
1331 /*
1332  *   Shutdown a socket.
1333  */
1334
1335 asmlinkage long sys_shutdown(int fd, int how)
1336 {
1337     int err;
1338     struct socket *sock;
1339
1340     if ((sock = sockfd_lookup(fd, &err))!=NULL)
1341     {
1342         err=sock->ops->shutdown(sock, how);
1343         sockfd_put(sock);
1344     }
1345     return err;
1346 }
1347
1348 /*
1349  *   BSD sendmsg interface
1350  */
1351
1352 asmlinkage long sys_sendmsg(int fd, struct msghdr *msg, unsigned flags)
1353 {
1354     struct socket *sock;
1355     char address[MAX_SOCKET_ADDR];
1356     struct iovec iovstack[UIO_FASTIOV], *iov = iovstack;
1357     unsigned char ctl[ sizeof(struct cmsghdr) + 20]; /* 20 is size of ipv6_pktinfo

```

Figure 180: Socket Glue - sendmsg i: `linux/net/socket.c`

```

1359     struct msghdr msg_sys;
1360     int err, ctl_len, iov_size, total_len;
1361
1362     err = -EFAULT;
1363     if (copy_from_user(&msg_sys, msg, sizeof(struct msghdr)))
1364         goto out;
1365
1366     sock = sockfd_lookup(fd, &err);
1367     if (!sock)
1368         goto out;
1369
1370     /* do not move before msg_sys is valid */
1371     err = -EINVAL;
1372     if (msg_sys.msg_iovlen > UIO_MAXIOV)
1373         goto out_put;
1374
1375     /* Check whether to allocate the iovec area */
1376     err = -ENOMEM;
1377     iov_size = msg_sys.msg_iovlen * sizeof(struct iovec);
1378     if (msg_sys.msg_iovlen > UIO_FASTIOV) {
1379         iov = sock_kmalloc(sock->sk, iov_size, GFP_KERNEL);
1380         if (!iov)
1381             goto out_put;
1382     }
1383
1384     /* This will also move the address data into kernel space */
1385     err = verify_iovec(&msg_sys, iov, address, VERIFY_READ);
1386     if (err < 0)
1387         goto out_freeiov;
1388     total_len = err;
1389
1390     err = -ENOBUFS;
1391
1392     if (msg_sys.msg_controllen > INT_MAX)
1393         goto out_freeiov;
1394     ctl_len = msg_sys.msg_controllen;
1395     if (ctl_len)
1396     {
1397         if (ctl_len > sizeof(ctl))
1398         {
1399             ctl_buf = sock_kmalloc(sock->sk, ctl_len, GFP_KERNEL);

```

Figure 181: Socket Glue - sendmsg ii: linux/net/socket.c

```

1401         goto out_freeiov ;
1402     }
1403     err = -EFAULT;
1404     if (copy_from_user(ctl_buf, msg_sys.msg_control, ctl_len))
1405         goto out_freectl ;
1406     msg_sys.msg_control = ctl_buf;
1407 }
1408 msg_sys.msg_flags = flags;
1409
1410 if (sock->file->f_flags & O_NONBLOCK)
1411     msg_sys.msg_flags |= MSG_DONTWAIT;
1412 err = sock_sendmsg(sock, &msg_sys, total_len);
1413
1414 out_freectl :
1415     if (ctl_buf != ctl)
1416         sock_kfree_s(sock->sk, ctl_buf, ctl_len);
1417 out_freeiov :
1418     if (iov != iovstack)
1419         sock_kfree_s(sock->sk, iov, iov_size);
1420 out_put :
1421     sockfd_put(sock);
1422 out :
1423     return err;
1424 }
1425
1426 /*
1427  *   BSD recvmsg interface
1428  */
1429
1430 asmlinkage long sys_recvmsg(int fd, struct msghdr *msg, unsigned int flags)
1431 {
1432     struct socket *sock;
1433     struct iovec iovstack[UIO_FASTIOV];
1434     struct iovec *iov=iovstack;
1435     struct msghdr msg_sys;
1436     unsigned long cmsg_ptr;
1437     int err, iov_size, total_len, len;
1438
1439     /* kernel mode address */
1440     char addr[MAX_SOCKET_ADDR];

```

Figure 182: Socket Glue - recvmsg i: linux/net/socket.c

```

1442     /* user mode address pointers */
1443     struct sockaddr *uaddr;
1444     int *uaddr_len;
1445
1446     err = -EFAULT;
1447     if (copy_from_user(&msg_sys, msg, sizeof(struct msghdr)))
1448         goto out;
1449
1450     sock = sockfd_lookup(fd, &err);
1451     if (!sock)
1452         goto out;
1453
1454     err = -EINVAL;
1455     if (msg_sys.msg_iovlen > UIO_MAXIOV)
1456         goto out_put;
1457
1458     /* Check whether to allocate the iovec area*/
1459     err = -ENOMEM;
1460     iov_size = msg_sys.msg_iovlen * sizeof(struct iovec);
1461     if (msg_sys.msg_iovlen > UIO_FASTIOV) {
1462         iov = sock_kmalloc(sock->sk, iov_size, GFP_KERNEL);
1463         if (!iov)
1464             goto out_put;
1465     }
1466
1467     /*
1468     *   Save the user-mode address (verify_iovec will change the
1469     *   kernel msghdr to use the kernel address space)
1470     */
1471
1472     uaddr = msg_sys.msg_name;
1473     uaddr_len = &msg->msg_namelen;
1474     err = verify_iovec(&msg_sys, iov, addr, VERIFY_WRITE);
1475     if (err < 0)
1476         goto out_freeiov;
1477     total_len = err;
1478
1479     cmsg_ptr = (unsigned long)msg_sys.msg_control;
1480     msg_sys.msg_flags = 0;
1481
1482     if (sock->file->f_flags & O_NONBLOCK)
1483         flags |= MSG_DONTWAIT;
1484     err = sock_recvmsg(sock, &msg_sys, total_len, flags);
1485     if (err < 0)

```

Figure 183: Socket Glue - recvmsg ii: linux/net/socket.c

```

1488
1489     if (uaddr != NULL && msg_sys.msg_namelen) {
1490         err = move_addr_to_user(addr, msg_sys.msg_namelen, uaddr, uaddr_len);
1491         if (err < 0)
1492             goto out_freeiov ;
1493     }
1494     err = _put_user(msg_sys.msg_flags, &msg->msg_flags);
1495     if (err)
1496         goto out_freeiov ;
1497     err = _put_user(((unsigned long)msg_sys.msg_control - cmsg_ptr,
1498                    &msg->msg_controllen));
1499     if (err)
1500         goto out_freeiov ;
1501     err = len;
1502
1503 out_freeiov :
1504     if (iov != iovstack)
1505         sock_kfree_s(sock->sk, iov, iov_size);
1506 out_put :
1507     sockfd_put(sock);
1508 out :
1509     return err;
1510 }

```

Figure 184: Socket Glue - file control (fcntl): `linux/net/socket.c`

```

1512
1513 /*
1514 *   Perform a file control on a socket file descriptor.
1515 *
1516 *   Doesn't acquire a fd lock, because no network fcntl
1517 *   function sleeps currently.
1518 */
1519
1520 int sock_fcntl(struct file *filp, unsigned int cmd, unsigned long arg)
1521 {
1522     struct socket *sock;
1523
1524     sock = socki_lookup(filp->f_dentry->d_inode);
1525     if (sock && sock->ops)
1526         return sock_no_fcntl(sock, cmd, arg);
1527     return(-EINVAL);
1528 }
1529
1530 /* Argument list sizes for sys_socketcall */
1531 #define AL(x) ((x) * sizeof(unsigned long))
1532 static unsigned char nargs[18]={AL(0),AL(3),AL(3),AL(3),AL(2),AL(3),
1533                                AL(3),AL(3),AL(4),AL(4),AL(4),AL(6),

```

Figure 185: Socket Glue - system call vector: `linux/net/socket.c`


```

1536
1537 /*
1538 *   System call vectors.
1539 *
1540 *   Argument checking cleaned up. Saved 20% in size.
1541 *   This function doesn't need to set the kernel lock because
1542 *   it is set by the callees.
1543 */
1544
1545 asmlinkage long sys_socketcall (int call, unsigned long *args)
1546 {
1547     unsigned long a [6];
1548     unsigned long a0,a1;
1549     int err;
1550
1551     if ( call < 1 || call > SYS_RECVMSG)
1552         return -EINVAL;
1553
1554     /* copy_from_user should be SMP safe. */
1555     if (copy_from_user(a, args, nargs[ call ]))
1556         return -EFAULT;
1557
1558     a0=a[0];
1559     a1=a[1];
1560
1561     switch( call )
1562     {
1563         case SYS_SOCKET:
1564             err = sys_socket (a0,a1,a [2] );
1565             break;
1566         case SYS_BIND:
1567             err = sys_bind(a0,(struct sockaddr *)a1, a [2] );
1568             break;
1569         case SYS_CONNECT:
1570             err = sys_connect(a0, (struct sockaddr *)a1, a [2] );
1571             break;
1572         case SYS_LISTEN:
1573             err = sys_listen (a0,a1);
1574             break;
1575         case SYS_ACCEPT:

```

Figure 186: Socket Glue - system call vector: `linux/net/socket.c`

```

1576         err = sys_accept(a0,(struct sockaddr *)a1, (int *)a[2]);
1577         break;
1578     case SYS_GETSOCKNAME:
1579         err = sys_getsockname(a0,(struct sockaddr *)a1, (int *)a[2]);
1580         break;
1581     case SYS_GETPEERNAME:
1582         err = sys_getpeername(a0, (struct sockaddr *)a1, (int *)a[2]);
1583         break;
1584     case SYS_SOCKETPAIR:
1585         err = sys_socketpair(a0,a1, a[2], (int *)a[3]);
1586         break;
1587     case SYS_SEND:
1588         err = sys_send(a0, (void *)a1, a[2], a[3]);
1589         break;
1590     case SYS_SENDTO:
1591         err = sys_sendto(a0,(void *)a1, a[2], a[3],
1592                        (struct sockaddr *)a[4], a[5]);
1593         break;
1594     case SYS_RECV:
1595         err = sys_recv(a0, (void *)a1, a[2], a[3]);
1596         break;

```

Figure 187: Socket Glue - system call vector: `linux/net/socket.c`

```

1598         err = sys_recvfrom(a0, (void *)a1, a [2], a [3],
1599                             (struct sockaddr *)a [4], (int *)a [5]);
1600         break;
1601     case SYS_SHUTDOWN:
1602         err = sys_shutdown(a0,a1);
1603         break;
1604     case SYS_SETSOCKOPT:
1605         err = sys_setsockopt(a0, a1, a [2], (char *)a [3], a [4]);
1606         break;
1607     case SYS_GETSOCKOPT:
1608         err = sys_getsockopt(a0, a1, a [2], (char *)a [3], (int *)a [4]);
1609         break;
1610     case SYS_SENDMSG:
1611         err = sys_sendmsg(a0, (struct msghdr *) a1, a [2]);
1612         break;
1613     case SYS_RECVMSG:
1614         err = sys_recvmsg(a0, (struct msghdr *) a1, a [2]);
1615         break;
1616     default :
1617         err = -EINVAL;
1618         break;
1619     }
1620     return err;

```

Figure 188: Socket Glue - registering a protocol family with the socket module:
[linux/net/socket.c](#)

```

1622
1623 /*
1624 *   This function is called by a protocol handler that wants to
1625 *   advertise its address family, and have it linked into the
1626 *   SOCKET module.
1627 */
1628
1629 int sock_register (struct net_proto_family *ops)
1630 {
1631     int err;
1632
1633     if (ops->family >= NPROTO) {
1634         printk(KERN_CRIT "protocol_%d_>=_NPROTO(%d)\n", ops->family,
1635                NPROTO);
1636         return -ENOBUFS;
1637     }
1638     net_family_write_lock ();

```

Figure 189: Socket Glue - un-registering a protocol family with the socket module: `linux/net/socket.c`

```

1646
1647 /*
1648 *   This function is called by a protocol handler that wants to
1649 *   remove its address family, and have it unlinked from the
1650 *   SOCKET module.
1651 */
1652
1653 int sock_unregister(int family)
1654 {
1655     if (family < 0 || family >= NPROTO)
1656         return -1;
1657
1658     net_family_write_lock ();
1659     net_families [family]=NULL;
1660     net_family_write_unlock ();
1661     return 0;
1662 }
1663
1664
1665 extern void sk_init (void);
1666
1667 #ifdef CONFIG_WAN_ROUTER
1668 extern void wanrouter_init(void);
1669 #endif
1670
1671 void _init sock_init (void)
1672 {
1673     int i;
1674
1675     printk(KERN_INFO "Linux.NET4.0_for_Linux.2.4\n");
1676     printk(KERN_INFO "Based_upon_Swansea_University_Computer_Society.NET3
1677             .039\n");
1678
1679     /*
1680      *   Initialize all address (protocol) families.

```

Figure 190: Socket Glue - initialising the socket VFS module:
linux/net/socket.c

```

1681
1682     for (i = 0; i < NPROTO; i++)
1683         net_families[i] = NULL;
1684
1685     /*
1686      *     Initialize sock SLAB cache.
1687      */
1688
1689     sk_init ();
1690
1691 #ifdef SLAB_SKB
1692     /*
1693      *     Initialize skbuff SLAB cache
1694      */
1695     skb_init ();
1696 #endif
1697
1698     /*
1699      *     Wan router layer.
1700      */
1701
1702 #ifdef CONFIG_WAN_ROUTER
1703     wanrouter_init();
1704 #endif

```

Figure 191: Socket Glue - initialising the socket VFS module:
linux/net/socket.c

```

1707     *       Initialize the protocols module.
1708     */
1709
1710     register_filesystem (&sock_fs_type);
1711     sock_mnt = kern_mount(&sock_fs_type);
1712     /* The real protocol initialization is performed when
1713     * do_initcalls is run.
1714     */
1715
1716
1717     /*
1718     * The netlink device handler may be needed early.
1719     */
1720
1721 #ifdef CONFIG_RTNETLINK
1722     rtnetlink_init ();
1723 #endif
1724 #ifdef CONFIG_NETLINK_DEV
1725     init_netlink ();
1726 #endif
1727 #ifdef CONFIG_NETFILTER
1728     netfilter_init ();
1729 #endif
1730 }
1731
1732 int socket_get_info (char *buffer, char **start, off_t offset, int length)

```

Figure 192: Socket Glue - socket stats per CPU: `linux/net/socket.c`



Linear Buffers - the use of `sk_buff` memory for packet processing

Figure 193: `sk_buff` structure layout


```
60 *
61 *   HW: device is required to csum packet as seen by hard_start_xmit
62 *   from skb->h.raw to the end and to record the checksum
63 *   at skb->h.raw+skb->csum.
64 *
65 *   Device must show its capabilities in dev->features, set
66 *   at device setup time.
67 *   NETIF_F_HW_CSUM - it is clever device, it is able to checksum
68 *   everything.
```

Figure 194: Memory Allocation:- Socket Buffers i:
linux/include/linux/skbuff.h

```

70 *   NETIF_F_IP_CSUM – device is dumb. It is able to csum only
71 *   TCP/UDP over IPv4. Sigh. Vendors like this
72 *   way by an unknown reason. Though, see comment above
73 *   about CHECKSUM_UNNECESSARY. 8)
74 *
75 *   Any questions? No questions, good.           --ANK
76 */
77
78 #ifdef _i386_
79 #define NET_CALLER(arg) (*((void*)&arg)-1)
80 #else
81 #define NET_CALLER(arg) __builtin_return_address(0)
82 #endif
83
84 #ifdef CONFIG_NETFILTER
85 struct nf_contrack {
86     atomic_t use;
87     void (*destroy)(struct nf_contrack *);
88 };
89
90 struct nf_ct_info {
91     struct nf_contrack *master;
92 };
93 #endif
94
95 struct sk_buff_head {
96     /* These two members must be first. */
97     struct sk_buff * next;
98     struct sk_buff * prev;

```

Figure 195: Memory Allocation:- Socket Buffers ii:
[linux/include/linux/skbuff.h](#)

```

160     union
161     {
162         struct ethhdr *ethernet;
163         unsigned char *raw;
164     } mac;
165
166     struct dst_entry *dst;
167
168     /*
169     * This is the control buffer. It is free to use for every
170     * layer. Please put your private variables there. If you
171     * want to keep them across layers you have to do a skb_clone()
172     * first. This is owned by whoever has the skb queued ATM.
173     */
174     char          cb[48];

```

Figure 196: Memory Allocation:- Socket Buffers Routines:
[linux/include/linux/skbuff.h](#)

```

165 struct sk_buff *alloc_skb(unsigned int size, int gfp_mask)
166 {
167     struct sk_buff *skb;
168     u8 *data;
169
170     if (in_interrupt() && (gfp_mask & _GFP_WAIT)) {
171         static int count = 0;
172         if (++count < 5) {
173             printk(KERN_ERR "alloc_skb_called_nonatomically_
174                "from_interrupt_%p\n", NET_CALLER(size));
175             BUG();
176         }
177         gfp_mask &= ~_GFP_WAIT;
178     }
179
180     /* Get the HEAD */
181     skb = skb_head_from_pool();
182     if (skb == NULL) {
183         skb = kmem_cache_alloc(skbuff_head_cache, gfp_mask);
184         if (skb == NULL)
185             goto nohead;
186     }
187
188     /* Get the DATA. Size must match skb_add_mtu(). */
189     size = SKB_DATA_ALIGN(size);
190     data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
191     if (data == NULL)
192         goto nodata;

```

Figure 197: Memory Allocation:- Socket Buffers Code

```

194     /* XXX: does not include slab overhead */
195     skb->truesize = size + sizeof(struct sk_buff);
196
197     /* Load the data pointers. */
198     skb->head = data;
199     skb->data = data;
200     skb->tail = data;
201     skb->end = data + size;
202
203     /* Set up other state */
204     skb->len = 0;
205     skb->cloned = 0;
206     skb->data_len = 0;
207
208     atomic_set(&skb->users, 1);
209     atomic_set(&(skb_shinfo(skb)->dateref), 1);
210     skb_shinfo(skb)->nr_frags = 0;
211     skb_shinfo(skb)->frag_list = NULL;
212     return skb;
213
214 nodata:
215     skb_head_to_pool(skb);

```

Figure 198: Memory Allocation:- Socket Buffers Code

```

1680     if (sk->forward_alloc >= TCP_MEM_QUANTUM)
1681         _tcp_mem_reclaim(sk);
1682     }
1683
1684 static inline void tcp_enter_memory_pressure(void)
1685 {
1686     if (!tcp_memory_pressure) {
1687         NET_INC_STATS(TCPMemoryPressures);
1688         tcp_memory_pressure = 1;
1689     }
1690 }
1691
1692 static inline void tcp_moderate_sndbuf(struct sock *sk)
1693 {
1694     if (!(sk->userlocks&SOCK_SNDBUF_LOCK)) {

```

Figure 199: TCP Memory Allocation:- Socket Buffers Code

```
65 int sysctl_hot_list_len = 128;
66
67 static kmem_cache_t *skbuff_head_cache;
68
69 static union {
70     struct sk_buff_head list;
71     char pad[SMP_CACHE_BYTES];
72 } skb_head_pool[NR_CPUS];
```

Figure 200: Socket Buffer Kernel Globals: `linux/net/core/skbuff.c`

```

113 static __inline__ struct sk_buff *skb_head_from_pool(void)
114 {
115     struct sk_buff_head *list = &skb_head_pool[smp_processor_id()].list;
116
117     if (skb_queue_len(list)) {
118         struct sk_buff *skb;
119         unsigned long flags;
120
121         local_irq_save(flags);
122         skb = __skb_dequeue(list);
123         local_irq_restore(flags);
124         return skb;
125     }
126     return NULL;
127 }
128
129 static __inline__ void skb_head_to_pool(struct sk_buff *skb)
130 {
131     struct sk_buff_head *list = &skb_head_pool[smp_processor_id()].list;
132
133     if (skb_queue_len(list) < sysctl_hot_list_len) {
134         unsigned long flags;
135
136         local_irq_save(flags);
137         __skb_queue_head(list, skb);
138         local_irq_restore(flags);
139
140         return;
141     }
142     kmem_cache_free(skbuff_head_cache, skb);
143 }

```

Figure 201: Socket Buffer Head from/to Pool: `linux/net/core/skbuff.c`


```
152 /**
153  *   alloc_skb      -   allocate a network buffer
154  *   @size: size to allocate
155  *   @gfp_mask: allocation mask
156  *
157  *   Allocate a new &sk_buff. The returned buffer has no headroom and a
158  *   tail room of size bytes. The object has a reference count of one.
159  *   The return is the buffer. On a failure the return is %NULL.
160  *
161  *   Buffers may only be allocated from interrupts using a @gfp_mask of
162  *   %GFP_ATOMIC.
163  */
```

Figure 202: Allocate Socket Buffer - comment: `linux/net/core/skbuff.c`

```

165 struct sk_buff *alloc_skb(unsigned int size, int gfp_mask)
166 {
167     struct sk_buff *skb;
168     u8 *data;
169
170     if (in_interrupt() && (gfp_mask & _GFP_WAIT)) {
171         static int count = 0;
172         if (++count < 5) {
173             printk(KERN_ERR "alloc_skb_called_nonatomically_
174                "from_interrupt_%p\n", NET_CALLER(size));
175             BUG();
176         }
177         gfp_mask &= ~_GFP_WAIT;
178     }
179
180     /* Get the HEAD */
181     skb = skb_head_from_pool();
182     if (skb == NULL) {
183         skb = kmem_cache_alloc(skbuff_head_cache, gfp_mask);
184         if (skb == NULL)
185             goto nohead;
186     }
187
188     /* Get the DATA. Size must match skb_add_mtu(). */
189     size = SKB_DATA_ALIGN(size);
190     data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
191     if (data == NULL)
192         goto no_data;
193
194     /* XXX: does not include slab overhead */
195     skb->truesize = size + sizeof(struct sk_buff);
196
197     /* Load the data pointers. */
198     skb->head = data;
199     skb->data = data;
200     skb->tail = data;
201     skb->end = data + size;
202
203     /* Set up other state */
204     skb->len = 0;
205     skb->cloned = 0;
206     skb->data_len = 0;
207
208     atomic_set(&skb->users, 1);
209     atomic_set(&(skb_shinfo(skb)->dateref), 1);
210     skb_shinfo(skb)->nr_frags = 0;
211     skb_shinfo(skb)->frag_list = NULL;
212     return skb;
213
214 no_data:
215     skb_head_to_pool(skb);

```

Figure 203: Allocate Socket Buffer: `linux/net/core/skbuff.c`

```

218 }
219
220
221 /*
222  *   Slab constructor for a skb head.
223  */
224 static inline void skb_headerinit(void *p, kmem_cache_t *cache,
225                                 unsigned long flags)
226 {
227     struct sk_buff *skb = p;
228
229     skb->next = NULL;
230     skb->prev = NULL;
231     skb->list = NULL;
232     skb->sk = NULL;
233     skb->stamp.tv_sec=0; /* No idea about time */
234     skb->dev = NULL;
235     skb->dst = NULL;
236     memset(skb->cb, 0, sizeof(skb->cb));
237     skb->pkt_type = PACKET_HOST; /* Default type */
238     skb->ip_summed = 0;
239     skb->priority = 0;
240     skb->security = 0; /* By default packets are insecure */
241     skb->destructor = NULL;
242
243 #ifdef CONFIG_NETFILTER
244     skb->nfmark = skb->nfcache = 0;
245     skb->nfct = NULL;
246 #ifdef CONFIG_NETFILTER_DEBUG
247     skb->nf_debug = 0;
248 #endif
249 #endif
250 #ifdef CONFIG_NET_SCHED

```

Figure 204: Initialise Socket Buffer: `linux/net/core/skbuff.c`

```
252 #endif
253 }
254
255 static void skb_drop_fraglist (struct sk_buff *skb)
256 {
257     struct sk_buff * list = skb_shinfo(skb)->frag_list;
258     skb_shinfo(skb)->frag_list = NULL;
259
260     do {
261
```

Figure 205: Free Socket Buffer: `linux/net/core/skbuff.c`

```

295 */
296 void kfree_skbmem(struct sk_buff *skb)
297 {
298     skb_release_data(skb);
299     skb_head_to_pool(skb);
300 }
301
302 /**
303  *   __kfree_skb - private function
304  *   @skb: buffer
305  *
306  *   Free an sk_buff. Release anything attached to the buffer.
307  *   Clean the state. This is an internal helper function. Users should
308  *   always call kfree_skb
309  */
310
311 void __kfree_skb(struct sk_buff *skb)
312 {
313     if (skb->list) {
314         printk(KERN_WARNING "Warning: __kfree_skb passed an skb still
315                "on a list (from %p).\n", NET_CALLER(skb));
316         BUG();
317     }
318
319     dst_release(skb->dst);
320     if (skb->destructor) {
321         if (in_irq()) {
322             printk(KERN_WARNING "Warning: __kfree_skb on hard IRQ %p\
323                    n",
324                    NET_CALLER(skb));
325         }
326         skb->destructor(skb);
327     }
328     #ifdef CONFIG_NETFILTER
329     nf_conntrack_put(skb->nfct);
330 #endif
331     skb_headerinit(skb, NULL, 0); /* clean state */
332     kfree_skbmem(skb);
333 }
334
335 /**
336  *   skb_clone - duplicate an sk_buff

```

Figure 206: Clone Socket Buffer: linux/net/core/skbuff.c

```

337 *      @gfp_mask: allocation priority
338 *
339 *      Duplicate an &skb_buff. The new one is not owned by a socket. Both
340 *      copies share the same packet data but not structure. The new
341 *      buffer has a reference count of 1. If the allocation fails the
342 *      function returns %NULL otherwise the new buffer is returned.
343 *
344 *      If this function is called from an interrupt gfp_mask() must be
345 *      %GFP_ATOMIC.
346 */
347
348 struct skb_buff *skb_clone(struct skb_buff *skb, int gfp_mask)
349 {
350     struct skb_buff *n;
351
352     n = skb_head_from_pool();
353     if (!n) {
354         n = kmem_cache_alloc(skbuff_head_cache, gfp_mask);
355         if (!n)
356             return NULL;
357     }
358
359 #define C(x) n->x = skb->x
360
361     n->next = n->prev = NULL;
362     n->list = NULL;
363     n->sk = NULL;
364     C(stamp);
365     C(dev);
366     C(h);
367     C(nh);
368     C(mac);
369     C(dst);
370     dst_clone(n->dst);
371     memcpy(n->cb, skb->cb, sizeof(skb->cb));
372     C(len);

```

Figure 207: Copy Socket Buffer Structure Header: `linux/net/core/skbuff.c`

```

374     C(csum);
375     n->cloned = 1;
376     C(pkt_type);
377     C(ip_summed);
378     C(priority);
379     atomic_set(&n->users, 1);
380     C(protocol);
381     C(security);
382     C(truesize);
383     C(head);
384     C(data);
385     C(tail);
386     C(end);
387     n->destructor = NULL;
388 #ifdef CONFIG_NETFILTER
389     C(nfmark);
390     C(nfcache);
391     C(nfct);
392 #ifdef CONFIG_NETFILTER_DEBUG
393     C(nf_debug);
394 #endif
395 #endif /*CONFIG_NETFILTER*/
396 #if defined(CONFIG_HIPPI)
397     C(private);
398 #endif
399 #ifdef CONFIG_NET_SCHED
400     C(tc_index);
401 #endif
402
403     atomic_inc(&(skb_shinfo(skb)->dataref));
404     skb->cloned = 1;
405 #ifdef CONFIG_NETFILTER
406     nf_conntrack_get(skb->nfct);
407 #endif
408     return n;
409 }

```

Figure 208: Copy Socket Buffer Structure *and* data:
linux/net/core/skbuff.c

```

412 {
413     /*
414      *      Shift between the two data areas in bytes
415      */
416     unsigned long offset = new->data - old->data;
417
418     new->list=NULL;
419     new->sk=NULL;
420     new->dev=old->dev;
421     new->priority=old->priority;
422     new->protocol=old->protocol;
423     new->dst=dst_clone(old->dst);
424     new->h.raw=old->h.raw+offset;
425     new->nh.raw=old->nh.raw+offset;
426     new->mac.raw=old->mac.raw+offset;
427     memcpy(new->cb, old->cb, sizeof(old->cb));
428     atomic_set(&new->users, 1);
429     new->pkt_type=old->pkt_type;
430     new->stamp=old->stamp;
431     new->destructor = NULL;
432     new->security=old->security;
433 #ifdef CONFIG_NETFILTER
434     new->nfmark=old->nfmark;
435     new->nfcache=old->nfcache;
436     new->nfct=old->nfct;
437     nf_contrack_get(new->nfct);
438 #ifdef CONFIG_NETFILTER_DEBUG
439     new->nf_debug=old->nf_debug;
440 #endif
441 #endif
442 #ifdef CONFIG_NET_SCHED
443     new->tc_index = old->tc_index;
444 #endif
445 }
446
447 /**
448  *      skb_copy      -      create private copy of an sk_buff
449  *      @skb: buffer to copy
450  *      @gfp_mask: allocation priority
451  *
452  *      Make a copy of both an &sk_buff and its data. This is used when the
453  *      caller wishes to modify the data and needs a private copy of the
454  *      data to alter. Returns %NULL on failure or the pointer to the buffer
455  *      on success. The returned buffer has a reference count of 1.
456  *
457  *      As by-product this function converts non-linear &sk_buff to linear
458  *      one, so that &sk_buff becomes completely private and caller is allowed

```

Figure 209: Copy and Expand Socket Buffer Structure *and* data:
linux/net/core/skbuff.c


```
473     if (n==NULL)
474         return NULL;
475
476     /* Set the data pointer */
477     skb_reserve(n,headerlen);
478     /* Set the tail pointer and length */
479     skb_put(n,skb->len);
480     n->csum = skb->csum;
481     n->ip_summed = skb->ip_summed;
482
483     if (skb_copy_bits(skb, -headerlen, n->head, headerlen+skb->len))
484         BUG();
485
486     copy_skb_header(n, skb);
```

Figure 210: Initialise Socket Buffer Global data: `linux/net/core/skbuff.c`

```

59  * all day. To optimize this case we use a specific flag bit defined
60  * below. As we add sockets to a bind bucket list , we perform a
61  * check of : ( newsk->reuse && (newsk->state != TCP_LISTEN))
62  * As long as all sockets added to a bind bucket pass this test ,
63  * the flag bit will be set .
64  * The resulting situation is that tcp_v[46] _verify_bind () can just check
65  * for this flag bit , if it is set and the socket trying to bind has
66  * sk->reuse set, we don't even have to walk the owners list at all,
67  * we return that it is ok to bind this socket to the requested local port.
68  *
69  * Sounds like a lot of work, but it is worth it . In a more naive
70  * implementation (ie. current FreeBSD etc.) the entire list of ports
71  * must be walked for each data port opened by an ftp server. Needless
72  * to say , this does not scale at all . With a couple thousand FTP
73  * users logged onto your box, isn't it nice to know that new data
74  * ports are created in O(1) time? I thought so. ;- )    -DaveM
75  */
76  struct tcp_bind_bucket {
77      unsigned short      port;
78      unsigned short      fastreuse;
79      struct tcp_bind_bucket *next;
80      struct sock         *owners;
81      struct tcp_bind_bucket **pprev;
82  };
83
84  struct tcp_bind_hashbucket {
85      spinlock_t          lock;
86      struct tcp_bind_bucket *chain;
87  };
88
89  extern struct tcp_hashinfo {
90      /* This is for sockets with full identity only. Sockets here will

```

Figure 211: TCP States: linux/include/net/tcp.h

```

92      *
93      *      TCP_ESTABLISHED <= sk->state < TCP_CLOSE
94      *
95      * First half of the table is for sockets not in TIME_WAIT, second half
96      * is for TIME_WAIT sockets only.
97      */
98      struct tcp_ehash_bucket *_tcp_ehash;
99
100     /* Ok, let's try this, I give up, we do need a local binding
101      * TCP hash as well as the others for fast bind/connect.
102      */
103     struct tcp_bind_hashbucket *_tcp_bhash;
104
105     int _tcp_bhash_size;
106     int _tcp_ehash_size;
107
108     /* All sockets in TCP_LISTEN state will be in here. This is the only
109      * table where wildcard'd TCP sockets can exist. Hash function here
110      * is just local port number.
111      */
112     struct sock *_tcp_listening_hash [TCP_LHTABLE_SIZE];
113
114     /* All the above members are written once at bootup and
115      * never written again _or_ are predominantly read-access.

```

Figure 212: TCP State flags: linux/include/net/tcp.h

```

117     * Now align to a new cache line as all the following members
118     * are often dirty.
119     */
120     rwlock_t _tcp_lhash_lock
121         __attribute__((__aligned__(SMP_CACHE_BYTES)));
122     atomic_t _tcp_lhash_users;
123     wait_queue_head_t _tcp_lhash_wait;
124     spinlock_t _tcp_portalloc_lock;
125 } tcp_hashinfo;
126
127 #define tcp_ehash      (tcp_hashinfo._tcp_ehash)
128 #define tcp_bhash      (tcp_hashinfo._tcp_bhash)
129 #define tcp_ehash_size (tcp_hashinfo._tcp_ehash_size)
130 #define tcp_bhash_size (tcp_hashinfo._tcp_bhash_size)
131 #define tcp_listening_hash (tcp_hashinfo._tcp_listening_hash)
132 #define tcp_lhash_lock (tcp_hashinfo._tcp_lhash_lock)
133 #define tcp_lhash_users (tcp_hashinfo._tcp_lhash_users)
134 #define tcp_lhash_wait (tcp_hashinfo._tcp_lhash_wait)
135 #define tcp_portalloc_lock (tcp_hashinfo._tcp_portalloc_lock)
136
137 extern kmem_cache_t *tcp_bucket_cachep;
138 extern struct tcp_bind_bucket *tcp_bucket_create(struct tcp_bind_hashbucket *head,
139                                                unsigned short snum);
140 extern void tcp_bucket_unlock(struct sock *sk);
141 extern int tcp_port_rovers;
142 extern struct sock *tcp_v4_lookup_listener(u32 addr, unsigned short hnum, int dif);
143
144 /* These are AF independent. */
145 static __inline__ int tcp_bhashfn(__u16 lport)
146 {
147     return (lport & (tcp_bhash_size - 1));

```

Figure 213: TCP Conenction State: linux/include/net/tcp.h

```

149
150 /* This is a TIME_WAIT bucket. It works around the memory consumption
151 * problems of sockets in such a state on heavily loaded servers, but
152 * without violating the protocol specification.
153 */
154 struct tcp_tw_bucket {
155     /* These _must_ match the beginning of struct sock precisely.
156     * XXX Yes I know this is gross, but I'd have to edit every single
157     * XXX networking file if I created a "struct sock_header". -DaveM
158     */
159     __u32          daddr;
160     __u32          rcv_saddr;
161     __u16          dport;
162     unsigned short num;
163     int           bound_dev_if;
164     struct sock   *next;
165     struct sock   **pprev;
166     struct sock   *bind_next;
167     struct sock   **bind_pprev;
168     unsigned char state,
169     substate; /* "zapped" is replaced with "substate" */
170     __u16          sport;
171     unsigned short family;
172     unsigned char reuse,
173     rcv_wscale; /* It is also TW bucket specific */
174     atomic_t      refcnt;
175
176     /* And these are ours. */
177     int           hashent;
178     int           timeout;
179     __u32          rcv_nxt;
180     __u32          snd_nxt;
181     __u32          rcv_wnd;
182     __u32          ts_recent;
183     long          ts_recent_stamp;
184     unsigned long  ttd;
185     struct tcp_bind_bucket *tb;

```

Figure 214: TCP State variables: linux/include/net/tcp.h

```

243 struct tcp_sack_block {
244     _u32  start_seq;
245     _u32  end_seq;
246 };
247
248 struct tcp_opt {
249     int    tcp_header_len; /* Bytes of tcp header to send    */
250
251     /*
252     *   Header prediction flags
253     *   0x5?10 << 16 + snd_wnd in net byte order
254     */
255     _u32  pred_flags;

```

Figure 215: Socket TCP State i, start of...: linux/include/net/sock.h

```

196 };
197 #endif
198
199 #if defined(CONFIG_INET) || defined (CONFIG_INET_MODULE)
200 struct inet_opt
201 {
202     int          ttl;           /* TTL setting */
203     int          tos;           /* TOS */
204     unsigned     cmsg_flags;
205     struct ip_options *opt;
206     unsigned char hdrincl;
207     __u8         mc_ttl;        /* Multicasting TTL */
208     __u8         mc_loop;      /* Loopback */
209     unsigned     recverr : 1,
210                freebind : 1;
211     __u16        id;           /* ID counter for DF pkts */

```

Figure 216: Socket relevant IP option State: `linux/include/net/sock.h`

```

140 };
141 #endif
142
143 #if defined(CONFIG_IPV6) || defined (CONFIG_IPV6_MODULE)
144 struct ipv6_pinfo {
145     struct in6_addr      saddr;
146     struct in6_addr      rcv_saddr;
147     struct in6_addr      daddr;
148     struct in6_addr      *daddr_cache;
149
150     __u32                flow_label ;
151     __u32                frag_size ;
152     int                  hop_limit;
153     int                  mcast_hops;
154     int                  mcast_oif;
155
156     /* pktoption flags */
157     union {
158         struct {
159             __u8      srcrt :2,
160                     rxinfo:1,
161                     rxhlim:1,
162                     hopopts:1,
163                     dstopts:1,
164                     authhdr:1,
165                     rxflow:1;
166         } bits;
167         __u8      all ;
168     } rxopt;
169
170     /* sockopt flags */
171     __u8      mc_loop:1,
172             rcvrr:1,
173             sndflow:1,
174             pmtudisc:2;
175
176     struct ipv6_mc_socklist *ipv6_mc_list ;
177     struct ipv6_fl_socklist * ipv6_fl_list ;
178     __u32                dst_cookie;

```

Figure 217: Socket IPv6 State: linux/include/net/sock.h


```

482 do {    spin_lock_init (&((__sk)->lock.slock)); \
483         (__sk)->lock.users = 0; \
484         init_waitqueue_head(&((__sk)->lock.wq)); \
485     } while(0);
486
487 struct sock {
488     /* Socket demultiplex comparisons on incoming packets. */
489     __u32          daddr;      /* Foreign IPv4 addr
490     */
491     __u32          rcv_saddr;  /* Bound local IPv4 addr
492     */
493     __u16          dport;      /* Destination port
494     */
495     unsigned short num;        /* Local port
496     */
497     int            bound_dev_if; /* Bound device index if != 0
498     */

```

Figure 218: Socket State i, demultiplex data: `linux/include/net/sock.h`

```

496     struct sock          *next;
497     struct sock          **pprev;
498     struct sock          *bind_next;
499     struct sock          **bind_pprev;
500
501     volatile unsigned char state,          /* Connection state
502         */
503         zapped;          /* In ax25 & ipx means not linked
504         */
505     _u16                  sport;          /* Source port
506         */
507     unsigned short        family;          /* Address family
508         */
509     unsigned char         reuse;          /* SO_REUSEADDR setting
510         */
511     unsigned char         shutdown;
512     atomic_t              refcnt;          /* Reference count
513         */
514     socket_lock_t         lock;          /* Synchronizer...
515         */
516     int                    rcvbuf;          /* Size of receive buffer in bytes
517         */
518     wait_queue_head_t     *sleep;          /* Sock wait queue
519         */
520     struct dst_entry       *dst_cache;      /* Destination cache
521         */
522     rwlock_t              dst_lock;
523     atomic_t              rmem_alloc;      /* Receive queue bytes committed
524         */
525     struct sk_buff_head    receive_queue;  /* Incoming packets
526         */
527     atomic_t              wmem_alloc;      /* Transmit queue bytes committed
528         */
529     struct sk_buff_head    write_queue;    /* Packet sending queue
530         */
531     atomic_t              omem_alloc;      /* "o" is "option" or "other" */
532     int                    wmem_queued;    /* Persistent queue size */

```

Figure 219: Socket State ii, resource data: linux/include/net/sock.h

```

523     __u32          saddr;      /* Sending source
524         */
525     unsigned int  allocation;  /* Allocation mode
526         */
527     int           sndbuf;      /* Size of send buffer in bytes
528         */
529     struct sock   *prev;
530
531     /* Not all are volatile, but some are, so we might as well say they all
532        are.
533        * XXX Make this a flag word -DaveM
534        */
535     volatile char  dead,
536                 done,
537                 urginline,
538                 keepopen,
539                 linger,
540                 destroy,
541                 no_check,
542                 broadcast,
543                 bsdism;
544     unsigned char  debug;
545     unsigned char  rcvtstamp;
546     unsigned char  use_write_queue;
547     unsigned char  userlocks;
548     /* Hole of 3 bytes. Try to pack. */
549     int            route_caps;
550     int            proc;
551     unsigned long  lingertime;
552
553     int            hashent;
554     struct sock   *pair;
555
556     /* The backlog queue is special, it is always used with
557        * the per-socket spinlock held and requires low latency
558        * access. Therefore we special case it's implementation.
559        */
560     struct {

```

Figure 220: Socket State iii, socket state itself: linux/include/net/sock.h

```

558         struct sk_buff *tail;
559     } backlog;
560
561     rwlock_t          callback_lock;
562
563     /* Error queue, rarely used. */
564     struct sk_buff_head error_queue;
565
566     struct proto      *prot;
567
568     #if defined(CONFIG_IPV6) || defined (CONFIG_IPV6_MODULE)
569         union {
570             struct ipv6_pinfo      af_inet6;
571         } net_pinfo;
572     #endif
573
574         union {
575             struct tcp_opt          af_tcp;
576     #if defined(CONFIG_INET) || defined (CONFIG_INET_MODULE)
577             struct raw_opt          tp_raw4;
578     #endif
579     #if defined(CONFIG_IPV6) || defined (CONFIG_IPV6_MODULE)
580             struct raw6_opt         tp_raw;
581     #endif /* CONFIG_IPV6 */
582     #if defined(CONFIG_SPX) || defined (CONFIG_SPX_MODULE)
583             struct spx_opt          af_spx;
584     #endif /* CONFIG_SPX */
585
586         } tp_pinfo;
587
588     int          err, err_soft; /* Soft holds errors that don't
589                                cause failure but are the cause
590                                of a persistent failure not just
591                                'timed out' */
592
593     unsigned short  ack_backlog;
594     unsigned short  max_ack_backlog;
595     _u32            priority;
596     unsigned short  type;
597     unsigned char   localroute; /* Route locally only */
598     unsigned char   protocol;
599     struct ucred    peercred;

```

Figure 221: Socket State iv, routing, erroring and filtering: linux/include/net/sock.h

```

599     int                rcvlowat;
600     long               rcvtimeo;
601     long              sndtimeo;
602
603 #ifndef CONFIG_FILTER
604     /* Socket Filtering Instructions */
605     struct sk_filter    * filter ;
606 #endif /* CONFIG_FILTER */
607
608     /* This is where all the private (optional) areas that don't
609     * overlap will eventually live.
610     */
611     union {
612         void *destruct_hook;
613         struct unix_opt af_unix;
614 #if defined(CONFIG_INET) || defined(CONFIG_INET_MODULE)
615         struct inet_opt af_inet;
616 #endif
617 #if defined(CONFIG_ATALK) || defined(CONFIG_ATALK_MODULE)
618         struct atalk_sock af_at;
619 #endif
620 #if defined(CONFIG_IPX) || defined(CONFIG_IPX_MODULE)
621         struct ipx_opt af_ipx;
622 #endif
623 #if defined(CONFIG_DECNET) || defined(CONFIG_DECNET_MODULE)
624         struct dn_scp dn;
625 #endif
626 #if defined(CONFIG_PACKET) || defined(CONFIG_PACKET_MODULE)
627         struct packet_opt *af_packet;
628 #endif
629 #if defined(CONFIG_X25) || defined(CONFIG_X25_MODULE)
630         x25_cb *x25;
631 #endif
632 #if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
633         ax25_cb *ax25;
634 #endif
635 #if defined(CONFIG_NETROM) || defined(CONFIG_NETROM_MODULE)
636         nr_cb *nr;
637 #endif
638 #if defined(CONFIG_ROSE) || defined(CONFIG_ROSE_MODULE)
639         rose_cb *rose;
640 #endif
641 #if defined(CONFIG_PPPOE) || defined(CONFIG_PPPOE_MODULE)
642         struct pppox_opt *pppox;
643 #endif
644 #ifdef CONFIG_NETLINK
645         struct netlink_opt *af_netlink;
646 #endif
647 #if defined(CONFIG_ECONET) || defined(CONFIG_ECONET_MODULE)
648         struct econet_opt *af_econet;

```

Figure 222: Socket State v, private protocol info: `linux/include/net/sock.h`

```

610     */
611     union {
612         void *destruct_hook;
613         struct unix_opt af_unix;
614 #if defined(CONFIG_INET) || defined (CONFIG_INET_MODULE)
615         struct inet_opt af_inet;
616 #endif
617 #if defined(CONFIG_ATALK) || defined(CONFIG_ATALK_MODULE)
618         struct atalk_sock af_at;
619 #endif
620 #if defined(CONFIG_IPX) || defined(CONFIG_IPX_MODULE)
621         struct ipx_opt af_ipx;
622 #endif
623 #if defined (CONFIG_DECNET) || defined(CONFIG_DECNET_MODULE)
624         struct dn_sep dn;
625 #endif
626 #if defined (CONFIG_PACKET) || defined(CONFIG_PACKET_MODULE)
627         struct packet_opt *af_packet;
628 #endif
629 #if defined(CONFIG_X25) || defined(CONFIG_X25_MODULE)
630         x25_cb *x25;
631 #endif
632 #if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
633         ax25_cb *ax25;
634 #endif
635 #if defined(CONFIG_NETROM) || defined(CONFIG_NETROM_MODULE)
636         nr_cb *nr;
637 #endif
638 #if defined(CONFIG_ROSE) || defined(CONFIG_ROSE_MODULE)
639         rose_cb *rose;
640 #endif
641 #if defined(CONFIG_PPPOE) || defined(CONFIG_PPPOE_MODULE)
642         struct pppox_opt *pppox;
643 #endif
644 #ifdef CONFIG_NETLINK
645         struct netlink_opt *af_netlink;
646 #endif
647 #if defined(CONFIG_ECONET) || defined(CONFIG_ECONET_MODULE)
648         struct econet_opt *af_econet;

```

Figure 223: Socket State vi, timers and callbacks: `linux/include/net/sock.h`

```

649 #endif
650 #if defined(CONFIG_ATM) || defined(CONFIG_ATM_MODULE)
651     struct atm_vcc     *af_atm;
652 #endif
653 #if defined(CONFIG_IRDA) || defined(CONFIG_IRDA_MODULE)
654     struct irda_sock   *irda;
655 #endif
656 #if defined(CONFIG_WAN_ROUTER) || defined(CONFIG_WAN_ROUTER_MODULE)
657     struct wanpipe_opt *af_wanpipe;
658 #endif
659     } protinfo;
660
661
662     /* This part is used for the timeout functions. */
663     struct timer_list timer;          /* This is the sock cleanup timer. */
664     struct timeval stamp;
665
666     /* Identd and reporting IO signals */
667     struct socket *socket;
668
669     /* RPC layer private data */
670     void *user_data;

```

Figure 224: Socket State vi, timers and callbacks: `linux/include/net/sock.h`

```

418         int                linger2;
419     };
420
421
422     /*
423     * This structure really needs to be cleaned up.
424     * Most of it is for TCP, and not used by any of
425     * the other protocols.
426     */
427
428     /*
429     * The idea is to start moving to a newer struct gradually
430     *
431     * IMHO the newer struct should have the following format:
432     *
433     *     struct sock {
434     *         sockmem [mem, proto, callbacks]
435     *
436     *         union or struct {
437     *             ax25;
438     *         } ll_pinfo;
439     *
440     *         union {
441     *             ipv4;
442     *             ipv6;
443     *             ipx;
444     *             netrom;
445     *             rose;
446     *             x25;
447     *         } net_pinfo;
448     *
449     *         union {
450     *             tcp;
451     *             udp;
452     *             spx;
453     *             netrom;
454     *         } tp_pinfo;
455     *

```

Figure 225: Proposal to Tidy Up Socket State: linux/include/net/sock.h


```

947     if (atomic_dec_and_test(&fp->refcnt))
948         kfree(fp);
949 }
950
951 static inline void sk_filter_charge (struct sock *sk, struct sk_filter *fp)
952 {
953     atomic_inc(&fp->refcnt);
954     atomic_add(sk_filter_len(fp), &sk->omem_alloc);
955 }
956
957 #endif /* CONFIG_FILTER */
958
959 /*
960  * Socket reference counting postulates.
961  *
962  * * Each user of socket SHOULD hold a reference count.
963  * * Each access point to socket (an hash table bucket, reference from a list,
964  *   running timer, skb in flight MUST hold a reference count.
965  * * When reference count hits 0, it means it will never increase back.
966  * * When reference count hits 0, it means that no references from
967  *   outside exist to this socket and current process on current CPU
968  *   is last user and may/should destroy this socket.
969  * * sk_free is called from any context: process, BH, IRQ. When
970  *   it is called, socket has no references from outside -> sk_free
971  *   may release descendant resources allocated by the socket, but
972  *   to the time when it is called, socket is NOT referenced by any
973  *   hash tables, lists etc.
974  * * Packets, delivered from outside (from network or from another process)
975  *   and enqueued on receive/error queues SHOULD NOT grab reference count,
976  *   when they sit in queue. Otherwise, packets will leak to hole, when

```

Figure 226: Socket Reference Counting Comment: linux/include/net/sock.h

```

978 * It is true for udp/raw, netlink (leak to receive and error queues), tcp
979 * (leak to backlog). Packet socket does all the processing inside
980 * BR_NETPROTO_LOCK, so that it has not this race condition. UNIX sockets
981 * use separate SMP lock, so that they are prone too.
982 */
983
984 /* Grab socket reference count. This operation is valid only
985    when sk is ALREADY grabbed f.e. it is found in hash table
986    or a list and the lookup is made under lock preventing hash table
987    modifications.
988 */
989
990 static inline void sock_hold(struct sock *sk)
991 {
992     atomic_inc(&sk->refcnt);
993 }
994
995 /* Ungrab socket in the context, which assumes that socket refcnt
996    cannot hit zero, f.e. it is true in context of any socketcall.

```

Figure 227: Inline functions for Socket Reference Management:
linux/include/net/sock.h

```

998 static inline void __sock_put(struct sock *sk)
999 {
1000     atomic_dec(&sk->refcnt);
1001 }
1002
1003 /* Ungrab socket and destroy it, if it was the last reference. */
1004 static inline void sock_put(struct sock *sk)
1005 {
1006     if (atomic_dec_and_test(&sk->refcnt))
1007         sk_free(sk);
1008 }
1009
1010 /* Detach socket from process context.
1011  * Announce socket dead, detach it from wait queue and inode.
1012  * Note that parent inode held reference count on this struct sock,

```

Figure 228: Inline functions for Socket Reference Management:
linux/include/net/sock.h

```

1014  * probably wants some additional cleanups or even continuing
1015  * to work with this socket (TCP).
1016  */
1017  static inline void sock_orphan(struct sock *sk)
1018  {
1019      write_lock_bh(&sk->callback_lock);
1020      sk->dead = 1;
1021      sk->socket = NULL;
1022      sk->sleep = NULL;
1023      write_unlock_bh(&sk->callback_lock);
1024  }
1025
1026  static inline void sock_graft(struct sock *sk, struct socket *parent)
1027  {
1028      write_lock_bh(&sk->callback_lock);
1029      sk->sleep = &parent->wait;
1030      parent->sk = sk;
1031      sk->socket = parent;
1032      write_unlock_bh(&sk->callback_lock);
1033  }
1034
1035  static inline int sock_iuid(struct sock *sk)
1036  {
1037      int uid;
1038
1039      read_lock(&sk->callback_lock);
1040      uid = sk->socket ? sk->socket->inode->iuid : 0;
1041      read_unlock(&sk->callback_lock);

```

Figure 229: Inline functions for Socket Reference Management:
linux/include/net/sock.h

```

1043 }
1044
1045 static inline unsigned long sock_i_ino(struct sock *sk)
1046 {
1047     unsigned long ino;
1048
1049     read_lock(&sk->callback_lock);
1050     ino = sk->socket ? sk->socket->inode->i_ino : 0;
1051     read_unlock(&sk->callback_lock);
1052     return ino;
1053 }
1054
1055 static inline struct dst_entry *
1056 __sk_dst_get(struct sock *sk)
1057 {
1058     return sk->dst_cache;
1059 }
1060
1061 static inline struct dst_entry *
1062 sk_dst_get(struct sock *sk)
1063 {
1064     struct dst_entry *dst;
1065
1066     read_lock(&sk->dst_lock);
1067     dst = sk->dst_cache;
1068     if (dst)
1069         dst_hold(dst);
1070     read_unlock(&sk->dst_lock);
1071     return dst;
1072 }
1073
1074 static inline void
1075 __sk_dst_set(struct sock *sk, struct dst_entry *dst)
1076 {
1077     struct dst_entry *old_dst;

```

Figure 230: Socket Route State Management: `linux/include/net/sock.h`

```
1080     sk->dst_cache = dst;
1081     dst_release(old_dst);
1082 }
1083
1084 static inline void
1085 sk_dst_set(struct sock *sk, struct dst_entry *dst)
1086 {
1087     write_lock(&sk->dst_lock);
1088     _sk_dst_set(sk, dst);
1089     write_unlock(&sk->dst_lock);
1090 }
1091
1092 static inline void
1093 _sk_dst_reset(struct sock *sk)
1094 {
1095     struct dst_entry *old_dst;
```

Figure 231: Socket Route State Management: `linux/include/net/sock.h`

```

1098     sk->dst_cache = NULL;
1099     dst_release(old_dst);
1100 }
1101
1102 static inline void
1103 sk_dst_reset(struct sock *sk)
1104 {
1105     write_lock(&sk->dst_lock);
1106     __sk_dst_reset(sk);
1107     write_unlock(&sk->dst_lock);
1108 }
1109
1110 static inline struct dst_entry *
1111 __sk_dst_check(struct sock *sk, u32 cookie)
1112 {
1113     struct dst_entry *dst = sk->dst_cache;
1114
1115     if (dst && dst->obsolete && dst->ops->check(dst, cookie) == NULL) {
1116         sk->dst_cache = NULL;
1117         return NULL;
1118     }
1119
1120     return dst;
1121 }

```

Figure 232: Socket Route State Management: `linux/include/net/sock.h`

```

1125 {
1126     struct dst_entry *dst = sk_dst_get(sk);
1127
1128     if (dst && dst->obsolete && dst->ops->check(dst, cookie) == NULL) {
1129         sk_dst_reset(sk);
1130         return NULL;
1131     }
1132
1133     return dst;
1134 }
1135
1136
1137 /*
1138 * Queue a received datagram if it will fit. Stream and sequenced
1139 * protocols can't normally use this as they need to fit buffers in
1140 * and play with them.

```

Figure 233: Socket Queue Management Interface: `linux/include/net/sock.h`


```

1141  *
1142  *   Inlined as it's very short and called for pretty much every
1143  *   packet ever received.
1144  */
1145
1146  static inline void skb_set_owner_w(struct sk_buff *skb, struct sock *sk)
1147  {
1148      sock_hold(sk);
1149      skb->sk = sk;
1150      skb->destructor = sock_wfree;
1151      atomic_add(skb->truesize, &sk->wmem_alloc);
1152  }
1153
1154  static inline void skb_set_owner_r(struct sk_buff *skb, struct sock *sk)
1155  {
1156      skb->sk = sk;
1157      skb->destructor = sock_rfree;
1158      atomic_add(skb->truesize, &sk->rmem_alloc);
1159  }
1160
1161  static inline int sock_queue_rcv_skb(struct sock *sk, struct sk_buff *skb)
1162  {
1163      /* Cast skb->rcvbuf to unsigned... It's pointless, but reduces
1164       * number of warnings when compiling with -W --ANK
1165       */
1166      if (atomic_read(&sk->rmem_alloc) + skb->truesize >= (unsigned)sk->rcvbuf)
1167          return -ENOMEM;
1168
1169  #ifdef CONFIG_FILTER
1170      if (sk->filter) {
1171          int err = 0;
1172          struct sk_filter *filter;
1173
1174          /* It would be deadlock, if sock_queue_rcv_skb is used
1175           * with socket lock! We assume that users of this
1176           * function are lock free.
1177           */
1178          bh_lock_sock(sk);
1179          if ((filter = sk->filter) != NULL && sk_filter(skb, filter))
1180              err = -EPERM;
1181          bh_unlock_sock(sk);

```

Figure 234: Socket Queue Management Interface: `linux/include/net/sock.h`

```

1286 }
1287
1288 /*
1289  *   Enable debug/info messages
1290  */
1291
1292 #if 0
1293 #define NETDEBUG(x) do { } while (0)
1294 #else
1295 #define NETDEBUG(x) do { x; } while (0)
1296 #endif
1297
1298 /*
1299  * Macros for sleeping on a socket. Use them like this:
1300  *
1301  * SOCK_SLEEP_PRE(sk)
1302  * if (condition)
1303  *     schedule();
1304  * SOCK_SLEEP_POST(sk)
1305  *
1306  */
1307
1308 #define SOCK_SLEEP_PRE(sk) { struct task_struct *tsk = current; \

```

Figure 235: Socket Queue Management Interface: `linux/include/net/sock.h`

```

44 static __inline__ int scm_check_creds(struct ucred *creds)
45 {
46     if ((creds->pid == current->pid || capable(CAP_SYS_ADMIN)) &&
47         ((creds->uid == current->uid || creds->uid == current->euid ||
48          creds->uid == current->suid) || capable(CAP_SETUID)) &&
49         ((creds->gid == current->gid || creds->gid == current->egid ||
50          creds->gid == current->sgid) || capable(CAP_SETGID))) {
51         return 0;
52     }
53     return -EPERM;
54 }

```

Figure 236: Socket Credential Function: `linux/net/core/scm.c`

```
9 struct scm_fp_list
10 {
11     int          count;
12     struct file  *fp[SCM_MAX_FD];
13 };
14
15 struct scm_cookie
16 {
17     struct ucred  creds;          /* Skb credentials */
18     struct scm_fp_list *fp;      /* Passed files */
19     unsigned long seq;          /* Connection seqno */
20 };
```

Figure 237: Socket Credential Interface: `linux/include/net/scm.h`

```

1
2
3 -- 1/2 of Transport Protocol
4 --      Consumer of User events, Producer of Network requests...
5     for(ever)
6     {
7         sleep(tsdu-q-event);
8
9         prev = spl-ts()
10        tsdu-pkt = deq(tsdu-q);
11        spl-restore(prev)
12
13        -- make up some tpdu (transport pkts)
14        -- do all the rtx q stuff and set timers
15        -- if this is that service/kind of transport protocol
16        -- (and network service isn't enough to give us the
17        -- service we are asked for)
18
19        prev = spl-ns()
20        enqueue(nsdu-q, tpdu-pkt)
21        spl-restore(prev)
22
23        wakeup(nsdu-q-event);
24
25    }

```

Figure 238: BSD producer consumer chain - transport out

```

1
2 -- 1/2 of Network Protocol - NS1
3 --      Consumer of Transport events, Producer of Link requests
4   for(ever)
5   {
6       sleep(nsdu-q-event);
7
8       prev = spl-ns()
9       tsdu-pkt = deq(nsdu-q);
10      spl-previous(prev)
11
12      -- make up some npdu (network pkts)
13      -- do all the rtx q stuff and set timers
14      -- if this is that kind of Network protocol
15
16      prev = spl-ls()
17      enqueue(lsdu-q, npdu-pkt)
18      spl-restore(prev)
19
20  }

```

Figure 239: BSD producer consumer chain - network out

1
2 sleep - waits (suspends this task/process/thread) till some other task
3 wakes us up with appropriate event).
4
5 wakeup - wakeup any task sleeping for this event.
6
7 spl-? - set priority level to exclude any tasks from waking up who
8 need this level of access.
9
10 spl-x - restore all previous priority levels.
11
12 que - add a buffer/packet to a queue for another process to...
13
14 dequeue - remove from a queue some buffer.

Figure 240: BSD producer consumer chain - legend

```

118  /*
119  *          Add an ip header to a skbuff and send it out.
120  */
121  int ip_build_and_send_pkt(struct sk_buff *skb, struct sock *sk,
122                          u32 saddr, u32 daddr, struct ip_options *opt)
123  {
124      struct rtable *rt = (struct rtable *)skb->dst;
125      struct iphdr *iph;
126
127      /* Build the IP header. */
128      if (opt)
129          iph=(struct iphdr *)skb_push(skb,sizeof(struct iphdr) + opt->optlen);
130      else
131          iph=(struct iphdr *)skb_push(skb,sizeof(struct iphdr));
132
133      iph->version = 4;
134      iph->ihl     = 5;
135      iph->tos     = sk->protinfo.af_inet.tos;
136      iph->frag_off = 0;
137      if (ip_dont_fragment(sk, &rt->u.dst))
138          iph->frag_off |= htons(IP_DF);
139      iph->ttl     = sk->protinfo.af_inet.ttl;
140      iph->daddr  = rt->rt_dst;
141      iph->saddr  = rt->rt_src;
142      iph->protocol = sk->protocol;
143      iph->tot_len = htons(skb->len);
144      ip_select_ident (iph, &rt->u.dst, sk);
145      skb->nh.iph = iph;
146
147      if (opt && opt->optlen) {
148          iph->ihl += opt->optlen>>2;
149          ip_options_build(skb, opt, daddr, rt, 0);
150      }
151      ip_send_check(iph);
152
153      /* Send it out. */
154      return NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev,
155                   output_maybe_reroute);
156  }

```

Figure 241: IP Output - add header and send: linux/net/ipv4/ip_output.c


```

158 static inline int ip_finish_output2(struct sk_buff *skb)
159 {
160     struct dst_entry *dst = skb->dst;
161     struct hh_cache *hh = dst->hh;
162
163     #ifdef CONFIG_NETFILTER_DEBUG
164         nf_debug_ip_finish_output2(skb);
165     #endif /*CONFIG_NETFILTER_DEBUG*/
166
167     if (hh) {
168         read_lock_bh(&hh->hh_lock);
169         memcpy(skb->data - 16, hh->hh_data, 16);
170         read_unlock_bh(&hh->hh_lock);
171         skb_push(skb, hh->hh_len);
172         return hh->hh_output(skb);
173     } else if (dst->neighbour)
174         return dst->neighbour->output(skb);
175
176     printk(KERN_DEBUG "krm\n");
177     kfree_skb(skb);
178     return -EINVAL;
179 }
180
181 __inline__ int ip_finish_output(struct sk_buff *skb)
182 {
183     struct net_device *dev = skb->dst->dev;
184
185     skb->dev = dev;
186     skb->protocol = __constant_htons(ETH_P_IP);
187
188     return NF_HOOK(PF_INET, NF_IP_POST_ROUTING, skb, NULL, dev,
189                 ip_finish_output2);
190 }

```

Figure 242: IP Output - finish sending - note two stage process:
linux/net/ipv4/ip_output.c

```
87  /* Generate a checksum for an outgoing IP datagram. */
88  _inline_ void ip_send_check(struct iphdr *iph)
89  {
90      iph->check = 0;
91      iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);
92  }
```

Figure 243: Internet Protocol - output - checksum:
linux/net/ipv4/ip_output.c

```

94  /* dev_loopback_xmit for use with netfilter. */
95  static int ip_dev_loopback_xmit(struct sk_buff *newskb)
96  {
97      newskb->mac.raw = newskb->data;
98      __skb_pull(newskb, newskb->nh.raw - newskb->data);
99      newskb->pkt_type = PACKET_LOOPBACK;
100     newskb->ip_summed = CHECKSUM_UNNECESSARY;
101     BUG_TRAP(newskb->dst);
102
103     #ifdef CONFIG_NETFILTER_DEBUG
104         nf_debug_ip_loopback_xmit(newskb);
105     #endif
106     netif_rx(newskb);
107     return 0;
108 }

```

Figure 244: IP Output - loopback: `linux/net/ipv4/ip_output.c`

```
110 /* Don't just hand NF_HOOK skb->dst->output, in case netfilter hook
111     changes route */
112 static inline int
113 output_maybe_reroute(struct sk_buff *skb)
114 {
115     return skb->dst->output(skb);
116 }
```

Figure 245: IP Output - reroute?: `linux/net/ipv4/ip_output.c`

```

192 int ip_mc_output(struct sk_buff *skb)
193 {
194     struct sock *sk = skb->sk;
195     struct rtable *rt = (struct rtable*)skb->dst;
196     struct net_device *dev = rt->u.dst.dev;
197
198     /*
199      *   If the indicated interface is up and running, send the packet.
200      */
201     IP_INC_STATS(IpOutRequests);
202 #ifdef CONFIG_IP_ROUTE_NAT
203     if (rt->rt_flags & RTCF_NAT)
204         ip_do_nat(skb);
205 #endif
206
207     skb->dev = dev;
208     skb->protocol = _constant_htons(ETH_P_IP);
209
210     /*
211      *   Multicasts are looped back for other local users
212      */
213
214     if (rt->rt_flags & RTCF_MULTICAST && (!sk || sk->prot_info.af_inet.mc_loop)) {
215 #ifdef CONFIG_IP_MROUTE
216         /* Small optimization: do not loopback not local frames,
217          * which returned after forwarding; they will be dropped
218          * by ip_mr_input in any case.
219          * Note, that local frames are looped back to be delivered
220          * to local recipients.
221          *
222          * This check is duplicated in ip_mr_input at the moment.
223          */
224         if ((rt->rt_flags & RTCF_LOCAL) || !(IPCB(skb)->flags &
225             IPSKB_FORWARDED))
226 #endif
227         {
228             struct sk_buff *newskb = skb_clone(skb, GFP_ATOMIC);
229             if (newskb)
230                 NF_HOOK(PF_INET, NF_IP_POST_ROUTING, newskb,
231                     NULL,
232                     newskb->dev,
233                     ip_dev_loopback_xmit);
234         }
235     }
236 }

```

Figure 246: IP Output - multicast replication...: linux/net/ipv4/ip_output.c

```

234      /* Multicasts with ttl 0 must not go beyond the host */
235
236      if (skb->nh.iph->ttl == 0) {
237          kfree_skb(skb);
238          return 0;
239      }
240
241
242      if (rt->rt_flags&RTCF_BROADCAST) {
243          struct sk_buff *newskb = skb_clone(skb, GFP_ATOMIC);
244          if (newskb)
245              NF_HOOK(PF_INET, NF_IP_POST_ROUTING, newskb, NULL,
246                     newskb->dev, ip_dev_loopback_xmit);
247      }
248
249      return ip_finish_output(skb);
250 }

```

Figure 247: IP Output - multicast replication...: `linux/net/ipv4/ip_output.c`

```
252 int ip_output(struct sk_buff *skb)
253 {
254 #ifdef CONFIG_IP_ROUTE_NAT
255     struct rtable *rt = (struct rtable*)skb->dst;
256 #endif
257
258     IP_INC_STATS(IpOutRequests);
259
260 #ifdef CONFIG_IP_ROUTE_NAT
261     if (rt->rt_flags&RTCF_NAT)
262         ip_do_nat(skb);
263 #endif
264
265     return ip_finish_output(skb);
266 }
```

Figure 248: IP Output - drive output: `linux/net/ipv4/ip_output.c`

```

268 /* Queues a packet to be sent, and starts the transmitter if necessary.
269 * This routine also needs to put in the total length and compute the
270 * checksum. We use to do this in two stages, ip_build_header() then
271 * this, but that scheme created a mess when routes disappeared etc.
272 * So we do it all here, and the TCP send engine has been changed to
273 * match. (No more unroutable FIN disasters, etc. wheee...) This will
274 * most likely make other reliable transport layers above IP easier
275 * to implement under Linux.
276 */
277 static inline int ip_queue_xmit2(struct sk_buff *skb)
278 {
279     struct sock *sk = skb->sk;
280     struct rtable *rt = (struct rtable *)skb->dst;
281     struct net_device *dev;
282     struct iphdr *iph = skb->nh.iph;
283
284     dev = rt->u.dst.dev;
285
286     /* This can happen when the transport layer has segments queued
287     * with a cached route, and by the time we get here things are
288     * re-routed to a device with a different MTU than the original
289     * device. Sick, but we must cover it.
290     */
291     if (skb_headroom(skb) < dev->hard_header_len && dev->hard_header) {
292         struct sk_buff *skb2;
293
294         skb2 = skb_realloc_headroom(skb, (dev->hard_header_len + 15) & ~15);
295         kfree_skb(skb);
296         if (skb2 == NULL)
297             return -ENOMEM;
298         if (sk)
299             skb_set_owner_w(skb2, sk);
300         skb = skb2;
301         iph = skb->nh.iph;
302     }
303
304     if (skb->len > rt->u.dst.pmtu)
305         goto fragment;

```

Figure 249: IP Output - Queue for sending - part 2i:
linux/net/ipv4/ip_output.c


```

307     if (ip_dont_fragment(sk, &rt->u.dst))
308         iph->frag_off |= __constant_htons(IP_DF);
309
310     ip_select_ident (iph, &rt->u.dst, sk);
311
312     /* Add an IP checksum. */
313     ip_send_check(iph);
314
315     skb->priority = sk->priority;
316     return skb->dst->output(skb);
317
318 fragment:
319     if (ip_dont_fragment(sk, &rt->u.dst)) {
320         /* Reject packet ONLY if TCP might fragment
321          * it itself, if were careful enough.
322          */
323         iph->frag_off |= __constant_htons(IP_DF);
324         NETDEBUG(printk(KERN_DEBUG "sending_pkt_too_big_to_self\n"));
325
326         icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED,
327                 htonl(rt->u.dst.pmtu));
328         kfree_skb(skb);
329         return -EMSGSIZE;
330     }
331     ip_select_ident (iph, &rt->u.dst, sk);
332     if (skb->ip_summed == CHECKSUM_HW &&
333         (skb = skb_checksum_help(skb)) == NULL)

```

Figure 250: IP Output - Queue for sending - part 2ii:
linux/net/ipv4/ip_output.c

```

335     return ip_fragment(skb, skb->dst->output);
336 }
337
338 int ip_queue_xmit(struct sk_buff *skb)
339 {
340     struct sock *sk = skb->sk;
341     struct ip_options *opt = sk->protinfo.af_inet.opt;
342     struct rtable *rt;
343     struct iphdr *iph;
344
345     /* Make sure we can route this packet. */
346     rt = (struct rtable *) __sk_dst_check(sk, 0);
347     if (rt == NULL) {
348         u32 daddr;
349
350         /* Use correct destination address if we have options. */
351         daddr = sk->daddr;
352         if (opt && opt->srr)
353             daddr = opt->faddr;
354
355         /* If this fails, retransmit mechanism of transport layer will
356          * keep trying until route appears or the connection times itself
357          * out.
358          */
359         if (ip_route_output(&rt, daddr, sk->saddr,
360                             RT_TOS(sk->protinfo.af_inet.tos) | RTO_CONN | sk
361                                 ->localroute,
362                             sk->bound_dev_if))
363             goto no_route;
364         __sk_dst_set(sk, &rt->u.dst);
365         sk->route_caps = rt->u.dst.dev->features;
366     }

```

Figure 251: IP Output - Queue packet for sending - part 1:
linux/net/ipv4/ip_output.c

```

367         if (opt && opt->is_strictroute && rt->rt_dst != rt->rt_gateway)
368             goto no_route;
369
370
371         /* OK, we know where to send it, allocate and build IP header. */
372         iph = (struct iphdr *) skb_push(skb, sizeof(struct iphdr) + (opt ? opt->optlen
373             : 0));
374         *((_u16 *)iph) = htons((4 << 12) | (5 << 8) | (sk->protinfo.af_inet.tos & 0xff));
375         iph->tot_len = htons(skb->len);
376         iph->frag_off = 0;
377         iph->ttl = sk->protinfo.af_inet.ttl;
378         iph->protocol = sk->protocol;
379         iph->saddr = rt->rt_src;
380         iph->daddr = rt->rt_dst;
381         skb->nh.iph = iph;
382         /* Transport layer set skb->h.foo itself. */
383
384         if (opt && opt->optlen) {
385             iph->ihl += opt->optlen >> 2;
386             ip_options_build(skb, opt, sk->daddr, rt, 0);
387         }
388
389         return NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev,
390             ip_queue_xmit2);
391 no_route:

```

Figure 252: IP Output - Queue packet for sending - part 1:
linux/net/ipv4/ip_output.c

```

393     kfree_skb(skb);
394     return -EHOSTUNREACH;
395 }
396
397 /*
398  *   Build and send a packet, with as little as one copy
399  *
400  *   Doesn't care much about ip options... option length can be
401  *   different for fragment at 0 and other fragments.
402  *
403  *   Note that the fragment at the highest offset is sent first,
404  *   so the getfrag routine can fill in the TCP/UDP checksum header
405  *   field in the last fragment it sends... actually it also helps
406  *   the reassemblers, they can put most packets in at the head of
407  *   the fragment queue, and they know the total size in advance. This
408  *   last feature will measurably improve the Linux fragment handler one
409  *   day.
410  *
411  *   The callback has five args, an arbitrary pointer (copy of frag),
412  *   the source IP address (may depend on the routing table), the
413  *   destination address (char *), the offset to copy from, and the
414  *   length to be copied.
415  */
416
417 static int ip_build_xmit_slow(struct sock *sk,
418                             int getfrag (const void *,
419                                         char *,
420                                         unsigned int,
421                                         unsigned int),
422                             const void *frag,
423                             unsigned length,

```

Figure 253: IP Output - classic building of IP packet:
linux/net/ipv4/ip_output.c

```

424         struct ipcm_cookie *ipc,
425         struct rtable *rt,
426         int flags)
427     {
428         unsigned int fraglen, maxfraglen, fragheaderlen;
429         int err;
430         int offset, mf;
431         int mtu;
432         u16 id;
433
434         int hh_len = (rt->u.dst.dev->hard_header_len + 15)&~15;
435         int nfrags=0;
436         struct ip_options *opt = ipc->opt;
437         int df = 0;
438
439         mtu = rt->u.dst.pmtu;
440         if (ip_dont_fragment(sk, &rt->u.dst))
441             df = htons(IP_DF);
442
443         length -= sizeof(struct iphdr);
444
445         if (opt) {
446             fragheaderlen = sizeof(struct iphdr) + opt->optlen;
447             maxfraglen = ((mtu - sizeof(struct iphdr) - opt->optlen) & ~7) +
448                 fragheaderlen;
449         } else {
450             fragheaderlen = sizeof(struct iphdr);
451
452             /*
453              *   Fragheaderlen is the size of 'overhead' on each buffer. Now
454              *   work
455              *   out the size of the frames to send.

```

Figure 254: IP Output - classic building of IP packet:
linux/net/ipv4/ip_output.c

```

455         maxfraglen = ((mtu-sizeof(struct iphdr)) & ~7) + fragheaderlen;
456     }
457
458     if (length + fragheaderlen > 0xFFFF) {
459         ip_local_error(sk, EMSGSIZE, rt->rt_dst, sk->dport, mtu);
460         return -EMSGSIZE;
461     }
462
463     /*
464     *      Start at the end of the frame by handling the remainder.
465     */
466
467     offset = length - (length % (maxfraglen - fragheaderlen));
468
469     /*
470     *      Amount of memory to allocate for final fragment.
471     */
472
473     fraglen = length - offset + fragheaderlen;
474
475     if (length-offset==0) {
476         fraglen = maxfraglen;
477         offset -= maxfraglen-fragheaderlen;
478     }
479
480     /*
481

```

Figure 255: IP Output - classic building of IP packet:
linux/net/ipv4/ip_output.c

```
483     */
484
485     mf = 0;
486
487     /*
488     *     Don't fragment packets for path mtu discovery.
489     */
490
491     if ( offset > 0 && sk->protinfo.af_inet.pmtudisc==IP_PMTUDISC_DO) {
492         ip_local_error (sk, EMSGSIZE, rt->rt_dst, sk->dport, mtu);
```

Figure 256: IP Output - classic building of IP packet:
linux/net/ipv4/ip_output.c

```

494     }
495     if ( flags & MSG_PROBE)
496         goto out;
497
498     /*
499     *   Begin outputting the bytes.
500     */
501
502     id = (sk ? sk->protinfo.af_inet.id++ : 0);
503
504     do {
505         char *data;
506         struct sk_buff * skb;
507
508         /*
509         *   Get the memory we require with some space left for
510         *   alignment.
511         */
512         skb = sock_alloc_send_skb(sk, fraglen+hh_len+15, flags&
513             MSG_DONTWAIT, &err);
514         if (skb == NULL)
515             goto error;
516
517         /*
518         *   Fill in the control structures
519         */
520         skb->priority = sk->priority;
521         skb->dst = dst_clone(&rt->u.dst);
522         skb_reserve(skb, hh_len);

```

Figure 257: IP Output - classic building of IP packet:
linux/net/ipv4/ip_output.c


```

525     *      Find where to start putting bytes.
526     */
527
528     data = skb_put(skb, fraglen);
529     skb->nh.iph = (struct iphdr *)data;
530
531     /*
532     *      Only write IP header onto non-raw packets
533     */
534
535     {
536         struct iphdr *iph = (struct iphdr *)data;
537
538         iph->version = 4;
539         iph->ihl = 5;
540         if (opt) {
541             iph->ihl += opt->optlen>>2;
542             ip_options_build(skb, opt,

```

Figure 258: IP Output - classic building of IP packet:
[linux/net/ipv4/ip_output.c](#)

```

543         ipc->addr, rt, offset);
544     }
545     iph->tos = sk->protoinfo.af_inet.tos;
546     iph->tot_len = htons(fraglen - fragheaderlen + iph->ihl*4);
547     iph->frag_off = htons(offset >> 3) | mf | df;
548     iph->id = id;
549     if (!mf) {
550         if (offset || !df) {
551             /* Select an unpredictable ident only
552              * for packets without DF or having
553              * been fragmented.
554              */
555             _ip_select_ident (iph, &rt->u.dst);
556             id = iph->id;
557         }
558
559         /*
560          * Any further fragments will have MF set.
561          */
562         mf = htons(IP_MF);
563     }
564     if (rt->rt_type == RTN_MULTICAST)
565         iph->ttl = sk->protoinfo.af_inet.mc_ttl;
566     else
567         iph->ttl = sk->protoinfo.af_inet.ttl;
568     iph->protocol = sk->protocol;

```

Figure 259: IP Output - classic building of IP packet:
linux/net/ipv4/ip_output.c

```

569         iph->check = 0;
570         iph->saddr = rt->rt_src;
571         iph->daddr = rt->rt_dst;
572         iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);
573         data += iph->ihl*4;
574     }
575
576     /*
577     *   User data callback
578     */
579
580     if (getfrag(frag, data, offset, fraglen - fragheaderlen)) {
581         err = -EFAULT;
582         kfree_skb(skb);
583         goto error;

```

Figure 260: IP Output - classic building of IP packet:
linux/net/ipv4/ip_output.c

```

584         }
585
586         offset -= (maxfraglen-fragheaderlen);
587         fraglen = maxfraglen;
588
589         nfrags++;
590
591         err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL,
592                    skb->dst->dev, output_maybe_reroute);
593         if (err) {
594             if (err > 0)
595                 err = skb->protinfo.af_inet.recvrr ? net_xmit_errno(err)
596                    : 0;
597             if (err)
598                 goto error;
599         }
600     } while (offset >= 0);
601
602     if (nfrags>1)
603         ip_statistics [smp_processor_id()*2 + !in_softirq()].IpFragCreates +=
604             nfrags;
605 out:
606     return 0;

```

Figure 261: IP Output - classic building of IP packet:
linux/net/ipv4/ip_output.c

```

607     IP_INC_STATS(IpOutDiscards);
608     if (nfrags>1)
609         ip_statistics [smp_processor_id()*2 + !in_softirq()].IpFragCreates +=
            nfrags;
610     return err;
611 }
612
613 /*
614  *   Fast path for unfragmented packets.
615  */
616 int ip_build_xmit(struct sock *sk,
617                 int getfrag (const void *,
618                             char *,
619                             unsigned int,
620                             unsigned int),
621                 const void *frag,
622                 unsigned length,
623                 struct ipcm_cookie *ipc,
624                 struct rtable *rt,
625                 int flags)
626 {
627     int err;
628     struct sk_buff *skb;
629     int df;
630     struct iphdr *iph;
631
632     /*
633      *   Try the simple case first. This leaves fragmented frames, and by
634      *   choice RAW frames within 20 bytes of maximum size(rare) to the
635      *   long path
636      */
637     if (!sk->protinfo.af_inet.hdrincl) {
638         length += sizeof(struct iphdr);
639
640         /*
641          *   Check for slow path.
642          */
643         if (length > rt->u.dst.pmtu || ipc->opt != NULL)
644             return ip_build_xmit_slow(sk, getfrag, frag, length, ipc, rt, flags);
645     } else {
646         if (length > rt->u.dst.dev->mtu) {

```

Figure 262: IP Output - fast path for non fragmented packets:
linux/net/ipv4/ip_output.c

```

648         return -EMSGSIZE;
649     }
650 }
651 if ( flags &MSG_PROBE)
652     goto out;
653
654 /*
655  *   Do path mtu discovery if needed.
656  */
657 df = 0;
658 if (ip_dont_fragment(sk, &rt->u.dst))
659     df = htons(IP_DF);
660
661 /*
662  *   Fast path for unfragmented frames without options.
663  */
664 {
665     int hh_len = (rt->u.dst.dev->hard_header_len + 15)&~15;
666
667     skb = sock_alloc_send_skb(sk, length+hh_len+15,
668                               flags &MSG_DONTWAIT, &err);
669     if (skb==NULL)
670         goto error;
671     skb_reserve(skb, hh_len);
672 }
673
674 skb->priority = sk->priority;
675 skb->dst = dst_clone(&rt->u.dst);
676
677 skb->nh.iph = iph = (struct iphdr *)skb_put(skb, length);
678
679 if (!sk->proto.af_inet_hdrincl) {
680     iph->version=4;
681     iph->ihl=5;
682     iph->tos=sk->proto.af_inet.tos;
683     iph->tot_len = htons(length);
684     iph->frag_off = df;
685     iph->ttl=sk->proto.af_inet.mc_ttl;
686     ip_select_ident (iph, &rt->u.dst, sk);
687     if (rt->rt_type != RTN_MULTICAST)
688         iph->ttl=sk->proto.af_inet.ttl;
689     iph->protocol=sk->protocol;
690     iph->saddr=rt->rt_src;
691     iph->daddr=rt->rt_dst;

```

Figure 263: IP Output - fast path for non fragmented packets:
linux/net/ipv4/ip_output.c

```

693         iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);
694         err = getfrag(frag, ((char *)iph)+iph->ihl*4, 0, length-iph->ihl*4);
695     }
696     else
697         err = getfrag(frag, (void *)iph, 0, length);
698
699     if (err)
700         goto error_fault ;
701
702     err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev,
703                 output_maybe_reroute);
704     if (err > 0)
705         err = sk->protinfo.af_inet.recvrr ? net_xmit_errno(err) : 0;
706     if (err)
707         goto error;
708 out:
709     return 0;
710
711 error_fault :

```

Figure 264: IP Output - fast path for non fragmented packets:
linux/net/ipv4/ip_output.c

```

713     kfree_skb(skb);
714 error:
715     IP_INC_STATS(IpOutDiscards);
716     return err;
717 }
718
719 /*
720 *   This IP datagram is too large to be sent in one piece. Break it up into
721 *   smaller pieces (each of size equal to IP header plus
722 *   a block of the data of the original IP data part) that will yet fit in a
723 *   single device frame, and queue such a frame for sending.
724 *
725 *   Yes this is inefficient, feel free to submit a quicker one.
726 */
727
728 int ip_fragment(struct sk_buff *skb, int (*output)(struct sk_buff*))
729 {
730     struct iphdr *iph;
731     int raw = 0;
732     int ptr;
733     struct net_device *dev;
734     struct sk_buff *skb2;
735     unsigned int mtu, hlen, left, len;
736     int offset;
737     int not_last_frag;
738     struct rtable *rt = (struct rtable*)skb->dst;
739     int err = 0;
740
741     dev = rt->u.dst.dev;

```

Figure 265: IP Output - fragment: linux/net/ipv4/ip_output.c


```

744     *      Point into the IP datagram header.
745     */
746
747     iph = skb->nh.iph;
748
749     /*
750     *      Setup starting values.
751     */
752
753     hlen = iph->ihl * 4;
754     left = skb->len - hlen;      /* Space per frame */
755     mtu = rt->u.dst.pmtu - hlen; /* Size of data space */
756     ptr = raw + hlen;          /* Where to start from */
757
758     /*
759     *      Fragment the datagram.
760     */
761
762     offset = (ntohs(iph->frag_off) & IP_OFFSET) << 3;
763     not_last_frag = iph->frag_off & htons(IP_MF);
764
765     /*
766     *      Keep copying data until we run out.
767     */
768
769     while(left > 0) {
770         len = left;
771         /* IF: it doesn't fit, use 'mtu' - the data space left */
772         if (len > mtu)
773             len = mtu;
774         /* IF: we are not sending upto and including the packet end
775         then align the next start on an eight byte boundary */
776         if (len < left) {

```

Figure 266: IP Output - fragment: linux/net/ipv4/ip_output.c

```

778     }
779     /*
780     *   Allocate buffer.
781     */
782
783     if ((skb2 = alloc_skb(len+hlen+dev->hard_header_len+15,GFP_ATOMIC
784     )) == NULL) {
785         NETDEBUG(printk(KERN_INFO "IP: frag: no memory for new
786         fragment!\n"));
787         err = -ENOMEM;
788         goto fail ;
789     }
790
791     /*
792     *   Set up data on packet
793     */
794
795     skb2->pkt_type = skb->pkt_type;
796     skb2->priority = skb->priority;
797     skb_reserve(skb2, (dev->hard_header_len+15)&~15);
798     skb_put(skb2, len + hlen);
799     skb2->nh.raw = skb2->data;
800     skb2->h.raw = skb2->data + hlen;
801
802     /*
803     *   Charge the memory for the fragment to any owner
804     *   it might possess
805     */
806
807     if (skb->sk)
808         skb_set_owner_w(skb2, skb->sk);
809     skb2->dst = dst_clone(skb->dst);
810     skb2->dev = skb->dev;
811
812     /*
813     *   Copy the packet header into the new buffer.
814     */
815
816     memcpy(skb2->nh.raw, skb->data, hlen);

```

Figure 267: IP Output - fragment: linux/net/ipv4/ip_output.c

```

817     *      Copy a block of the IP datagram.
818     */
819     if (skb_copy_bits(skb, ptr, skb2->h.raw, len))
820         BUG();
821     left -= len;
822
823     /*
824     *      Fill in the new header fields.
825     */
826     iph = skb2->nh.iph;
827     iph->frag_off = htons((offset >> 3));
828
829     /* ANK: dirty, but effective trick. Upgrade options only if
830     * the segment to be fragmented was THE FIRST (otherwise,
831     * options are already fixed) and make it ONCE
832     * on the initial skb, so that all the following fragments
833     * will inherit fixed options.
834     */
835     if (offset == 0)
836         ip_options_fragment(skb);
837
838     /* Copy the flags to each fragment. */
839     IPCB(skb2)->flags = IPCB(skb)->flags;
840
841     /*
842     *      Added AC : If we are fragmenting a fragment that's not the
843     *                  last fragment then keep MF on each bit
844     */
845     if (left > 0 || not_last_frag)
846         iph->frag_off |= htons(IP_MF);
847     ptr += len;
848     offset += len;

```

Figure 268: IP Output - fragment: linux/net/ipv4/ip_output.c

```

850 #ifdef CONFIG_NETFILTER
851     /* Connection association is same as pre-frag packet */
852     skb2->nfct = skb->nfct;
853     nf_conntrack_get(skb2->nfct);
854 #ifdef CONFIG_NETFILTER_DEBUG
855     skb2->nf_debug = skb->nf_debug;
856 #endif
857 #endif
858
859     /*
860     *     Put this fragment into the sending queue.
861     */
862
863     IP_INC_STATS(IpFragCreates);
864
865     iph->tot_len = htons(len + hlen);
866
867     ip_send_check(iph);
868
869     err = output(skb2);
870     if (err)
871         goto fail;
872 }

```

Figure 269: IP Output - fragment: `linux/net/ipv4/ip_output.c`

```

874     IP_INC_STATS(IpFragOKs);
875     return err;
876
877 fail:
878     kfree_skb(skb);
879     IP_INC_STATS(IpFragFails);
880     return err;
881 }
882
883 /*
884  *   Fetch data from kernel space and fill in checksum if needed.
885  */
886 static int ip_reply_glue_bits(const void *dptr, char *to, unsigned int offset,
887                             unsigned int fraglen)
888 {
889     struct ip_reply_arg *dp = (struct ip_reply_arg *)dptr;
890     u16 *pktp = (u16 *)to;
891     struct iovec *iov;
892     int len;
893     int hdrflag = 1;
894
895     iov = &dp->iov[0];
896     if (offset >= iov->iov_len) {
897         offset -= iov->iov_len;
898         iov++;
899         hdrflag = 0;
900     }
901     len = iov->iov_len - offset;
902     if (fraglen > len) { /* overlapping. */
903         dp->csum = csum_partial_copy_nocheck(iov->iov_base+offset, to, len,
904                                             dp->csum);
905         offset = 0;
906         fraglen -= len;
907         to += len;
908         iov++;

```

Figure 270: IP Output - deal with iovec and checksum for next function:
linux/net/ipv4/ip_output.c

```

910     dp->csum = csum_partial_copy_nocheck(iov->iov_base+offset, to, fraglen,
911     dp->csum);
912
913
914     if (hdrflag && dp->csumoffset)
915         *(pktp + dp->csumoffset) = csum_fold(dp->csum); /* fill in
          checksum */
916     return 0;
917 }
918
919 /*
920  * Generic function to send a packet as reply to another packet.
921  * Used to send TCP resets so far. ICMP should use this function too.
922  *
923  * Should run single threaded per socket because it uses the sock
924  * structure to pass arguments.
925  */
926 void ip_send_reply(struct sock *sk, struct sk_buff *skb, struct ip_reply_arg *arg,
927 unsigned int len)
928 {
929     struct {
930         struct ip_options    opt;
931         char                 data[40];
932     } replyopts;
933     struct ipcm_cookie ipc;
934     u32 daddr;
935     struct rtable *rt = (struct rtable*)skb->dst;
936
937     if (ip_options_echo(&replyopts.opt, skb))
938         return;
939
940     daddr = ipc.addr = rt->rt_src;
941     ipc.opt = NULL;
942
943     if (replyopts.opt.optlen) {
944         ipc.opt = &replyopts.opt;
945
946         if (ipc.opt->srr)
947             daddr = replyopts.opt.faddr;
948     }
949
950     if (ip_route_output(&rt, daddr, rt->rt_spec_dst, RT_TOS(skb->nh.iph->tos), 0)
951         )
952         return;
953
954     /* And let IP do all the hard work.
955
956        This chunk is not reenterable, hence spinlock.
957        Note that it uses the fact, that this function is called
958        with locally disabled BH and that sk cannot be already spinlocked.
959    */

```

Figure 271: IP Output - build a packet that is a response to something:
linux/net/ipv4/ip_output.c

```

960     sk->protinfo.af_inet.tos = sk->nh.iph->tos;
961     sk->priority = sk->priority;
962     sk->protocol = sk->nh.iph->protocol;
963     ip_build_xmit(sk, ip_reply_glue_bits, arg, len, &ipc, rt, MSG_DONTWAIT);
964     bh_unlock_sock(sk);
965
966     ip_rt_put(rt);
967 }
968
969 /*
970  *   IP protocol layer initialiser
971  */
972
973 static struct packet_type ip_packet_type =
974 {
975     _constant_htons(ETH_P_IP),
976     NULL, /* All devices */
977     ip_rcv,
978     (void*)1,
979     NULL,
980 };
981
982 /*
983  *   IP registers the packet type and then calls the subprotocol initialisers
984  */
985
986 void _init ip_init(void)
987 {

```

Figure 272: IP Output - initialise IP layer: `linux/net/ipv4/ip_output.c`

```
103 * To Fix:
104 *      IP fragmentation wants rewriting cleanly . The RFC815 algorithm is much
105 *      more efficient
106 *      and could be made very efficient with the addition of some virtual
107 *      memory hacks to permit
108 *      the allocation of a buffer that can then be 'grown' by twiddling page
109 *      tables.
110 *      Output fragmentation wants updating along with the buffer management to
111 *      use a single
112 *      interleaved copy algorithm so that fragmenting has a one copy overhead.
113 *      Actual packet
114 *      output should probably do its own fragmentation at the UDP/RAW layer.
115 *      TCP shouldn't cause
116 *      fragmentation anyway.
```

Figure 273: Internet Protocol - Input- preamble: `linux/net/ipv4/ip_input.c`


```
151 struct ip_mib ip_statistics [NR_CPUS*2];
```

Figure 274: IP Input - stats for network managers:
linux/net/ipv4/ip_input.c

```

153  /*
154  *      Process Router Attention IP option
155  */
156  int ip_call_ra_chain (struct sk_buff *skb)
157  {
158      struct ip_ra_chain *ra;
159      u8 protocol = skb->nh.iph->protocol;
160      struct sock *last = NULL;
161
162      read_lock(&ip_ra_lock);
163      for (ra = ip_ra_chain; ra; ra = ra->next) {
164          struct sock *sk = ra->sk;
165
166          /* If socket is bound to an interface, only report
167           * the packet if it came from that interface.
168           */
169          if (sk && sk->num == protocol
170              && ((sk->bound_dev_if == 0)
171                  || (sk->bound_dev_if == skb->dev->ifindex))) {
172              if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
173                  skb = ip_defrag(skb);
174                  if (skb == NULL) {
175                      read_unlock(&ip_ra_lock);
176                      return 1;
177                  }
178              }
179              if (last) {
180                  struct sk_buff *skb2 = skb_clone(skb, GFP_ATOMIC);
181                  if (skb2)
182                      raw_rcv(last, skb2);
183              }
184              last = sk;
185          }
186      }
187
188      if (last) {
189          raw_rcv(last, skb);
190          read_unlock(&ip_ra_lock);
191          return 1;
192      }
193      read_unlock(&ip_ra_lock);
194      return 0;
195  }

```

Figure 275: IP Input - Router Alter Handler: `linux/net/ipv4/ip_input.c`

```

197 /* Handle this out of line, it is rare. */
198 static int ip_run_ipprot(struct sk_buff *skb, struct iphdr *iph,
199                        struct inet_protocol *ipprot, int force_copy)
200 {
201     int ret = 0;
202
203     do {
204         if (ipprot->protocol == iph->protocol) {
205             struct sk_buff *skb2 = skb;
206             if (ipprot->copy || force_copy)
207                 skb2 = skb_clone(skb, GFP_ATOMIC);
208             if (skb2 != NULL) {
209                 ret = 1;
210                 ipprot->handler(skb2);
211             }
212             ipprot = (struct inet_protocol *) ipprot->next;
213         } while(ipprot != NULL);
214     }
215
216     return ret;
217 }

```

Figure 276: IP Input - find ip proto - not inline...: linux/net/ipv4/ip_input.c

```

220 {
221     int ihl = skb->nh.iph->ihl*4;
222
223 #ifdef CONFIG_NETFILTER_DEBUG
224     nf_debug_ip_local_deliver(skb);
225 #endif /*CONFIG_NETFILTER_DEBUG*/
226
227     /* Pull out additional 8 bytes to save some space in protocols. */
228     if (!pskb_may_pull(skb, ihl+8))
229         goto out;
230     __skb_pull(skb, ihl);
231
232 #ifdef CONFIG_NETFILTER
233     /* Free reference early: we don't need it any more, and it may
234        hold ip_conntrack module loaded indefinitely. */
235     nf_conntrack_put(skb->nfct);
236     skb->nfct = NULL;
237 #endif /*CONFIG_NETFILTER*/
238
239     /* Point into the IP datagram, just past the header. */
240     skb->h.raw = skb->data;
241
242     {
243         /* Note: See raw.c and net/raw.h, RAWV4_HTABLE_SIZE==
244            MAX_INET_PROTOS */
245         int protocol = skb->nh.iph->protocol;
246         int hash = protocol & (MAX_INET_PROTOS - 1);
247         struct sock *raw_sk = raw_v4_htable[hash];
248         struct inet_protocol *ipprot;
249         int flag;
250
251         /* If there maybe a raw socket we must check - if not we
252            * don't care less
253            */
254         if (raw_sk != NULL)
255             raw_sk = raw_v4_input(skb, skb->nh.iph, hash);
256
257         ipprot = (struct inet_protocol *) inet_protos[hash];
258         flag = 0;
259         if (ipprot != NULL) {
260             if (raw_sk == NULL &&
261                 ipprot->next == NULL &&
262                 ipprot->protocol == protocol) {
263                 int ret;
264
265                 /* Fast path... */
266                 ret = ipprot->handler(skb);

```

Figure 277: IP Input - deliver IP packets 2nd stage:
linux/net/ipv4/ip_input.c

```

279         raw_rcv(raw_sk, skb);
280         sock_put(raw_sk);
281     } else if (!flag) {          /* Free and report errors */
282         icmp_send(skb, ICMP_DEST_UNREACH,
283                    ICMP_PROT_UNREACH, 0);
284 out:
285         kfree_skb(skb);
286     }
287
288     return 0;
289 }
290
291 /*
292  *   Deliver IP Packets to the higher protocol layers.
293  */
294 int ip_local_deliver (struct sk_buff *skb)
295 {
296     /*
297      *   Reassemble IP fragments.
298      */

```

Figure 278: IP Input - deliver IP packets 1st stage:
linux/net/ipv4/ip_input.c

```

300     if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
301         skb = ip_defrag(skb);
302         if (!skb)
303             return 0;
304     }
305
306     return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,
307                   ip_local_deliver_finish );
308 }
309
310 static inline int ip_rcv_finish (struct sk_buff *skb)
311 {
312     struct net_device *dev = skb->dev;
313     struct iphdr *iph = skb->nh.iph;
314
315     /*
316      *   Initialise the virtual path cache for the packet. It describes
317      *   how the packet travels inside Linux networking.
318      */
319     if (skb->dst == NULL) {
320         if (ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev))
321             goto drop;
322     }

```

Figure 279: IP Input - finish IP reception: `linux/net/ipv4/ip_input.c`

```

325     if (skb->dst->tclassid) {
326         struct ip_rt_acct *st = ip_rt_acct + 256*smp_processor_id();
327         u32 idx = skb->dst->tclassid;
328         st[idx&0xFF].o_packets++;
329         st[idx&0xFF].o_bytes+=skb->len;
330         st[(idx>>16)&0xFF].i_packets++;
331         st[(idx>>16)&0xFF].i_bytes+=skb->len;
332     }
333 #endif
334
335     if (iph->ihl > 5) {
336         struct ip_options *opt;
337
338         /* It looks as overkill, because not all
339          * IP options require packet mangling.
340          * But it is the easiest for now, especially taking
341          * into account that combination of IP options
342          * and running sniffer is extremely rare condition.
343          * --ANK (980813)
344          */
345
346         if (skb_cow(skb, skb_headroom(skb)))
347             goto drop;
348         iph = skb->nh.iph;
349
350         skb->ip_summed = 0;
351         if (ip_options_compile(NULL, skb))
352             goto inhdr_error;
353
354         opt = &(IPCB(skb)->opt);
355         if (opt->srr) {
356             struct in_device *in_dev = in_dev_get(dev);
357             if (in_dev) {
358                 if (!IN_DEV_SOURCE_ROUTE(in_dev)) {
359                     if (IN_DEV_LOG_MARTIANS(in_dev) &&
360                         net_ratelimit())
361                         printk(KERN_INFO "source_route_option
362                                %u.%u.%u.%u->%u.%u.%u.%u\n",
363                                NIPQUAD(iph->saddr),
364                                NIPQUAD(iph->daddr));
365                     in_dev_put(in_dev);
366                     goto drop;
367                 }
368                 in_dev_put(in_dev);
369             }
370             if (ip_options_rcv_srr(skb))
371                 goto drop;
372         }
373     }

```

Figure 280: IP Input - finish IP reception: linux/net/ipv4/ip_input.c

```

372     return skb->dst->input(skb);
373
374 inhdr_error:
375     IP_INC_STATS_BH(IpInHdrErrors);
376 drop:
377     kfree_skb(skb);
378     return NET_RX_DROP;
379 }
380
381 /*
382  *   Main IP Receive routine.
383  */
384 int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)
385 {
386     struct iphdr *iph = skb->nh.iph;
387
388     /* When the interface is in promisc. mode, drop all the crap
389      * that it receives, do not try to analyse it.
390      */
391     if (skb->pkt_type == PACKET_OTHERHOST)
392         goto drop;
393
394     IP_INC_STATS_BH(IpInReceives);
395
396     if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL)
397         goto out;
398
399     if (!pskb_may_pull(skb, sizeof(struct iphdr)))

```

Figure 281: IP Input - Main IP reception routine: `linux/net/ipv4/ip_input.c`


```

401     iph = skb->nh.iph;
402
403
404     /*
405     *     RFC1122: 3.1.2.2 MUST silently discard any IP frame that fails the
406     *     checksum.
407     *
408     *     Is the datagram acceptable?
409     *
410     *     1.     Length at least the size of an ip header
411     *     2.     Version of 4
412     *     3.     Checksums correctly. [Speed optimisation for later, skip
413     *     loopback checksums]
414     *     4.     Doesn't have a bogus length
415     */
416     if (iph->ihl < 5 || iph->version != 4)
417         goto inhdr_error;
418
419     if (!pskb_may_pull(skb, iph->ihl*4))
420         goto inhdr_error;
421
422     if (ip_fast_csum((u8 *)iph, iph->ihl) != 0)
423         goto inhdr_error;
424
425     {
426         __u32 len = ntohs(iph->tot_len);
427         if (skb->len < len || len < (iph->ihl<<2))
428             goto inhdr_error;

```

Figure 282: IP Input - Main IP reception routine: `linux/net/ipv4/ip_input.c`

```

44 static inline int ip_forward_finish(struct sk_buff *skb)
45 {
46     struct ip_options *opt = &(IPCB(skb)->opt);
47
48     IP_INC_STATS_BH(IpForwDatagrams);
49
50     if (opt->optlen == 0) {
51 #ifdef CONFIG_NET_FASTROUTE
52         struct rtable *rt = (struct rtable*)skb->dst;
53
54         if (rt->rt_flags&RTCF_FAST && !netdev_fastroute_obstacles) {
55             struct dst_entry *old_dst;
56             unsigned h = ((*u8*)&rt->key.dst)^(*u8*)&rt->key.src)&
57                 NETDEV_FASTROUTE_HMASK;
58
59             write_lock_irq(&skb->dev->fastpath_lock);
60             old_dst = skb->dev->fastpath[h];
61             skb->dev->fastpath[h] = dst_clone(&rt->u.dst);
62             write_unlock_irq(&skb->dev->fastpath_lock);
63
64             dst_release(old_dst);
65         }
66 #endif
67         return (ip_send(skb));
68     }
69     ip_forward_options(skb);
70     return (ip_send(skb));
71 }

```

Figure 283: Internet Protocol - Forward last part:
linux/net/ipv4/ip_forward.c

```

73 int ip_forward(struct sk_buff *skb)
74 {
75     struct net_device *dev2;      /* Output device */
76     struct iphdr *iph;           /* Our header */
77     struct rtable *rt;          /* Route we use */
78     struct ip_options *opt = &(IPCB(skb)->opt);
79     unsigned short mtu;
80
81     if (IPCB(skb)->opt.router_alert && ip_call_ra_chain(skb))
82         return NET_RX_SUCCESS;
83
84     if (skb->pkt_type != PACKET_HOST)
85         goto drop;
86
87     skb->ip_summed = CHECKSUM_NONE;
88
89     /*
90      *   According to the RFC, we must first decrease the TTL field. If
91      *   that reaches zero, we must reply an ICMP control message telling
92      *   that the packet's lifetime expired.
93      */
94
95     iph = skb->nh.iph;
96     rt = (struct rtable*)skb->dst;
97
98     if (iph->ttl <= 1)
99         goto too_many_hops;
100
101     if (opt->is_strictroute && rt->rt_dst != rt->rt_gateway)
102         goto sr_failed ;
103
104     /*
105      *   Having picked a route we can now send the frame out
106      *   after asking the firewall permission to do so.
107      */
108
109     skb->priority = rt_tos2priority(iph->tos);
110     dev2 = rt->u.dst.dev;
111     mtu = rt->u.dst.pmtu;
112
113     /*
114      *   We now generate an ICMP HOST REDIRECT giving the route
115      *   we calculated.
116      */

```

Figure 284: Internet Protocol - Forward first bit:
linux/net/ipv4/ip_forward.c

```

118         ip_rt_send_redirect(skb);
119
120     /* We are about to mangle packet. Copy it! */
121     if (skb_cow(skb, dev2->hard_header_len))
122         goto drop;
123     iph = skb->nh.iph;
124
125     /* Decrease ttl after skb cow done */
126     ip_decrease_ttl(iph);
127
128     /*
129     * We now may allocate a new buffer, and copy the datagram into it.
130     * If the indicated interface is up and running, kick it.
131     */
132
133     if (skb->len > mtu && (ntohs(iph->frag_off) & IP_DF))
134         goto frag_needed;
135
136 #ifdef CONFIG_IP_ROUTE_NAT
137     if (rt->rt_flags & RTCF_NAT) {
138         if (ip_do_nat(skb)) {
139             kfree_skb(skb);
140             return NET_RX_BAD;
141         }
142     }
143 #endif
144
145     return NF_HOOK(PF_INET, NF_IP_FORWARD, skb, skb->dev, dev2,
146                  ip_forward_finish);
147
148 frag_needed:
149     IP_INC_STATS_BH(IpFragFails);
150     icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED, htonl(mtu));
151     goto drop;
152
153 sr_failed :
154     /*
155     *   Strict routing permits no gatewaying
156     */
157     icmp_send(skb, ICMP_DEST_UNREACH, ICMP_SR_FAILED, 0);
158     goto drop;
159
160 too_many_hops:
161     /* Tell the sender its packet died... */
162     icmp_send(skb, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL, 0);
163 drop:
164     kfree_skb(skb);
165     return NET_RX_DROP;

```

Figure 285: Internet Protocol - Forward first bit:
linux/net/ipv4/ip_forward.c

```

41 /* NOTE. Logic of IP defragmentation is parallel to corresponding IPv6
42 * code now. If you change something here, _PLEASE_ update ipv6/reassembly.c
43 * as well. Or notify me, at least. --ANK
44 */
45
46 /* Fragment cache limits. We will commit 256K at one time. Should we
47 * cross that limit we will prune down to 192K. This should cope with
48 * even the most extreme cases without allowing an attacker to measurably
49 * harm machine performance.
50 */
51 int sysctl_ipfrag_high_thresh = 256*1024;
52 int sysctl_ipfrag_low_thresh = 192*1024;
53
54 /* Important NOTE! Fragment queue must be destroyed before MSL expires.
55 * RFC791 is wrong proposing to prolongate timer each fragment arrival by TTL.
56 */
57 int sysctl_ipfrag_time = IP_FRAG_TIME;
58
59 struct ipfrag_skb_cb
60 {
61     struct inet_skb_parm  h;
62     int                   offset ;
63 };
64
65 #define FRAG_CB(skb) ((struct ipfrag_skb_cb*)((skb)->cb))
66
67 /* Describe an entry in the "incomplete datagrams" queue. */
68 struct ipq {
69     struct ipq      *next;           /* linked list pointers          */
70     u32              saddr;
71     u32              daddr;
72     u16              id;
73     u8               protocol;
74     u8               last_in ;
75 #define COMPLETE    4
76 #define FIRST_IN   2
77 #define LAST_IN    1
78
79     struct sk_buff  *fragments;     /* linked list of received fragments */
80     int              len;           /* total length of original datagram */
81     int              meat;
82     spinlock_t      lock;
83     atomic_t        refcnt;
84     struct timer_list timer;       /* when will this queue expire?    */
85     struct ipq      **pprev;
86     int              iif;
87     struct timeval  stamp;

```

Figure 286: Internet Protocol - Fragmentation- Data:
linux/net/ipv4/ip_fragment.c

```

89  /* Hash table. */
90
91
92  #define IPQ_HASHSZ 64
93
94  /* Per-bucket lock is easy to add now. */
95  static struct ipq *ipq_hash[IPQ_HASHSZ];
96  static rwlock_t ipfrag_lock = RW_LOCK_UNLOCKED;
97  int ip_frag_nqueues = 0;
98
99  static __inline__ void __ipq_unlink(struct ipq *qp)
100 {
101     if (qp->next)
102         qp->next->pprev = qp->pprev;
103     *qp->pprev = qp->next;
104     ip_frag_nqueues--;
105 }
106
107 static __inline__ void ipq_unlink(struct ipq *ipq)
108 {
109     write_lock(&ipfrag_lock);
110     __ipq_unlink(ipq);
111     write_unlock(&ipfrag_lock);
112 }
113
114 /*
115  * Was: (((id) >> 1) ^ (saddr) ^ (daddr) ^ (prot)) & (IPQ_HASHSZ - 1)
116  *
117  * I see, I see evil hand of bigendian mafia. On Intel all the packets hit
118  * one hash bucket with this hash function. 8)
119  */
120 static __inline__ unsigned int ipqhashfn(u16 id, u32 saddr, u32 daddr, u8 prot)
121 {
122     unsigned int h = saddr ^ daddr;
123
124     h ^= (h >> 16) ^ id;
125     h ^= (h >> 8) ^ prot;
126     return h & (IPQ_HASHSZ - 1);

```

Figure 287: Fragmentation - Hash Data: linux/net/ipv4/ip_fragment.c

```

129 atomic_t ip_frag_mem = ATOMIC_INIT(0); /* Memory used for fragments */
130
131
132 /* Memory Tracking Functions. */
133 extern __inline__ void frag_kfree_skb(struct sk_buff *skb)
134 {
135     atomic_sub(skb->truesize, &ip_frag_mem);
136     kfree_skb(skb);
137 }
138
139 extern __inline__ void frag_free_queue(struct ipq *qp)
140 {
141     atomic_sub(sizeof(struct ipq), &ip_frag_mem);
142     kfree(qp);
143 }
144
145 extern __inline__ struct ipq *frag_alloc_queue(void)
146 {
147     struct ipq *qp = kmalloc(sizeof(struct ipq), GFP_ATOMIC);
148
149     if (!qp)
150         return NULL;
151     atomic_add(sizeof(struct ipq), &ip_frag_mem);
152     return qp;

```

Figure 288: Fragmentation - memory tracking:
linux/net/ipv4/ip_fragment.c

```

154
155
156 /* Destruction primitives. */
157
158 /* Complete destruction of ipq. */
159 static void ip_frag_destroy (struct ipq *qp)
160 {
161     struct sk_buff *fp;
162
163     BUG_TRAP(qp->last_in & COMPLETE);
164     BUG_TRAP(del_timer(&qp->timer) == 0);
165
166     /* Release all fragment data. */
167     fp = qp->fragments;
168     while (fp) {
169         struct sk_buff *xp = fp->next;
170
171         frag_kfree_skb (fp);
172         fp = xp;
173     }
174
175     /* Finally, release the queue descriptor itself. */
176     frag_free_queue (qp);
177 }
178
179 static __inline__ void ipq_put (struct ipq *ipq)
180 {
181     if (atomic_dec_and_test (&ipq->refcnt))
182         ip_frag_destroy (ipq);
183 }
184
185 /* Kill ipq entry. It is not destroyed immediately,
186 * because caller (and someone more) holds reference count.
187 */
188 static __inline__ void ipq_kill (struct ipq *ipq)
189 {
190     if (del_timer (&ipq->timer))
191         atomic_dec (&ipq->refcnt);
192
193     if (!(ipq->last_in & COMPLETE)) {
194         ipq_unlink (ipq);
195         atomic_dec (&ipq->refcnt);
196         ipq->last_in |= COMPLETE;
197     }

```

Figure 289: Fragmentation - free up ipq: linux/net/ipv4/ip_fragment.c


```

199
200 /* Memory limiting on fragments. Evictor trashes the oldest
201 * fragment queue until we are back under the low threshold.
202 */
203 static void ip_evictor(void)
204 {
205     int i, progress;
206
207     do {
208         if (atomic_read(&ipfrag_mem) <= sysctl_ipfrag_low_thresh)
209             return;
210         progress = 0;
211         /* FIXME: Make LRU queue of frag heads. -DaveM */
212         for (i = 0; i < IPQ_HASHSZ; i++) {
213             struct ipq *qp;
214             if (ipq_hash[i] == NULL)
215                 continue;
216
217             read_lock(&ipfrag_lock);
218             if ((qp = ipq_hash[i]) != NULL) {
219                 /* find the oldest queue for this hash bucket */
220                 while (qp->next)
221                     qp = qp->next;
222                 atomic_inc(&qp->refcnt);
223                 read_unlock(&ipfrag_lock);
224
225                 spin_lock(&qp->lock);
226                 if (!(qp->last_in & COMPLETE))
227                     ipq_kill(qp);
228                 spin_unlock(&qp->lock);
229
230                 ipq_put(qp);
231                 IP_INC_STATS_BH(IpReasmFails);
232                 progress = 1;
233                 continue;
234             }
235             read_unlock(&ipfrag_lock);
236         }
237     } while (progress);
238 }

```

Figure 290: Fragmentation - free up space starting with oldest frags:
linux/net/ipv4/ip_fragment.c

```

240  /*
241  * Oops, a fragment queue timed out. Kill it and send an ICMP reply.
242  */
243  static void ip_expire(unsigned long arg)
244  {
245      struct ipq *qp = (struct ipq *) arg;
246
247      spin_lock(&qp->lock);
248
249      if (qp->last_in & COMPLETE)
250          goto out;
251
252      ipq_kill(qp);
253
254      IP_INC_STATS_BH(IpReasmTimeout);
255      IP_INC_STATS_BH(IpReasmFails);
256
257      if ((qp->last_in&FIRST_IN) && qp->fragments != NULL) {
258          struct sk_buff *head = qp->fragments;
259          /* Send an ICMP "Fragment Reassembly Timeout" message. */
260          if ((head->dev = dev_get_by_index(qp->iif)) != NULL) {
261              icmp_send(head, ICMP_TIME_EXCEEDED,
262                        ICMP_EXC_FRAGTIME, 0);
263              dev_put(head->dev);
264          }
265      }
266      out:
267      spin_unlock(&qp->lock);
268      ipq_put(qp);
269  }

```

Figure 291: Fragmentation - reassembly timeout, expire a packet:
linux/net/ipv4/ip_fragment.c

```

273 {
274     struct ipq *qp;
275
276     write_lock(&ipfrag_lock);
277 #ifdef CONFIG_SMP
278     /* With SMP race we have to recheck hash table, because
279      * such entry could be created on other cpu, while we
280      * promoted read lock to write lock.
281      */
282     for(qp = ipq_hash[hash]; qp; qp = qp->next) {
283         if(qp->id == qp_in->id    &&
284            qp->saddr == qp_in->saddr &&
285            qp->daddr == qp_in->daddr &&
286            qp->protocol == qp_in->protocol) {
287             atomic_inc(&qp->refcnt);
288             write_unlock(&ipfrag_lock);
289             qp_in->last_in |= COMPLETE;
290             ipq_put(qp_in);
291             return qp;
292         }
293     }
294 #endif
295     qp = qp_in;
296
297     if (!mod_timer(&qp->timer, jiffies + sysctl_ipfrag_time))
298         atomic_inc(&qp->refcnt);
299
300     atomic_inc(&qp->refcnt);
301     if ((qp->next = ipq_hash[hash]) != NULL)
302         qp->next->pprev = &qp->next;
303     ipq_hash[hash] = qp;
304     qp->pprev = &ipq_hash[hash];
305     ip_frag_nqueues++;
306     write_unlock(&ipfrag_lock);
307     return qp;
308 }

```

Figure 292: Fragmentation - create fragment: `linux/net/ipv4/ip_fragment.c`

```

311 static struct ipq * ip_frag_create (unsigned hash, struct iphdr *iph)
312 {
313     struct ipq *qp;
314
315     if ((qp = frag_alloc_queue()) == NULL)
316         goto out_nomem;
317
318     qp->protocol = iph->protocol;
319     qp->last_in = 0;
320     qp->id = iph->id;
321     qp->saddr = iph->saddr;
322     qp->daddr = iph->daddr;
323     qp->len = 0;
324     qp->meat = 0;
325     qp->fragments = NULL;
326     qp->iif = 0;
327
328     /* Initialize a timer for this entry. */
329     init_timer (&qp->timer);
330     qp->timer.data = (unsigned long) qp; /* pointer to queue */
331     qp->timer.function = ip_expire; /* expire function */
332     qp->lock = SPIN_LOCK_UNLOCKED;
333     atomic_set(&qp->refcnt, 1);
334
335     return ip_frag_intern (hash, qp);
336
337 out_nomem:
338     NETDEBUG (printk (KERN_ERR "ip_frag_create: no memory left.\n"));
339     return NULL;
340 }

```

Figure 293: Fragmentation - queue whole packet:
linux/net/ipv4/ip_fragment.c

```

343  * this IP datagram, and create new one, if nothing is found.
344  */
345  static inline struct ipq *ip_find(struct iphdr *iph)
346  {
347      __u16 id = iph->id;
348      __u32 saddr = iph->saddr;
349      __u32 daddr = iph->daddr;
350      __u8 protocol = iph->protocol;
351      unsigned int hash = ipqhashfn(id, saddr, daddr, protocol);
352      struct ipq *qp;
353
354      read_lock(&ipfrag_lock);
355      for(qp = ipq_hash[hash]; qp; qp = qp->next) {
356          if (qp->id == id &&
357              qp->saddr == saddr &&
358              qp->daddr == daddr &&
359              qp->protocol == protocol) {
360              atomic_inc(&qp->refcnt);
361              read_unlock(&ipfrag_lock);
362              return qp;
363          }
364      }
365      read_unlock(&ipfrag_lock);
366
367      return ip_frag_create(hash, iph);
368  }

```

Figure 294: Fragmentation - match a fragment to rest of datagram in partial queue: `linux/net/ipv4/ip_fragment.c`

```

371 static void ip_frag_queue(struct ipq *qp, struct sk_buff *skb)
372 {
373     struct sk_buff *prev, *next;
374     int flags, offset;
375     int ihl, end;
376
377     if (qp->last_in & COMPLETE)
378         goto err;
379
380     offset = ntohs(skb->nh.iph->frag_off);
381     flags = offset & ~IP_OFFSET;
382     offset &= IP_OFFSET;
383     offset <<= 3; /* offset is in 8-byte chunks */
384     ihl = skb->nh.iph->ihl * 4;
385
386     /* Determine the position of this fragment. */
387     end = offset + skb->len - ihl;
388
389     /* Is this the final fragment? */

```

Figure 295: Fragmentation - add segment to queue:
linux/net/ipv4/ip_fragment.c

```

391     /* If we already have some bits beyond end
392     * or have different end, the segment is corrupted.
393     */
394     if (end < qp->len ||
395         ((qp->last_in & LAST_IN) && end != qp->len))
396         goto err;
397     qp->last_in |= LAST_IN;
398     qp->len = end;
399 } else {
400     if (end&7) {
401         end &= ~7;
402         if (skb->ip_summed != CHECKSUM_UNNECESSARY)
403             skb->ip_summed = CHECKSUM_NONE;
404     }
405     if (end > qp->len) {
406         /* Some bits beyond end -> corruption. */
407         if (qp->last_in & LAST_IN)
408             goto err;
409         qp->len = end;
410     }
411 }
412 if (end == offset)
413     goto err;

```

Figure 296: Fragmentation - add segment to queue:
linux/net/ipv4/ip_fragment.c

```

414     if (pskb_pull(skb, ihl) == NULL)
415         goto err;
416     if (pskb_trim(skb, end - offset))
417         goto err;
418
419
420     /* Find out which fragments are in front and at the back of us
421     * in the chain of fragments so far. We must know where to put
422     * this fragment, right?
423     */
424     prev = NULL;
425     for(next = qp->fragments; next != NULL; next = next->next) {
426         if (FRAG_CB(next)->offset >= offset)
427             break; /* bingo! */
428         prev = next;
429     }
430
431     /* We found where to put this one. Check for overlap with
432     * preceding fragment, and, if needed, align things so that
433     * any overlaps are eliminated.
434     */
435     if (prev) {
436         int i = (FRAG_CB(prev)->offset + prev->len) - offset;
437
438         if (i > 0) {
439             offset += i;
440             if (end <= offset)
441                 goto err;
442             if (!pskb_pull(skb, i))
443                 goto err;
444             if (skb->ip_summed != CHECKSUM_UNNECESSARY)
445                 skb->ip_summed = CHECKSUM_NONE;
446         }
447     }

```

Figure 297: Fragmentation - add segment to queue:
linux/net/ipv4/ip_fragment.c


```

449 while (next && FRAG_CB(next)->offset < end) {
450     int i = end - FRAG_CB(next)->offset; /* overlap is 'i' bytes */
451
452     if (i < next->len) {
453         /* Eat head of the next overlapped fragment
454          * and leave the loop. The next ones cannot overlap.
455          */
456         if (!pskb_pull(next, i))
457             goto err;
458         FRAG_CB(next)->offset += i;
459         qp->meat -= i;
460         if (next->ip_summed != CHECKSUM_UNNECESSARY)
461             next->ip_summed = CHECKSUM_NONE;
462         break;
463     } else {
464         struct sk_buff * free_it = next;
465
466         /* Old fragmnet is completely overridden with
467          * new one drop it.
468          */
469         next = next->next;
470
471         if (prev)
472             prev->next = next;
473         else
474             qp->fragments = next;
475
476         qp->meat -= free_it->len;
477         frag_kfree_skb ( free_it );
478     }

```

Figure 298: Fragmentation - add segment to queue:
linux/net/ipv4/ip_fragment.c

```

480     FRAG_CB(skb)->offset = offset;
481
482
483     /* Insert this fragment in the chain of fragments. */
484     skb->next = next;
485     if (prev)
486         prev->next = skb;
487     else
488         qp->fragments = skb;
489
490     if (skb->dev)
491         qp->iif = skb->dev->ifindex;
492     skb->dev = NULL;
493     qp->stamp = skb->stamp;
494     qp->meat += skb->len;
495     atomic_add(skb->truesize, &ip_frag_mem);
496     if (offset == 0)
497         qp->last_in |= FIRST_IN;
498
499     return;
500
501 err:

```

Figure 299: Fragmentation - add segment to queue:
linux/net/ipv4/ip_fragment.c

```

504
505
506 /* Build a new IP datagram from all its fragments. */
507
508 static struct sk_buff *ip_frag_reasm(struct ipq *qp, struct net_device *dev)
509 {
510     struct iphdr *iph;
511     struct sk_buff *fp, *head = qp->fragments;
512     int len;
513     int ihlen;
514
515     ipq_kill(qp);
516
517     BUG_TRAP(head != NULL);
518     BUG_TRAP(FRAG_CB(head)->offset == 0);
519
520     /* Allocate a new buffer for the datagram. */
521     ihlen = head->nh.iph->ihl*4;
522     len = ihlen + qp->len;
523
524     if (len > 65535)
525         goto out_oversize;
526
527     /* Head of list must not be cloned. */
528     if (skb_cloned(head) && pskb_expand_head(head, 0, 0, GFP_ATOMIC))
529         goto out_nomem;
530
531     /* If the first fragment is fragmented itself, we split
532      * it to two chunks: the first with data and paged part

```

Figure 300: Fragmentation - put fragments together:
linux/net/ipv4/ip_fragment.c

```

534     if (skb_shinfo(head)->frag_list) {
535         struct sk_buff *clone;
536         int i, plen = 0;
537
538         if ((clone = alloc_skb(0, GFP_ATOMIC)) == NULL)
539             goto out_nomem;
540         clone->next = head->next;
541         head->next = clone;
542         skb_shinfo(clone)->frag_list = skb_shinfo(head)->frag_list;
543         skb_shinfo(head)->frag_list = NULL;
544         for (i=0; i<skb_shinfo(head)->nr_frags; i++)
545             plen += skb_shinfo(head)->frags[i].size;
546         clone->len = clone->data_len = head->data_len - plen;
547         head->data_len -= clone->len;
548         head->len -= clone->len;
549         clone->csum = 0;
550         clone->ip_summed = head->ip_summed;
551         atomic_add(clone->truesize, &ip_frag_mem);
552     }
553
554     skb_shinfo(head)->frag_list = head->next;
555     skb_push(head, head->data - head->nh.raw);
556     atomic_sub(head->truesize, &ip_frag_mem);

```

Figure 301: Fragmentation - put fragments together:
linux/net/ipv4/ip_fragment.c

```

559     head->data_len += fp->len;
560     head->len += fp->len;
561     if (head->ip_summed != fp->ip_summed)
562         head->ip_summed = CHECKSUM_NONE;
563     else if (head->ip_summed == CHECKSUM_HW)
564         head->csum = csum_add(head->csum, fp->csum);
565     head->truesize += fp->truesize;
566     atomic_sub(fp->truesize, &ip_frag_mem);
567 }
568
569     head->next = NULL;
570     head->dev = dev;
571     head->stamp = qp->stamp;
572
573     iph = head->nh.iph;
574     iph->frag_off = 0;
575     iph->tot_len = htons(len);
576     IP_INC_STATS_BH(IpReasmOKs);
577     qp->fragments = NULL;
578     return head;
579
580 out_nomem:
581     NETDEBUG(printk(KERN_ERR
582                 "IP: queue glue: no memory for gluing queue.%p\n",
583                 qp));
584     goto out_fail ;
585 out_oversize :
586     if ( net_ratelimit () )
587         printk(KERN_INFO
588                 "Oversized IP packet from.%d.%d.%d.%d.\n",
589                 NIPQUAD(qp->saddr));
590 out_fail :
591     IP_INC_STATS_BH(IpReasmFails);
592     return NULL;
593 }
594
595 /* Process an incoming IP datagram fragment. */
596 struct sk_buff *ip_defrag(struct sk_buff *skb)
597 {
598     struct iphdr *iph = skb->nh.iph;
599     struct ipq *qp;

```

Figure 302: Fragmentation - put fragments together:
linux/net/ipv4/ip_fragment.c

```

601     IP_INC_STATS_BH(IpReasmReqds);
602
603
604     /* Start by cleaning up the memory. */
605     if (atomic_read(&ip_frag_mem) > sysctl_ipfrag_high_thresh)
606         ip_evictor();
607
608     dev = skb->dev;
609
610     /* Lookup (or create) queue header */
611     if ((qp = ip_find(iph)) != NULL) {
612         struct sk_buff *ret = NULL;
613
614         spin_lock(&qp->lock);
615
616         ip_frag_queue(qp, skb);
617
618         if (qp->last_in == (FIRST_IN|LAST_IN) &&
619             qp->meat == qp->len)
620             ret = ip_frag_reasm(qp, dev);
621
622         spin_unlock(&qp->lock);
623         ipq_put(qp);
624         return ret;
625     }
626
627     IP_INC_STATS_BH(IpReasmFails);
628     kfree_skb(skb);
629     return NULL;
630 }

```

Figure 303: Fragmentation - deal with arrival of a fragment:
linux/net/ipv4/ip_fragment.c

```

25  /*
26  * Write options to IP header, record destination address to
27  * source route option, address of outgoing interface
28  * (we should already know it, so that this function is allowed be
29  * called only after routing decision) and timestamp,
30  * if we originate this datagram.
31  *
32  * daddr is real destination address, next hop is recorded in IP header.
33  * saddr is address of outgoing interface.
34  */
35
36 void ip_options_build(struct sk_buff *skb, struct ip_options *opt,
37                     u32 daddr, struct rtable *rt, int is_frag)
38 {
39     unsigned char *iph = skb->nh.raw;
40
41     memcpy(&(IPCB(skb)->opt), opt, sizeof(struct ip_options));
42     memcpy(iph+sizeof(struct iphdr), opt->_data, opt->optlen);
43     opt = &(IPCB(skb)->opt);
44     opt->is_data = 0;
45
46     if (opt->srr)
47         memcpy(iph+opt->srr+iph[opt->srr+1]-4, &daddr, 4);
48
49     if (!is_frag) {
50         if (opt->rr_needaddr)
51             ip_rt_get_source(iph+opt->rr+iph[opt->rr+2]-5, rt);
52         if (opt->ts_needaddr)
53             ip_rt_get_source(iph+opt->ts+iph[opt->ts+2]-9, rt);
54         if (opt->ts_needtime) {
55             struct timeval tv;
56             __u32 midtime;
57             do_gettimeofday(&tv);
58             midtime = htonl(((tv.tv_sec % 86400) * 1000 + tv.tv_usec / 1000));
59             memcpy(iph+opt->ts+iph[opt->ts+2]-5, &midtime, 4);
60         }
61         return;
62     }
63     if (opt->rr) {
64         memset(iph+opt->rr, IPOPT_NOP, iph[opt->rr+1]);
65         opt->rr = 0;
66         opt->rr_needaddr = 0;
67     }
68     if (opt->ts) {
69         memset(iph+opt->ts, IPOPT_NOP, iph[opt->ts+1]);
70         opt->ts = 0;
71         opt->ts_needaddr = opt->ts_needtime = 0;
72     }
73 }

```

Figure 304: Internet Protocol - Options - Build IP option header:
linux/net/ipv4/ip_options.c

```

75  /*
76  * Provided (sopt, skb) points to received options,
77  * build in dopt compiled option set appropriate for answering.
78  * i.e. invert SRR option, copy anothers,
79  * and grab room in RR/TS options.
80  *
81  * NOTE: dopt cannot point to skb.
82  */
83
84  int ip_options_echo(struct ip_options * dopt, struct sk_buff * skb)
85  {
86      struct ip_options *sopt;
87      unsigned char *sptr, *dptr;
88      int  soffset , doffset ;
89      int   optlen;
90      u32   daddr;
91
92      memset(dopt, 0, sizeof(struct ip_options));
93
94      dopt->is_data = 1;
95
96      sopt = &(IPCB(skb)->opt);
97
98      if (sopt->optlen == 0) {
99          dopt->optlen = 0;
100         return 0;
101     }
102
103     sptr = skb->nh.raw;
104     dptr = dopt->_data;
105
106     if (skb->dst)
107         daddr = ((struct rtable*)skb->dst)->rt_spec_dst;
108     else
109         daddr = skb->nh.iph->daddr;

```

Figure 305: IP Options - Echo: `linux/net/ipv4/ip_options.c`


```

111     if (sopt->rr) {
112         optlen = sptr[sopt->rr+1];
113         soffset = sptr[sopt->rr+2];
114         dopt->rr = dopt->optlen + sizeof(struct iphdr);
115         memcpy(dptr, sptr+sopt->rr, optlen);
116         if (sopt->rr_needaddr && soffset <= optlen) {
117             if (soffset + 3 > optlen)
118                 return -EINVAL;
119             dptr[2] = soffset + 4;
120             dopt->rr_needaddr = 1;
121         }
122         dptr += optlen;
123         dopt->optlen += optlen;
124     }
125     if (sopt->ts) {
126         optlen = sptr[sopt->ts+1];
127         soffset = sptr[sopt->ts+2];
128         dopt->ts = dopt->optlen + sizeof(struct iphdr);
129         memcpy(dptr, sptr+sopt->ts, optlen);

```

Figure 306: IP Options - Echo: `linux/net/ipv4/ip_options.c`

```

130     if ( soffset <= optlen) {
131         if (sopt->ts_needaddr) {
132             if ( soffset + 3 > optlen)
133                 return -EINVAL;
134             dopt->ts_needaddr = 1;
135             soffset += 4;
136         }
137         if (sopt->ts_needtime) {
138             if ( soffset + 3 > optlen)
139                 return -EINVAL;
140             if ((dptr[3]&0xF) != IPOPT_TS_PRESPEC) {
141                 dopt->ts_needtime = 1;
142                 soffset += 4;
143             } else {
144                 dopt->ts_needtime = 0;
145
146                 if ( soffset + 8 <= optlen) {
147                     _u32 addr;
148
149                     memcpy(&addr, sptr+soffset-1, 4);
150                     if (inet_addr_type(addr) != RTN_LOCAL
151                         ) {
152                         dopt->ts_needtime = 1;
153                         soffset += 8;
154                     }
155                 }
156             }
157             dptr[2] = soffset ;
158         }
159         dptr += optlen;
160         dopt->optlen += optlen;
161     }

```

Figure 307: IP Options - Echo: linux/net/ipv4/ip_options.c

```

162     if (sopt->srr) {
163         unsigned char * start = sptr+sopt->srr;
164         u32 faddr;
165
166         optlen = start [1];
167         soffset = start [2];
168         doffset = 0;
169         if ( soffset > optlen)
170             soffset = optlen + 1;
171         soffset -= 4;
172         if ( soffset > 3) {
173             memcpy(&faddr, &start[soffset-1], 4);
174             for ( soffset -=4, doffset=4; soffset > 3; soffset -=4, doffset+=4)
175                 memcpy(&dptr[doffset-1], &start[soffset-1], 4);
176             /*
177              * RFC1812 requires to fix illegal source routes.
178              */
179             if (memcmp(&skb->nh.iph->saddr, &start[soffset+3], 4) == 0)
180                 doffset -= 4;
181         }
182         if ( doffset > 3) {
183             memcpy(&start[doffset-1], &daddr, 4);
184             dopt->faddr = faddr;
185             dptr[0] = start [0];
186             dptr[1] = doffset +3;
187             dptr[2] = 4;
188             dptr += doffset+3;
189             dopt->srr = dopt->optlen + sizeof(struct iphdr);
190             dopt->optlen += doffset+3;
191             dopt->is_strictroute = sopt->is_strictroute;
192         }
193     }
194     while (dopt->optlen & 3) {
195         *dptr++ = IPOPT_END;
196         dopt->optlen++;
197     }
198     return 0;
199 }

```

Figure 308: IP Options - Echo: linux/net/ipv4/ip_options.c

```

201  /*
202  *   Options "fragmenting", just fill options not
203  *   allowed in fragments with NOOPs.
204  *   Simple and stupid 8), but the most efficient way.
205  */
206
207 void ip_options_fragment(struct sk_buff * skb)
208 {
209     unsigned char * optptr = skb->nh.raw;
210     struct ip_options * opt = &(IPCB(skb)->opt);
211     int l = opt->optlen;
212     int optlen;
213
214     while (l > 0) {
215         switch (*optptr) {
216             case IPOPT_END:
217                 return;
218             case IPOPT_NOOP:
219                 l--;
220                 optptr++;
221                 continue;
222         }
223         optlen = optptr[1];
224         if (optlen < 2 || optlen > l)
225             return;
226         if (!IPOPT_COPIED(*optptr))
227             memset(optptr, IPOPT_NOOP, optlen);
228         l -= optlen;
229         optptr += optlen;
230     }
231     opt->ts = 0;
232     opt->rr = 0;
233     opt->rr_needaddr = 0;
234     opt->ts_needaddr = 0;
235     opt->ts_needtime = 0;
236     return;
237 }

```

Figure 309: IP Options - zero out disallowed options in frags:
linux/net/ipv4/ip_options.c

```

239  /*
240  * Verify options and fill pointers in struct options.
241  * Caller should clear *opt, and set opt->data.
242  * If opt == NULL, then skb->data should point to IP header.
243  */
244
245  int ip_options_compile(struct ip_options * opt, struct sk_buff * skb)
246  {
247      int l;
248      unsigned char * iph;
249      unsigned char * optptr;
250      int optlen;
251      unsigned char * pp_ptr = NULL;
252      struct rtable *rt = skb ? (struct rtable*)skb->dst : NULL;
253
254      if (!opt) {
255          opt = &(IPCB(skb)->opt);
256          memset(opt, 0, sizeof(struct ip_options));
257          iph = skb->nh.raw;
258          opt->optlen = ((struct iphdr *)iph)->ihl*4 - sizeof(struct iphdr);
259          optptr = iph + sizeof(struct iphdr);
260          opt->is_data = 0;
261      } else {
262          optptr = opt->is_data ? opt->_data : (unsigned char*)&(skb->nh.iph
263          [1]);
264          iph = optptr - sizeof(struct iphdr);

```

Figure 310: IP Options - Validate inbound packet header options:
linux/net/ipv4/ip_options.c

```

266     for (l = opt->optlen; l > 0; ) {
267         switch (*optptr) {
268             case IPOPT_END:
269                 for (optptr++, l--, l>0; l--) {
270                     if (*optptr != IPOPT_END) {
271                         *optptr = IPOPT_END;
272                         opt->is_changed = 1;
273                     }
274                 }
275                 goto eol;
276             case IPOPT_NOOP:
277                 l--;
278                 optptr++;
279                 continue;
280         }
281         optlen = optptr[1];
282         if (optlen < 2 || optlen > 1) {
283             pp_ptr = optptr;
284             goto error;
285         }

```

Figure 311: IP Options - Validate inbound packet header options:
linux/net/ipv4/ip_options.c

```

286         switch (*optptr) {
287             case IPOPT_SSRR:
288             case IPOPT_LSRR:
289                 if (optlen < 3) {
290                     pp_ptr = optptr + 1;
291                     goto error;
292                 }
293                 if (optptr[2] < 4) {
294                     pp_ptr = optptr + 2;
295                     goto error;
296                 }
297                 /* NB: cf RFC-1812 5.2.4.1 */
298                 if (opt->srr) {
299                     pp_ptr = optptr;
300                     goto error;
301                 }
302                 if (!skb) {
303                     if (optptr[2] != 4 || optlen < 7 || ((optlen-3) & 3)) {
304                         pp_ptr = optptr + 1;
305                         goto error;
306                     }
307                     memcpy(&opt->faddr, &optptr[3], 4);
308                     if (optlen > 7)
309                         memmove(&optptr[3], &optptr[7], optlen-7);
310                 }
311                 opt->is_strictroute = (optptr[0] == IPOPT_SSRR);
312                 opt->srr = optptr - iph;
313                 break;

```

Figure 312: IP Options - Validate inbound packet header options:
linux/net/ipv4/ip_options.c

```

314         case IPOPT_RR:
315             if (opt->rr) {
316                 pp_ptr = optptr;
317                 goto error;
318             }
319             if (optlen < 3) {
320                 pp_ptr = optptr + 1;
321                 goto error;
322             }
323             if (optptr[2] < 4) {
324                 pp_ptr = optptr + 2;
325                 goto error;
326             }
327             if (optptr[2] <= optlen) {
328                 if (optptr[2]+3 > optlen) {
329                     pp_ptr = optptr + 2;
330                     goto error;
331                 }
332                 if (skb) {
333                     memcpy(&optptr[optptr[2]-1], &rt->rt_spec_dst
334                             , 4);
335                     opt->is_changed = 1;
336                 }
337                 optptr[2] += 4;
338                 opt->rr_needaddr = 1;
339             }
340             opt->rr = optptr - iph;
341             break;

```

Figure 313: IP Options - Validate inbound packet header options:
linux/net/ipv4/ip_options.c


```

341     case IPOPT_TIMESTAMP:
342         if (opt->ts) {
343             pp_ptr = optptr;
344             goto error;
345         }
346         if (optlen < 4) {
347             pp_ptr = optptr + 1;
348             goto error;
349         }
350         if (optptr[2] < 5) {
351             pp_ptr = optptr + 2;
352             goto error;
353         }
354         if (optptr[2] <= optlen) {
355             __u32 * timeptr = NULL;
356             if (optptr[2]+3 > optptr[1]) {
357                 pp_ptr = optptr + 2;
358                 goto error;
359             }
360             switch (optptr[3]&0xF) {
361                 case IPOPT_TS_TSONLY:
362                     opt->ts = optptr - iph;
363                     if (skb)
364                         timeptr = (__u32*)&optptr[optptr[2]-1];
365                     opt->ts_needtime = 1;
366                     optptr[2] += 4;
367                     break;
368                 case IPOPT_TS_TSANDADDR:
369                     if (optptr[2]+7 > optptr[1]) {
370                         pp_ptr = optptr + 2;
371                         goto error;
372                     }
373                     opt->ts = optptr - iph;
374                     if (skb) {
375                         memcpy(&optptr[optptr[2]-1], &rt->
376                             rt_spec_dst, 4);
377                         timeptr = (__u32*)&optptr[optptr[2]+3];
378                     }
379                     opt->ts_needaddr = 1;
380                     opt->ts_needtime = 1;
381                     optptr[2] += 8;
382                     break;

```

Figure 314: IP Options - Validate inbound packet header options:
linux/net/ipv4/ip_options.c

```

382         case IPOPT_TS_PRESPEC:
383             if (optptr[2]+7 > optptr[1]) {
384                 pp_ptr = optptr + 2;
385                 goto error;
386             }
387             opt->ts = optptr - iph;
388             {
389                 u32 addr;
390                 memcpy(&addr, &optptr[optptr[2]-1], 4);
391                 if (inet_addr_type(addr) ==
392                     RTN_UNICAST)
393                     break;
394                 if (skb)
395                     timeptr = (_u32*)&optptr[optptr
396                         [2]+3];
397             }
398             opt->ts_needtime = 1;
399             optptr[2] += 8;
400             break;
401         default:
402             if (!skb && !capable(CAP_NET_RAW)) {
403                 pp_ptr = optptr + 3;
404                 goto error;
405             }
406             break;
407     }
408     if (timeptr) {
409         struct timeval tv;
410         _u32 midtime;
411         do_gettimeofday(&tv);
412         midtime = htonl((tv.tv_sec % 86400) * 1000 + tv.
413             tv_usec / 1000);
414         memcpy(timeptr, &midtime, sizeof(_u32));
415         opt->is_changed = 1;
416     }
417 } else {
418     unsigned overflow = optptr[3]>>4;
419     if (overflow == 15) {
420         pp_ptr = optptr + 3;
421         goto error;
422     }
423     opt->ts = optptr - iph;
424     if (skb) {
425         optptr[3] = (optptr[3]&0xF)|((overflow+1)<<4);
426         opt->is_changed = 1;
427     }
428 }
429 break;

```

Figure 315: IP Options - Validate inbound packet header options:
linux/net/ipv4/ip_options.c

```

427         case IPOPT_RA:
428             if (optlen < 4) {
429                 pp_ptr = optptr + 1;
430                 goto error;
431             }
432             if (optptr[2] == 0 && optptr[3] == 0)
433                 opt->router_alert = optptr - iph;
434             break;
435         case IPOPT_SEC:
436         case IPOPT_SID:
437         default:
438             if (!skb && !capable(CAP_NET_RAW)) {
439                 pp_ptr = optptr;
440                 goto error;
441             }
442             break;
443     }
444     l -= optlen;
445     optptr += optlen;
446 }
447
448 eol:
449     if (!pp_ptr)
450         return 0;
451
452 error:
453     if (skb) {
454         icmp_send(skb, ICMP_PARAMETERPROB, 0, htonl((pp_ptr-iph)<<24));
455     }
456     return -EINVAL;
457 }

```

Figure 316: IP Options - end of options processing:
linux/net/ipv4/ip_options.c

```

460  /*
461  *      Undo all the changes done by ip_options_compile().
462  */
463
464  void ip_options_undo(struct ip_options * opt)
465  {
466      if (opt->srr) {
467          unsigned char * optptr = opt->_data+opt->srr-sizeof(struct iphdr);
468          memmove(optptr+7, optptr+3, optptr[1]-7);
469          memcpy(optptr+3, &opt->faddr, 4);
470      }
471      if (opt->rr_needaddr) {
472          unsigned char * optptr = opt->_data+opt->rr-sizeof(struct iphdr);
473          optptr[2] -= 4;
474          memset(&optptr[optptr[2]-1], 0, 4);
475      }
476      if (opt->ts) {
477          unsigned char * optptr = opt->_data+opt->ts-sizeof(struct iphdr);
478          if (opt->ts_needtime) {
479              optptr[2] -= 4;
480              memset(&optptr[optptr[2]-1], 0, 4);
481              if ((optptr[3]&0xF) == IPOPT_TS_PRESPEC)
482                  optptr[2] -= 4;
483          }
484          if (opt->ts_needaddr) {
485              optptr[2] -= 4;
486              memset(&optptr[optptr[2]-1], 0, 4);
487          }
488      }
489  }

```

Figure 317: IP Options - unmangle what previous function did!: `linux/net/ipv4/ip_options.c`

```

491 int ip_options_get(struct ip_options **optp, unsigned char *data, int optlen, int user)
492 {
493     struct ip_options *opt;
494
495     opt = kmalloc(sizeof(struct ip_options)+((optlen+3)&~3), GFP_KERNEL);
496     if (!opt)
497         return -ENOMEM;
498     memset(opt, 0, sizeof(struct ip_options));
499     if (optlen) {
500         if (user) {
501             if (copy_from_user(opt->_data, data, optlen)) {
502                 kfree(opt);
503                 return -EFAULT;
504             }
505         } else
506             memcpy(opt->_data, data, optlen);
507     }
508     while (optlen & 3)
509         opt->_data[optlen++] = IPOPT_END;
510     opt->optlen = optlen;
511     opt->is_data = 1;
512     opt->is_setbyuser = 1;
513     if (optlen && ip_options_compile(opt, NULL)) {
514         kfree(opt);
515         return -EINVAL;
516     }
517     *optp = opt;
518     return 0;
519 }

```

Figure 318: IP Options - make space for options:
linux/net/ipv4/ip_options.c

```

521 void ip_forward_options(struct sk_buff *skb)
522 {
523     struct ip_options * opt      = &(IPCB(skb)->opt);
524     unsigned char * optptr;
525     struct rtable *rt = (struct rtable*)skb->dst;
526     unsigned char *raw = skb->nh.raw;
527
528     if (opt->rr_needaddr) {
529         optptr = (unsigned char *)raw + opt->rr;
530         ip_rt_get_source (&optptr[optptr[2]-5], rt);
531         opt->is_changed = 1;
532     }
533     if (opt->srr_is_hit) {
534         int srrptr, srrspace;
535
536         optptr = raw + opt->srr;
537
538         for ( srrptr=optptr[2], srrspace = optptr[1];
539              srrptr <= srrspace;
540              srrptr += 4
541              ) {
542             if (srrptr + 3 > srrspace)
543                 break;
544             if (memcmp(&rt->rt_dst, &optptr[srrptr-1], 4) == 0)
545                 break;
546         }
547         if (srrptr + 3 <= srrspace) {
548             opt->is_changed = 1;
549             ip_rt_get_source (&optptr[srrptr-1], rt);
550             skb->nh.iph->daddr = rt->rt_dst;
551             optptr[2] = srrptr+4;
552         } else
553             printk(KERN_CRIT "ip_forward():_Argh!_Destination_Lost!\n");
554         if (opt->ts_needaddr) {
555             optptr = raw + opt->ts;
556             ip_rt_get_source (&optptr[optptr[2]-9], rt);
557             opt->is_changed = 1;
558         }
559     }
560     if (opt->is_changed) {
561         opt->is_changed = 0;
562         ip_send_check(skb->nh.iph);
563     }
564 }

```

Figure 319: IP Options - forwarding options - e.g. route record:
linux/net/ipv4/ip_options.c

```

568     struct ip_options *opt = &(IPCB(skb)->opt);
569     int srrspace, srrptr;
570     u32 nexthop;
571     struct iphdr *iph = skb->nh.iph;
572     unsigned char *optptr = skb->nh.raw + opt->srr;
573     struct rtable *rt = (struct rtable*)skb->dst;
574     struct rtable *rt2;
575     int err;
576
577     if (!opt->srr)
578         return 0;
579
580     if (skb->pkt_type != PACKET_HOST)
581         return -EINVAL;
582     if (rt->rt_type == RTN_UNICAST) {
583         if (!opt->is_strictroute)
584             return 0;
585         icmp_send(skb, ICMP_PARAMETERPROB, 0, htonl(16<<24));
586         return -EINVAL;
587     }
588     if (rt->rt_type != RTN_LOCAL)
589         return -EINVAL;

```

Figure 320: IP Options - receive SSR and process:
linux/net/ipv4/ip_options.c

```

591     for (srrptr=optptr[2], srrspace = optptr[1]; srrptr <= srrspace; srrptr += 4) {
592         if (srrptr + 3 > srrspace) {
593             icmp_send(skb, ICMP_PARAMETERPROB, 0, htonl((opt->srr
594                 +2)<<24));
595             return -EINVAL;
596         }
597         memcpy(&nexthop, &optptr[srrptr-1], 4);
598         rt = (struct rtable*)skb->dst;
599         skb->dst = NULL;
600         err = ip_route_input(skb, nexthop, iph->saddr, iph->tos, skb->dev);
601         rt2 = (struct rtable*)skb->dst;
602         if (err || (rt2->rt_type != RTN_UNICAST && rt2->rt_type !=
603             RTN_LOCAL)) {
604             ip_rt_put(rt2);
605             skb->dst = &rt->u.dst;
606             return -EINVAL;
607         }
608         ip_rt_put(rt);
609         if (rt2->rt_type != RTN_LOCAL)
610             break;
611         /* Superfast 8) loopback forward */
612         memcpy(&iph->daddr, &optptr[srrptr-1], 4);
613         opt->is_changed = 1;
614     }
615     if (srrptr <= srrspace) {
616         opt->srr_is_hit = 1;
617         opt->is_changed = 1;
618     }
619     return 0;

```

Figure 321: IP Options - receive SSR and process:
linux/net/ipv4/ip_options.c


```
48 #define IP_CMSG_PKTINFO 1
49 #define IP_CMSG_TTL 2
50 #define IP_CMSG_TOS 4
51 #define IP_CMSG_RECVOPTS 8
52 #define IP_CMSG_RETOPTS 16
```

Figure 322: Internet Protocol - Socket Glue -some spare constants:
linux/net/ipv4/ip_sockglue.c

```

54  /*
55  *   SOL_IP control messages.
56  */
57
58  static void ip_cmsg_rcv_pktinfo(struct msghdr *msg, struct sk_buff *skb)
59  {
60      struct in_pktinfo info;
61      struct rtable *rt = (struct rtable *)skb->dst;
62
63      info.ipi_addr.s_addr = skb->nh.iph->daddr;
64      if (rt) {
65          info.ipi_ifindex = rt->rt_iif;
66          info.ipi_spec_dst.s_addr = rt->rt_spec_dst;
67      } else {
68          info.ipi_ifindex = 0;
69          info.ipi_spec_dst.s_addr = 0;
70      }
71
72      put_cmsg(msg, SOL_IP, IP_PKTINFO, sizeof(info), &info);
73  }

```

Figure 323: IP Socket Glue - various CMSGs: `linux/net/ipv4/ip_sockglue.c`

```

75 static void ip_cmsg_recv_ttl(struct msghdr *msg, struct sk_buff *skb)
76 {
77     int ttl = skb->nh.iph->ttl;
78     put_cmsg(msg, SOL_IP, IP_TTL, sizeof(int), &ttl);
79 }
80
81 static void ip_cmsg_recv_tos(struct msghdr *msg, struct sk_buff *skb)
82 {
83     put_cmsg(msg, SOL_IP, IP_TOS, 1, &skb->nh.iph->tos);
84 }
85
86 static void ip_cmsg_recv_opts(struct msghdr *msg, struct sk_buff *skb)
87 {
88     if (IPCB(skb)->opt.optlen == 0)
89         return;
90
91     put_cmsg(msg, SOL_IP, IP_RECVOPTS, IPCB(skb)->opt.optlen, skb->nh.iph
92             +1);
93 }
94
95 void ip_cmsg_recv_retopts(struct msghdr *msg, struct sk_buff *skb)
96 {
97     unsigned char optbuf[sizeof(struct ip_options) + 40];
98     struct ip_options *opt = (struct ip_options*)optbuf;
99
100    if (IPCB(skb)->opt.optlen == 0)
101        return;
102
103    if (ip_options_echo(opt, skb)) {
104        msg->msg_flags |= MSG_CTRUNC;
105        return;
106    }
107    ip_options_undo(opt);
108
109    put_cmsg(msg, SOL_IP, IP_RETOPTS, opt->optlen, opt->_data);
110 }

```

Figure 324: IP Socket Glue - various CMSGs: `linux/net/ipv4/ip_sockglue.c`

```

113 void ip_cmsg_recv(struct msghdr *msg, struct sk_buff *skb)
114 {
115     unsigned flags = skb->sk->protinfo.af_inet.cmsg_flags;
116
117     /* Ordered by supposed usage frequency */
118     if (flags & 1)
119         ip_cmsg_recv_pktinfo(msg, skb);
120     if ((flags >>=1) == 0)
121         return;
122
123     if (flags & 1)
124         ip_cmsg_recv_ttl(msg, skb);
125     if ((flags >>=1) == 0)
126         return;
127
128     if (flags & 1)
129         ip_cmsg_recv_tos(msg, skb);
130     if ((flags >>=1) == 0)
131         return;
132
133     if (flags & 1)
134         ip_cmsg_recv_opts(msg, skb);
135     if ((flags >>=1) == 0)
136         return;
137
138     if (flags & 1)
139         ip_cmsg_recv_retopts(msg, skb);
140 }

```

Figure 325: IP Socket Glue - field cmsgs to right place:
linux/net/ipv4/ip_sockglue.c

```

142 int ip_cmsg_send(struct msghdr *msg, struct ipcm_cookie *ipc)
143 {
144     int err;
145     struct cmsghdr *cmsg;
146
147     for (cmsg = CMSG_FIRSTHDR(msg); cmsg; cmsg = CMSG_NXTHDR(msg, cmsg
148 )) {
149         if (cmsg->cmsg_len < sizeof(struct cmsghdr) ||
150             (unsigned long)((((char*)cmsg - (char*)msg->msg_control)
151                 + cmsg->cmsg_len) > msg->msg_controllen) {
152             return -EINVAL;
153         }
154         if (cmsg->cmsg_level != SOL_IP)
155             continue;
156         switch (cmsg->cmsg_type) {
157             case IP_RETOPTS:
158                 err = cmsg->cmsg_len - CMSG_ALIGN(sizeof(struct cmsghdr));
159                 err = ip_options_get(&ipc->opt, CMSG_DATA(cmsg), err < 40 ?
160                     err : 40, 0);
161                 if (err)
162                     return err;
163                 break;
164             case IP_PKTINFO:
165                 {
166                     struct in_pktinfo *info;
167                     if (cmsg->cmsg_len != CMSG_LEN(sizeof(struct in_pktinfo)))
168                         return -EINVAL;
169                     info = (struct in_pktinfo *)CMSG_DATA(cmsg);
170                     ipc->oif = info->ipi_ifindex;
171                     ipc->addr = info->ipi_spec_dst.s_addr;
172                     break;
173                 }
174             default:
175                 return -EINVAL;
176         }
177     }
178     return 0;
179 }

```

Figure 326: IP Socket Glue - send cmsgs: `linux/net/ipv4/ip_sockglue.c`

```

180 /* Special input handler for packets caught by router alert option.
181 They are selected only by protocol field, and then processed likely
182 local ones; but only if someone wants them! Otherwise, router
183 not running rsvpd will kill RSVP.
184
185 It is user level problem, what it will make with them.
186 I have no idea, how it will masquerade or NAT them (it is joke, joke :-)),
187 but receiver should be enough clever f.e. to forward mtrace requests,
188 sent to multicast group to reach destination designated router.
189 */
190 struct ip_ra_chain *ip_ra_chain;
191 rwlock_t ip_ra_lock = RW_LOCK_UNLOCKED;
192
193 int ip_ra_control(struct sock *sk, unsigned char on, void (*destructor)(struct sock *))
194 {
195     struct ip_ra_chain *ra, *new_ra, **rap;
196
197     if (sk->type != SOCK_RAW || sk->num == IPPROTO_RAW)
198         return -EINVAL;
199
200     new_ra = on ? kmalloc(sizeof(*new_ra), GFP_KERNEL) : NULL;

```

Figure 327: IP Socket Glue - Router Alert handler for user level:
linux/net/ipv4/ip_sockglue.c

```

202     write_unlock_bh(&ip_ra_lock);
203     for (rap = &ip_ra_chain; (ra=*rap) != NULL; rap = &ra->next) {
204         if (ra->sk == sk) {
205             if (on) {
206                 write_unlock_bh(&ip_ra_lock);
207                 if (new_ra)
208                     kfree(new_ra);
209                 return -EADDRINUSE;
210             }
211             *rap = ra->next;
212             write_unlock_bh(&ip_ra_lock);
213
214             if (ra->destructor)
215                 ra->destructor(sk);
216             sock_put(sk);
217             kfree(ra);
218             return 0;
219         }
220     }
221     if (new_ra == NULL) {
222         write_unlock_bh(&ip_ra_lock);
223         return -ENOBUFS;
224     }
225     new_ra->sk = sk;
226     new_ra->destructor = destructor;
227
228     new_ra->next = ra;
229     *rap = new_ra;
230     sock_hold(sk);
231     write_unlock_bh(&ip_ra_lock);
232
233     return 0;
234 }

```

Figure 328: IP Socket Glue - Router Alert handler for user level:
linux/net/ipv4/ip_sockglue.c

```

236 void ip_icmp_error(struct sock *sk, struct sk_buff *skb, int err,
237                  u16 port, u32 info, u8 *payload)
238 {
239     struct sock_exterr_skb *serr;
240
241     if (!sk->protinfo.af_inet.recvrr)
242         return;
243
244     skb = skb_clone(skb, GFP_ATOMIC);
245     if (!skb)
246         return;
247
248     serr = SKB_EXT_ERR(skb);
249     serr->ee.errno = err;
250     serr->ee.origin = SO_EE_ORIGIN_ICMP;
251     serr->ee.type = skb->h.icmph->type;
252     serr->ee.code = skb->h.icmph->code;
253     serr->ee.pad = 0;
254     serr->ee.info = info;
255     serr->ee.data = 0;
256     serr->addr_offset = (u8*)&(((struct iphdr*)(skb->h.icmph+1))->daddr) - skb
        ->nh.raw;
257     serr->port = port;
258
259     skb->h.raw = payload;
260     if (!skb_pull(skb, payload - skb->data) ||
261         sock_queue_err_skb(sk, skb))
262         kfree_skb(skb);
263 }

```

Figure 329: IP Socket Glue - ICMP error to socket error mapping:
[linux/net/ipv4/ip_sockglue.c](#)


```

266 {
267     struct sock_exterr_skb *serr;
268     struct iphdr *iph;
269     struct sk_buff *skb;
270
271     if (!sk->protinfo.af_inet.recvrr)
272         return;
273
274     skb = alloc_skb(sizeof(struct iphdr), GFP_ATOMIC);
275     if (!skb)
276         return;
277
278     iph = (struct iphdr*)skb_put(skb, sizeof(struct iphdr));
279     skb->nh.iph = iph;
280     iph->daddr = daddr;
281
282     serr = SKB_EXT_ERR(skb);
283     serr->ee_errno = err;
284     serr->ee_origin = SO_EE_ORIGIN_LOCAL;
285     serr->ee_type = 0;
286     serr->ee_code = 0;
287     serr->ee_pad = 0;
288     serr->ee_info = info;
289     serr->ee_data = 0;
290     serr->addr_offset = (u8*)&iph->daddr - skb->nh.raw;
291     serr->port = port;
292
293     skb->h.raw = skb->tail;
294     __skb_pull(skb, skb->tail - skb->data);
295
296     if (sock_queue_err_skb(sk, skb))
297         kfree_skb(skb);
298 }

```

Figure 330: IP Socket Glue - local error problem:
linux/net/ipv4/ip_sockglue.c

```

301  *      Handle MSG_ERRQUEUE
302  */
303  int ip_rcv_error(struct sock *sk, struct msghdr *msg, int len)
304  {
305      struct sock_exterr_skb *serr;
306      struct sk_buff *skb, *skb2;
307      struct sockaddr_in *sin;
308      struct {
309          struct sock_extended_err ee;
310          struct sockaddr_in  offender;
311      } errhdr;
312      int err;
313      int copied;
314
315      err = -EAGAIN;
316      skb = skb_dequeue(&sk->error_queue);
317      if (skb == NULL)
318          goto out;
319
320      copied = skb->len;
321      if (copied > len) {
322          msg->msg_flags |= MSG_TRUNC;
323          copied = len;
324      }
325      err = skb_copy_datagram_iovec(skb, 0, msg->msg_iov, copied);
326      if (err)
327          goto out_free_skb;
328
329      sock_rcv_timestamp(msg, sk, skb);
330
331      serr = SKB_EXT_ERR(skb);
332
333      sin = (struct sockaddr_in *)msg->msg_name;
334      if (sin) {
335          sin->sin_family = AF_INET;
336          sin->sin_addr.s_addr = *(u32*)(skb->nh.raw + serr->addr_offset);
337          sin->sin_port = serr->port;
338          memset(&sin->sin_zero, 0, sizeof(sin->sin_zero));
339      }

```

Figure 331: IP Socket Glue - map IP reception errors to socket:
linux/net/ipv4/ip_sockglue.c

```

341     memcpy(&errhdr.ee, &serr->ee, sizeof(struct sock_extended_err));
342     sin = &errhdr.offender;
343     sin->sin_family = AF_UNSPEC;
344     if (serr->ee.ee_origin == SO_EE_ORIGIN_ICMP) {
345         sin->sin_family = AF_INET;
346         sin->sin_addr.s_addr = skb->nh.iph->saddr;
347         sin->sin_port = 0;
348         memset(&sin->sin_zero, 0, sizeof(sin->sin_zero));
349         if (sk->proto.af_inet.cmsg_flags)
350             ip_cmsg_rcv(msg, skb);
351     }
352
353     put_cmsg(msg, SOL_IP, IP_RECVERR, sizeof(errhdr), &errhdr);
354
355     /* Now we could try to dump offended packet options */
356
357     msg->msg_flags |= MSG_ERRQUEUE;
358     err = copied;
359
360     /* Reset and regenerate socket error */
361     spin_lock_irq(&sk->error_queue.lock);
362     sk->err = 0;
363     if ((skb2 = skb_peek(&sk->error_queue)) != NULL) {
364         sk->err = SKB_EXT_ERR(skb2->ee.ee_errno);
365         spin_unlock_irq(&sk->error_queue.lock);
366         sk->error_report(sk);
367     } else {
368         spin_unlock_irq(&sk->error_queue.lock);
369     }
370
371 out_free_skb:
372     kfree_skb(skb);
373 out:

```

Figure 332: IP Socket Glue - map IP reception errors to socket:
linux/net/ipv4/ip_sockglue.c

```

376
377
378 /*
379 *   Socket option code for IP. This is the end of the line after any TCP,UDP
    etc options on
380 *   an IP socket.
381 */
382
383 int ip_setsockopt(struct sock *sk, int level, int optname, char *optval, int optlen)
384 {
385     int val=0,err;
386
387     if (level != SOL_IP)
388         return -ENOPROTOPT;
389
390     if (((1<<optname) & ((1<<IP_PKTINFO) | (1<<IP_RECVTTL) |
391         (1<<IP_RECVOPTS) | (1<<IP_RECVTOS) |
392         (1<<IP_RETOPTS) | (1<<IP_TOS) |
393         (1<<IP_TTL) | (1<<IP_HDRINCL) |
394         (1<<IP_MTU_DISCOVER) | (1<<IP_RECVERR) |
395         (1<<IP_ROUTER_ALERT) | (1<<IP_FREEBIND)))) ||
396         optname == IP_MULTICAST_TTL ||
397         optname == IP_MULTICAST_LOOP) {
398         if (optlen >= sizeof(int)) {
399             if (get_user(val, (int *) optval))
400                 return -EFAULT;
401         } else if (optlen >= sizeof(char)) {
402             unsigned char ucval;
403
404             if (get_user(ucval, (unsigned char *) optval))
405                 return -EFAULT;
406             val = (int) ucval;

```

Figure 333: IP Socket Glue - setsockopt entry for IP:
linux/net/ipv4/ip_sockglue.c

```

408     }
409
410     /* If optlen==0, it is equivalent to val == 0 */
411
412     #ifdef CONFIG_IP_MROUTE
413     if (optname >= MRT_BASE && optname <= (MRT_BASE + 10))
414         return ip_mroute_setsockopt(sk,optname,optval,optlen);
415     #endif
416
417     err = 0;
418     lock_sock(sk);
419
420     switch (optname) {
421     case IP_OPTIONS:
422     {
423         struct ip_options * opt = NULL;
424         if (optlen > 40 || optlen < 0)
425             goto e_inval;
426         err = ip_options_get(&opt, optval, optlen, 1);
427         if (err)
428             break;
429         if (sk->type == SOCK_STREAM) {
430             struct tcp_opt *tp = &sk->tp_pinfo.af_tcp;
431             #if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
432             if (sk->family == PF_INET ||
433                 (!(1 << sk->state) & (TCPF_LISTEN|TCPF_CLOSE))
434                 && sk->daddr != LOOPBACK4_IPV6)) {
435             #endif
436                 if (opt)
437                     tp->ext_header_len = opt->optlen;
438                 tcp_sync_mss(sk, tp->pmtu_cookie);
439             #if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
440             }
441             #endif
442         }
443         opt = xchg(&sk->protinfo.af_inet.opt, opt);
444         if (opt)
445             kfree(opt);

```

Figure 334: IP Socket Glue - setsockopt entry for IP:
linux/net/ipv4/ip_sockglue.c

```

446         break;
447     }
448     case IP_PKTINFO:
449         if (val)
450             sk->protinfo.af_inet.cmsg_flags |= IP_CMSG_PKTINFO;
451         else
452             sk->protinfo.af_inet.cmsg_flags &= ~IP_CMSG_PKTINFO;
453         break;
454     case IP_RECVTTL:
455         if (val)
456             sk->protinfo.af_inet.cmsg_flags |= IP_CMSG_TTL;
457         else
458             sk->protinfo.af_inet.cmsg_flags &= ~IP_CMSG_TTL;
459         break;
460     case IP_RECVTOS:
461         if (val)
462             sk->protinfo.af_inet.cmsg_flags |= IP_CMSG_TOS;
463         else
464             sk->protinfo.af_inet.cmsg_flags &= ~IP_CMSG_TOS;
465         break;
466     case IP_RECVOPTS:
467         if (val)
468             sk->protinfo.af_inet.cmsg_flags |=
469                 IP_CMSG_RECVOPTS;
470         else
471             sk->protinfo.af_inet.cmsg_flags &= ~
472                 IP_CMSG_RECVOPTS;
473         break;
474     case IP_RETOPTS:
475         if (val)
476             sk->protinfo.af_inet.cmsg_flags |= IP_CMSG_RETOPTS;
477         else

```

Figure 335: IP Socket Glue - setsockopt entry for IP:
linux/net/ipv4/ip_sockglue.c

```

476         sk->protinfo.af_inet.cmsg_flags &= ~
477             IP_CMSG_RETOPTS;
478         break;
479         case IP_TOS: /* This sets both TOS and Precedence */
480             /* Reject setting of unused bits */
481             #ifndef CONFIG_INET_ECN
482                 if (val & ~(IPTOS_TOS_MASK|IPTOS_PREC_MASK))
483                     goto e_inval;
484             #else
485                 if (sk->type == SOCK_STREAM) {
486                     val &= ~3;
487                     val |= sk->protinfo.af_inet.tos & 3;
488                 }
489             #endif
490             if (IPTOS_PREC(val) >= IPTOS_PREC_CRITIC_ECP &&
491                 !capable(CAP_NET_ADMIN)) {
492                 err = -EPERM;
493                 break;
494             }
495             if (sk->protinfo.af_inet.tos != val) {
496                 sk->protinfo.af_inet.tos=val;
497                 sk->priority = rt_tos2priority(val);
498                 sk_dst_reset(sk);
499             }
500             break;
501         case IP_TTL:
502             if (optlen<1)
503                 goto e_inval;
504             if (val==-1)
505                 val = sysctl_ip_default_ttl;
506             if (val<1||val>255)
507                 goto e_inval;
508             sk->protinfo.af_inet.ttl=val;
509             break;
510         case IP_HDRINCL:
511             if (sk->type!=SOCK_RAW) {
512                 err = -ENOPROTOOPT;
513                 break;
514             }
515             sk->protinfo.af_inet.hdrincl=val?1:0;
516             break;
517         case IP_MTU_DISCOVER:
518             if (val<0 || val>2)
519                 goto e_inval;
520             sk->protinfo.af_inet.pmtudisc = val;
521             break;
522         case IP_RECVERR:
523             sk->protinfo.af_inet.recvrr = !!val;
524             if (!val)

```

Figure 336: IP Socket Glue - setsockopt entry for IP: linux/net/ipv4/ip_sockglue.c

```

524         skb_queue_purge(&sk->error_queue);
525         break;
526     case IP_MULTICAST_TTL:
527         if (sk->type == SOCK_STREAM)
528             goto e_inval;
529         if (optlen < 1)
530             goto e_inval;
531         if (val == -1)
532             val = 1;
533         if (val < 0 || val > 255)
534             goto e_inval;
535         sk->protinfo.af_inet.mc_ttl = val;
536         break;
537     case IP_MULTICAST_LOOP:
538         if (optlen < 1)
539             goto e_inval;

```

Figure 337: IP Socket Glue - setsockopt entry for IP:
linux/net/ipv4/ip_sockglue.c


```

540         sk->protinfo.af_inet.mc_loop = val ? 1 : 0;
541         break;
542     case IP_MULTICAST_IF:
543     {
544         struct ip_mreqn mreq;
545         struct net_device *dev = NULL;
546
547         if (sk->type == SOCK_STREAM)
548             goto e_inval;
549
550         /*
551          *   Check the arguments are allowable
552          */
553         err = -EFAULT;
554         if (optlen >= sizeof(struct ip_mreqn)) {
555             if (copy_from_user(&mreq,optval,sizeof(mreq)))
556                 break;
557         } else {
558             memset(&mreq, 0, sizeof(mreq));
559             if (optlen >= sizeof(struct in_addr) &&
560                 copy_from_user(&mreq.imr_address,optval,sizeof(struct
                    in_addr)))

```

Figure 338: IP Socket Glue - setsockopt entry for IP:
linux/net/ipv4/ip_sockglue.c

```

562     }
563
564     if (!mreq.imr_ifindex) {
565         if (mreq.imr_address.s_addr == INADDR_ANY) {
566             sk->protoinfo.af_inet.mc_index = 0;
567             sk->protoinfo.af_inet.mc_addr = 0;
568             err = 0;
569             break;
570         }
571         dev = ip_dev_find(mreq.imr_address.s_addr);
572         if (dev) {
573             mreq.imr_ifindex = dev->ifindex;
574             dev_put(dev);
575         }
576     } else
577         dev = _dev_get_by_index(mreq.imr_ifindex);
578
579
580     err = -EADDRNOTAVAIL;
581     if (!dev)
582         break;
583
584     err = -EINVAL;
585     if (sk->bound_dev_if && mreq.imr_ifindex != sk->bound_dev_if)
586         break;
587
588     sk->protoinfo.af_inet.mc_index = mreq.imr_ifindex;
589     sk->protoinfo.af_inet.mc_addr = mreq.imr_address.s_addr;
590     err = 0;

```

Figure 339: IP Socket Glue - setsockopt entry for IP: linux/net/ipv4/ip_sockglue.c

```

592     }
593
594     case IP_ADD_MEMBERSHIP:
595     case IP_DROP_MEMBERSHIP:
596     {
597         struct ip_mreqn mreq;
598
599         if (optlen < sizeof(struct ip_mreq))
600             goto e_inval;
601         err = -EFAULT;
602         if (optlen >= sizeof(struct ip_mreqn)) {
603             if (copy_from_user(&mreq,optval,sizeof(mreq)))
604                 break;
605         } else {
606             memset(&mreq, 0, sizeof(mreq));
607             if (copy_from_user(&mreq,optval,sizeof(struct ip_mreqn)))
608                 break;
609         }
610
611         if (optname == IP_ADD_MEMBERSHIP)
612             err = ip_mc_join_group(sk,&mreq);
613         else
614             err = ip_mc_leave_group(sk,&mreq);
615         break;
616     }
617     case IP_ROUTER_ALERT:
618         err = ip_ra_control(sk, val ? 1 : 0, NULL);
619         break;
620
621     case IP_FREEBIND:
622         if (optlen < 1)
623             goto e_inval;
624         sk->protinfo.af_inet.freebind = !!val;
625         break;
626
627     default :
628     #ifdef CONFIG_NETFILTER
629         err = nf_setsockopt(sk, PF_INET, optname, optval,
630                             optlen);
631     #else
632         err = -ENOPROTOOPT;
633     #endif
634         break;
635     }
636     release_sock(sk);
637     return err;
638
639 e_inval :
640     release_sock(sk);

```

Figure 340: IP Socket Glue - setsockopt entry for IP:
linux/net/ipv4/ip_sockglue.c

```

642 }
643
644 /*
645  *   Get the options. Note for future reference. The GET of IP options gets
        the
646  *   _received_ ones. The set sets the _sent_ ones.
647  */
648
649 int ip_getsockopt(struct sock *sk, int level, int optname, char *optval, int *optlen)
650 {
651     int val;
652     int len;
653
654     if (level != SOL_IP)
655         return -EOPNOTSUPP;
656
657     #ifdef CONFIG_IP_MROUTE
658         if (optname >= MRT_BASE && optname <= MRT_BASE + 10)
659             {
660                 return ip_mroute_getsockopt(sk, optname, optval, optlen);
661             }
662     #endif
663
664     if (get_user(len, optlen))
665         return -EFAULT;

```

Figure 341: IP Socket Glue - getsockopt entry for IP:
linux/net/ipv4/ip_sockglue.c

```

667         return -EINVAL;
668
669     lock_sock(sk);
670
671     switch(optname) {
672         case IP_OPTIONS:
673             {
674                 unsigned char optbuf[sizeof(struct ip_options)+40];
675                 struct ip_options * opt = (struct ip_options*)optbuf;
676                 opt->optlen = 0;
677                 if (sk->protinfo.af_inet.opt)
678                     memcpy(optbuf, sk->protinfo.af_inet.opt,
679                            sizeof(struct ip_options)+
680                            sk->protinfo.af_inet.opt->optlen);
681                 release_sock(sk);
682
683                 if (opt->optlen == 0)
684                     return put_user(0, optlen);
685
686                 ip_options_undo(opt);
687
688                 len=min(len, opt->optlen);
689                 if (put_user(len, optlen))
690                     return -EFAULT;
691                 if (copy_to_user(optval, opt->_data, len))
692                     return -EFAULT;
693                 return 0;
694             }
695         case IP_PKTINFO:
696             val = (sk->protinfo.af_inet.msg_flags & IP_CMSG_PKTINFO)
697                 != 0;
698             break;
699         case IP_RECVTTL:
700             val = (sk->protinfo.af_inet.msg_flags & IP_CMSG_TTL) != 0;
701             break;
702         case IP_RECVTOS:
703             val = (sk->protinfo.af_inet.msg_flags & IP_CMSG_TOS) != 0;
704             break;
705         case IP_RECVOPTS:
706             val = (sk->protinfo.af_inet.msg_flags & IP_CMSG_RECVOPTS)
707                 != 0;
708             break;
709         case IP_RETOPTS:
710             val = (sk->protinfo.af_inet.msg_flags & IP_CMSG_RETOPTS)
711                 != 0;
712             break;
713         case IP_TOS:
714             val=sk->protinfo.af_inet.tos;
715             break;
716         case IP_TTL:
717             val=sk->protinfo.af_inet.ttl;

```

Figure 342: IP Socket Glue - getsockopt entry for IP:
linux/net/ipv4/ip_sockglue.c

```

715         break;
716     case IP_HDRINCL:
717         val=sk->protinfo.af_inet.hdrincl;
718         break;
719     case IP_MTU_DISCOVER:
720         val=sk->protinfo.af_inet.pmtudisc;
721         break;
722     case IP_MTU:
723     {
724         struct dst_entry *dst;
725         val = 0;
726         dst = sk_dst_get(sk);
727         if (dst) {
728             val = dst->pmtu;
729             dst_release(dst);
730         }
731         if (!val) {
732             release_sock(sk);
733             return -ENOTCONN;
734         }
735         break;
736     }
737     case IP_RECVERR:
738         val=sk->protinfo.af_inet.recvrr;
739         break;
740     case IP_MULTICAST_TTL:
741         val=sk->protinfo.af_inet.mc_ttl;
742         break;
743     case IP_MULTICAST_LOOP:
744         val=sk->protinfo.af_inet.mc_loop;
745         break;
746     case IP_MULTICAST_IF:
747     {
748         struct in_addr addr;
749         len = min(len, sizeof(struct in_addr));
750         addr.s_addr = sk->protinfo.af_inet.mc_addr;
751         release_sock(sk);
752
753         if (put_user(len, optlen))
754             return -EFAULT;

```

Figure 343: IP Socket Glue - getsockopt entry for IP:
linux/net/ipv4/ip_sockglue.c

```

755         if (copy_to_user((void *)optval, &addr, len))
756             return -EFAULT;
757         return 0;
758     }
759     case IP_PKTOPTIONS:
760     {
761         struct msghdr msg;
762
763         release_sock(sk);
764
765         if (sk->type != SOCK_STREAM)
766             return -ENOPROTOOPT;
767
768         msg.msg_control = optval;
769         msg.msg_controllen = len;
770         msg.msg_flags = 0;
771
772         if (sk->protoinfo.af_inet.cmsg_flags&IP_CMSG_PKTINFO) {
773             struct in_pktinfo info;
774
775             info.ipi_addr.s_addr = sk->rcv_saddr;
776             info.ipi_spec_dst.s_addr = sk->rcv_saddr;
777             info.ipi_ifindex = sk->protoinfo.af_inet.mc_index;
778             put_cmsg(&msg, SOL_IP, IP_PKTINFO, sizeof(info), &info
779                 );
780         }
781         if (sk->protoinfo.af_inet.cmsg_flags&IP_CMSG_TTL) {
782             int hlim = sk->protoinfo.af_inet.mc_ttl;
783             put_cmsg(&msg, SOL_IP, IP_TTL, sizeof(hlim), &hlim);

```

Figure 344: IP Socket Glue - getsockopt entry for IP:
linux/net/ipv4/ip_sockglue.c

```

783         }
784         len -= msg.msg_controllen;
785         return put_user(len, optlen);
786     }
787     case IP_FREEBIND:
788         val = sk->protinfo.af_inet.freebind;
789         break;
790     default:
791 #ifdef CONFIG_NETFILTER
792         val = nf_getsockopt(sk, PF_INET, optname, optval,
793                             &len);
794         release_sock(sk);
795         if (val >= 0)
796             val = put_user(len, optlen);
797         return val;
798 #else
799         release_sock(sk);
800         return -ENOPROTOOPT;
801 #endif
802     }
803     release_sock(sk);
804
805     if (len < sizeof(int) && len > 0 && val >= 0 && val < 255) {
806         unsigned char ucval = (unsigned char)val;
807         len = 1;
808         if (put_user(len, optlen))
809             return -EFAULT;
810         if (copy_to_user(optval, &ucval, 1))
811             return -EFAULT;
812     } else {
813         len = min(sizeof(int), len);
814         if (put_user(len, optlen))
815             return -EFAULT;
816         if (copy_to_user(optval, &val, len))

```

Figure 345: IP Socket Glue - getsockopt entry for IP:
linux/net/ipv4/ip_sockglue.c


```
70
71 #include <linux/types.h>
72 #include <linux/string.h>
73 #include <linux/kernel.h>
74 #include <linux/sched.h>
75 #include <linux/config.h>
76 #include <linux/socket.h>
77 #include <linux/sockios.h>
78 #include <linux/errno.h>
79 #include <linux/in.h>
```

Figure 346: Address Resolution Protocol - Data : `linux/net/ipv4/arp.c`

```

127 {
128     AF_INET,
129     NULL,
130     arp_solicit ,
131     arp_error_report ,
132     neigh_resolve_output ,
133     neigh_connected_output ,
134     dev_queue_xmit,
135     dev_queue_xmit
136 };
137
138 static struct neigh_ops arp_hh_ops =
139 {
140     AF_INET,
141     NULL,
142     arp_solicit ,
143     arp_error_report ,
144     neigh_resolve_output ,
145     neigh_resolve_output ,
146     dev_queue_xmit,
147     dev_queue_xmit
148 };
149
150 static struct neigh_ops arp_direct_ops =
151 {
152     AF_INET,
153     NULL,
154     NULL,
155     NULL,
156     dev_queue_xmit,
157     dev_queue_xmit,
158     dev_queue_xmit,

```

Figure 347: ARP Neighbour Operations : `linux/net/ipv4/arp.c`

```

160 };
161
162 struct neigh_ops arp_broken_ops =
163 {
164     AF_INET,
165     NULL,
166     arp_solicit ,
167     arp_error_report ,
168     neigh_compat_output,
169     neigh_compat_output,
170     dev_queue_xmit,
171     dev_queue_xmit,
172 };
173
174 struct neigh_table arp_tbl =
175 {
176     NULL,
177     AF_INET,
178     sizeof (struct neighbour) + 4,
179     4,
180     arp_hash,
181     arp_constructor ,
182     NULL,

```

Figure 348: ARP Neighbour Operations : `linux/net/ipv4/arp.c`

```

184     parp_redo,
185     "arp_cache",
186     { NULL, NULL, &arp_tbl, 0, NULL, NULL,
187       30*HZ, 1*HZ, 60*HZ, 30*HZ, 5*HZ, 3, 3, 0, 3, 1*HZ, (8*HZ)/10, 64, 1*
188         HZ },
189     30*HZ, 128, 512, 1024,
190 };
191 int arp_mc_map(u32 addr, u8 *haddr, struct net_device *dev, int dir)
192 {
193     switch (dev->type) {
194     case ARPHRD_ETHER:
195     case ARPHRD_FDDI:
196     case ARPHRD_IEEE802:
197         ip_eth_mc_map(addr, haddr) ;
198         return 0 ;
199     case ARPHRD_IEEE802_TR:

```

Figure 349: ARP Table Structure : `linux/net/ipv4/arp.c`

```

201         return 0;
202     default:
203         if (dir) {
204             memcpy(haddr, dev->broadcast, dev->addr_len);
205             return 0;
206         }
207     }
208     return -EINVAL;
209 }
210
211
212 static u32 arp_hash(const void *pkey, const struct net_device *dev)
213 {
214     u32 hash_val;
215
216     hash_val = *(u32*)pkey;
217     hash_val ^= (hash_val>>16);
218     hash_val ^= hash_val>>8;
219     hash_val ^= hash_val>>3;

```

Figure 350: ARP Map MAC type/IEEE type: `linux/net/ipv4/arp.c`

```
222     return hash_val;
223 }
224
225 static int arp_constructor(struct neighbour *neigh)
226 {
227     u32 addr = *(u32*)neigh->primary_key;
228     struct net_device *dev = neigh->dev;
229     struct in_device *in_dev = in_dev_get(dev);
230
231     if (in_dev == NULL)
232         return -EINVAL;
```

Figure 351: ARP Hash Function: `linux/net/ipv4/arp.c`

```

235     if (in_dev->arp_parms)
236         neigh->parms = in_dev->arp_parms;
237
238     in_dev_put(in_dev);
239
240     if (dev->hard_header == NULL) {
241         neigh->nud_state = NUD_NOARP;
242         neigh->ops = &arp_direct_ops;
243         neigh->output = neigh->ops->queue_xmit;
244     } else {
245         /* Good devices (checked by reading texts, but only Ethernet is
246            tested)
247
248            ARPHRD_ETHER: (ethernet, apfddi)
249            ARPHRD_FDDI: (fddi)
250            ARPHRD_IEEE802: (tr)
251            ARPHRD_METRICOM: (strip)
252            ARPHRD_ARCNET:
253            etc. etc. etc.
254
255            ARPHRD_IPDDP will also work, if author repairs it.
256            I did not it, because this driver does not work even
257            in old paradigm.
258            */
259
260     #if 1
261         /* So... these "amateur" devices are hopeless.
262            The only thing, that I can say now:
263            It is very sad that we need to keep ugly obsolete
264            code to make them happy.
265
266            They should be moved to more reasonable state, now
267            they use rebuild_header INSTEAD OF hard_start_xmit!!!
268            Besides that, they are sort of out of date

```

Figure 352: ARP Build Entry: linux/net/ipv4/arp.c

```

270         I wonder why people believe that they work.
271         */
272         switch (dev->type) {
273             default :
274                 break;
275             case ARPHRD_ROSE:
276 #if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
277                 case ARPHRD_AX25:
278 #if defined(CONFIG_NETROM) || defined(CONFIG_NETROM_MODULE)
279                 case ARPHRD_NETROM:
280 #endif
281                 neigh->ops = &arp_broken_ops;
282                 neigh->output = neigh->ops->output;
283                 return 0;
284 #endif
285             };
286 #endif
287         if (neigh->type == RTN_MULTICAST) {
288             neigh->nud_state = NUD_NOARP;
289             arp_mc_map(addr, neigh->ha, dev, 1);
290         } else if (dev->flags & (IFF_NOARP | IFF_LOOPBACK)) {
291             neigh->nud_state = NUD_NOARP;
292             memcpy(neigh->ha, dev->dev_addr, dev->addr_len);
293         } else if (neigh->type == RTN_BROADCAST || dev->flags &
294             IFF_POINTOPOINT) {
295             neigh->nud_state = NUD_NOARP;
296             memcpy(neigh->ha, dev->broadcast, dev->addr_len);
297         }
298         if (dev->hard_header_cache)
299             neigh->ops = &arp_hh_ops;
300         else
301             neigh->ops = &arp_generic_ops;
302         if (neigh->nud_state & NUD_VALID)
303             neigh->output = neigh->ops->connected_output;
304         else
305             neigh->output = neigh->ops->output;
306     }
307     return 0;
308 }
309 static void arp_error_report (struct neighbour *neigh, struct sk_buff *skb)
310 {
311     dst_link_failure (skb);
312     kfree_skb(skb);
313 }
314
315 static void arp_solicit (struct neighbour *neigh, struct sk_buff *skb)
316 {
317     u32 saddr;

```

Figure 353: ARP Build Entry: `linux/net/ipv4/arp.c`


```
319     struct net_device *dev = neigh->dev;
320     u32 target = *(u32*)neigh->primary_key;
321     int probes = atomic_read(&neigh->probes);
322
323     if (skb && inet_addr_type(skb->nh.iph->saddr) == RTN_LOCAL)
```

Figure 354: ARP Error Report: `linux/net/ipv4/arp.c`

```

325     else
326         saddr = inet_select_addr(dev, target, RT_SCOPE_LINK);
327
328     if ((probes -= neigh->parms->ucast_probes) < 0) {
329         if (!(neigh->nud_state & NUD_VALID))
330             printk(KERN_DEBUG "trying to ucast probe in NUD_INVALID\n");
331         dst_ha = neigh->ha;
332         read_lock_bh(&neigh->lock);
333     } else if ((probes -= neigh->parms->app_probes) < 0) {
334 #ifdef CONFIG_ARPD
335         neigh_app_ns(neigh);
336 #endif
337         return;
338     }
339
340     arp_send(ARPOP_REQUEST, ETH_P_ARP, target, dev, saddr,
341             dst_ha, dev->dev_addr, NULL);
342     if (dst_ha)
343         read_unlock_bh(&neigh->lock);
344 }
345
346 /* OBSOLETE FUNCTIONS */
347
348 /*
349  * Find an arp mapping in the cache. If not found, post a request.
350  *
351  * It is very UGLY routine: it DOES NOT use skb->dst->neighbour,
352  * even if it exists. It is supposed that skb->dev was mangled
353  * by a virtual device (eql, shaper). Nobody but broken devices
354  * is allowed to use this function, it is scheduled to be removed. --ANK

```

Figure 355: ARP Solicit (Who Has): linux/net/ipv4/arp.c

```

356 static int arp_set_predefined(int addr_hint, unsigned char * haddr, u32 paddr, struct
357 net_device * dev)
358 {
359     switch (addr_hint) {
360     case RTN_LOCAL:
361         printk(KERN_DEBUG "ARP: arp called for own IP address\n");
362         memcpy(haddr, dev->dev_addr, dev->addr_len);
363         return 1;
364     case RTN_MULTICAST:
365         arp_mc_map(paddr, haddr, dev, 1);
366         return 1;
367     case RTN_BROADCAST:
368         memcpy(haddr, dev->broadcast, dev->addr_len);
369         return 1;
370     }
371     return 0;
372 }
373
374
375 int arp_find(unsigned char *haddr, struct sk_buff *skb)
376 {
377     struct net_device *dev = skb->dev;
378     u32 paddr;
379     struct neighbour *n;
380
381     if (!skb->dst) {
382         printk(KERN_DEBUG "arp_find is called with dst==NULL\n");

```

Figure 356: ARP - even russians have Built in Obsolescence:
linux/net/ipv4/arp.c

```

384         return 1;
385     }
386
387     paddr = ((struct rtable*)skb->dst)->rt_gateway;
388
389     if (arp_set_predefined(inet_addr_type(paddr), haddr, paddr, dev))
390         return 0;
391
392     n = _neigh_lookup(&arp_tbl, &paddr, dev, 1);
393
394     if (n) {
395         n->used = jiffies;
396         if (n->nud_state & NUD_VALID || neigh_event_send(n, skb) == 0) {
397             read_lock_bh(&n->lock);
398             memcpy(haddr, n->ha, dev->addr_len);
399             read_unlock_bh(&n->lock);
400             neigh_release(n);
401             return 0;
402         }
403         neigh_release(n);
404     } else
405         kfree_skb(skb);
406     return 1;
407 }
408
409 /* END OF OBSOLETE FUNCTIONS */
410
411 int arp_bind_neighbour(struct dst_entry *dst)
412 {
413     struct net_device *dev = dst->dev;
414     struct neighbour *n = dst->neighbour;
415
416     if (dev == NULL)
417         return -EINVAL;
418     if (n == NULL) {
419         u32 nexthop = ((struct rtable*)dst)->rt_gateway;

```

Figure 357: ARP - even russians...: `linux/net/ipv4/arp.c`

```

421         nexthop = 0;
422         n = _neigh_lookup_errno(
423 #ifdef CONFIG_ATM_CLIP
424         dev->type == ARPHRD_ATM ? &clip_tbl :
425 #endif
426         &arp_tbl, &nexthop, dev);
427         if (IS_ERR(n))
428             return PTR_ERR(n);
429         dst->neighbour = n;
430     }
431     return 0;
432 }
433
434 /*
435  *   Interface to link layer: send routine and receive handler.
436  */
437
438 /*
439  *   Create and send an arp packet. If (dest_hw == NULL), we create a
440  *   broadcast
441  *   message.
442  */

```

Figure 358: ARP - Bind address to neighbour: linux/net/ipv4/arp.c

```

448     struct sk_buff *skb;
449     struct arphdr *arp;
450     unsigned char *arp_ptr;
451
452     /*
453      *   No arp on this interface.
454      */
455
456     if (dev->flags&IFF_NOARP)
457         return;
458
459     /*
460      *   Allocate a buffer
461      */
462
463     skb = alloc_skb(sizeof(struct arphdr)+ 2*(dev->addr_len+4)
464                   + dev->hard_header_len + 15, GFP_ATOMIC);
465     if (skb == NULL)
466         return;
467
468     skb_reserve(skb, (dev->hard_header_len+15)&~15);
469     skb->nh.raw = skb->data;
470     arp = (struct arphdr *) skb_put(skb, sizeof(struct arphdr) + 2*(dev->addr_len
471                                   +4));
472     skb->dev = dev;
473     skb->protocol = __constant_htons (ETH_P_ARP);
474     if (src_hw == NULL)
475         src_hw = dev->dev_addr;
476     if (dest_hw == NULL)
477         dest_hw = dev->broadcast;
478
479     /*
480      *   Fill the device header for the ARP frame
481      */
482     if (dev->hard_header &&
483         dev->hard_header(skb,dev,ptype,dest_hw,src_hw,skb->len) < 0)
484         goto out;
485
486     /*
487      *   Fill out the arp protocol part.

```

Figure 359: ARP - Send an ARP Message: `linux/net/ipv4/arp.c`

```

488     * The arp hardware type should match the device type, except for FDDI,
489     * which (according to RFC 1390) should always equal 1 (Ethernet).
490     */
491     /*
492     *     Exceptions everywhere. AX.25 uses the AX.25 PID value not the
493     *     DIX code for the protocol. Make these device structure fields.
494     */
495     switch (dev->type) {
496     default:
497         arp->ar_hrd = htons(dev->type);
498         arp->ar_pro = __constant_htons(ETH_P_IP);
499         break;
500
501     #if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
502     case ARPHRD_AX25:
503         arp->ar_hrd = __constant_htons(ARPHRD_AX25);
504         arp->ar_pro = __constant_htons(AX25_P_IP);
505         break;
506
507     #if defined(CONFIG_NETROM) || defined(CONFIG_NETROM_MODULE)
508     case ARPHRD_NETROM:
509         arp->ar_hrd = __constant_htons(ARPHRD_NETROM);
510         arp->ar_pro = __constant_htons(AX25_P_IP);
511         break;
512     #endif
513     #endif
514
515     #ifndef CONFIG_FDDI
516     case ARPHRD_FDDI:
517         arp->ar_hrd = __constant_htons(ARPHRD_ETHER);
518         arp->ar_pro = __constant_htons(ETH_P_IP);
519         break;
520     #endif
521     #ifdef CONFIG_TR
522     case ARPHRD_IEEE802_TR:
523         arp->ar_hrd = __constant_htons(ARPHRD_IEEE802);
524         arp->ar_pro = __constant_htons(ETH_P_IP);
525         break;
526     #endif
527     }
528
529     arp->ar_hln = dev->addr_len;
530     arp->ar_pln = 4;
531     arp->ar_op = htons(type);
532
533     arp_ptr=(unsigned char*)(arp+1);
534
535     memcpy(arp_ptr, src_hw, dev->addr_len);
536     arp_ptr+=dev->addr_len;
537     memcpy(arp_ptr, &src_ip,4);

```

Figure 360: ARP - Send an ARP Message: linux/net/ipv4/arp.c

```

539     if (target_hw != NULL)
540         memcpy(arp_ptr, target_hw, dev->addr_len);
541     else
542         memset(arp_ptr, 0, dev->addr_len);
543     arp_ptr+=dev->addr_len;
544     memcpy(arp_ptr, &dest_ip, 4);
545     skb->dev = dev;
546
547     dev_queue_xmit(skb);
548     return;
549
550 out:
551     kfree_skb(skb);
552 }
553
554 static void parp_redo(struct sk_buff *skb)
555 {
556     arp_rcv(skb, skb->dev, NULL);
557 }
558
559 /*
560 *   Receive an arp request by the device layer.
561 */

```

Figure 361: ARP - Send an ARP Message: `linux/net/ipv4/arp.c`


```
564 {
565     struct arphdr *arp = skb->nh.arph;
566     unsigned char *arp_ptr = (unsigned char *)(arp+1);
567     struct rtable *rt;
568     unsigned char *sha, *tha;
569     u32 sip, tip;
570     u16 dev_type = dev->type;
571     int addr_type;
```

Figure 362: ARP - Receive an ARP Message: `linux/net/ipv4/arp.c`

```

573     struct neighbour *n;
574
575     /*
576     *   The hardware length of the packet should match the hardware length
577     *   of the device. Similarly, the hardware types should match. The
578     *   device should be ARP-able. Also, if pln is not 4, then the lookup
579     *   is not from an IP number. We can't currently handle this, so toss
580     *   it.
581     */
582     if (in_dev == NULL ||
583         arp->ar_hln != dev->addr_len ||
584         dev->flags & IFF_NOARP ||
585         skb->pkt_type == PACKET_OTHERHOST ||
586         skb->pkt_type == PACKET_LOOPBACK ||
587         arp->ar_pln != 4)
588         goto out;
589
590     if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL)
591         goto out_of_mem;
592
593     if (skb_is_nonlinear(skb)) {
594         if (skb_linearize(skb, GFP_ATOMIC) != 0)
595             goto freeskb;
596         arp = skb->nh.arph;
597         arp_ptr = (unsigned char*)(arp+1);
598     }
599
600     switch (dev_type) {
601     default:

```

Figure 363: ARP - Receive an ARP Message: `linux/net/ipv4/arp.c`

```

603         goto out;
604         if (htons(dev_type) != arp->ar_hrd)
605             goto out;
606         break;
607 #ifdef CONFIG_NET_ETHERNET
608     case ARPHRD_ETHER:
609         /*
610          * ETHERNET devices will accept ARP hardware types of either
611          * 1 (Ethernet) or 6 (IEEE 802.2).
612          */
613         if (arp->ar_hrd != __constant_htons(ARPHRD_ETHER) &&
614             arp->ar_hrd != __constant_htons(ARPHRD_IEEE802))
615             goto out;
616         if (arp->ar_pro != __constant_htons(ETH_P_IP))
617             goto out;
618         break;
619 #endif
620 #ifdef CONFIG_TR
621     case ARPHRD_IEEE802_TR:
622         /*
623          * Token ring devices will accept ARP hardware types of either
624          * 1 (Ethernet) or 6 (IEEE 802.2).
625          */
626         if (arp->ar_hrd != __constant_htons(ARPHRD_ETHER) &&
627             arp->ar_hrd != __constant_htons(ARPHRD_IEEE802))
628             goto out;
629         if (arp->ar_pro != __constant_htons(ETH_P_IP))
630             goto out;
631         break;
632 #endif
633 #ifdef CONFIG_FDDI
634     case ARPHRD_FDDI:
635         /*

```

Figure 364: ARP - Receive an ARP Message: linux/net/ipv4/arp.c

```

636         * According to RFC 1390, FDDI devices should accept ARP hardware
637         * types
638         * of 1 (Ethernet). However, to be more robust, we'll accept hardware
639         * types of either 1 (Ethernet) or 6 (IEEE 802.2).
640         */
641         if (arp->ar_hrd != __constant_htons(ARPHRD_ETHER) &&
642             arp->ar_hrd != __constant_htons(ARPHRD_IEEE802))
643             goto out;
644         if (arp->ar_pro != __constant_htons(ETH_P_IP))
645             goto out;
646         break;
647 #endif
648 #ifdef CONFIG_NET_FC
649     case ARPHRD_IEEE802:
650         /*
651          * According to RFC 2625, Fibre Channel devices (which are IEEE
652          * 802 devices) should accept ARP hardware types of 6 (IEEE 802)
653          * and 1 (Ethernet).
654          */
655         if (arp->ar_hrd != __constant_htons(ARPHRD_ETHER) &&
656             arp->ar_hrd != __constant_htons(ARPHRD_IEEE802))
657             goto out;
658         if (arp->ar_pro != __constant_htons(ETH_P_IP))
659             goto out;
660         break;
661 #endif
662 #if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
663     case ARPHRD_AX25:
664         if (arp->ar_pro != __constant_htons(AX25_P_IP))
665             goto out;
666         if (arp->ar_hrd != __constant_htons(ARPHRD_AX25))
667             goto out;

```

Figure 365: ARP - Receive an ARP Message: `linux/net/ipv4/arp.c`

```

668 #if defined(CONFIG_NETROM) || defined(CONFIG_NETROM_MODULE)
669     case ARPHRD_NETROM:
670         if (arp->ar_pro != __constant_htons(AX25_P_IP))
671             goto out;
672         if (arp->ar_hrd != __constant_htons(ARPHRD_NETROM))
673             goto out;
674         break;
675 #endif
676 #endif
677     }
678
679     /* Understand only these message types */
680
681     if (arp->ar_op != __constant_htons(ARPOP_REPLY) &&
682         arp->ar_op != __constant_htons(ARPOP_REQUEST))
683         goto out;
684
685     /*
686     *   Extract fields
687     */
688     sha=arp_ptr;
689     arp_ptr += dev->addr_len;
690     memcpy(&sip, arp_ptr, 4);
691     arp_ptr += 4;
692     tha=arp_ptr;
693     arp_ptr += dev->addr_len;
694     memcpy(&tip, arp_ptr, 4);
695
696     /*
697     *   Check for bad requests for 127.x.x.x and requests for multicast
698     *   addresses. If this is one such, delete it.
699     */
700     if (LOOPBACK(tip) || MULTICAST(tip))
701         goto out;
702
703     /*
704     *   Special case: We must set Frame Relay source Q.922 address
705     */
706     if (dev_type == ARPHRD_DLCI)
707         sha = dev->broadcast;

```

Figure 366: ARP - Receive an ARP Message: linux/net/ipv4/arp.c

```

708  /*
709  * Process entry. The idea here is we want to send a reply if it is a
710  * request for us or if it is a request for someone else that we hold
711  * a proxy for. We want to add an entry to our cache if it is a reply
712  * to us or if it is a request for our address.
713  * (The assumption for this last is that if someone is requesting our
714  * address, they are probably intending to talk to us, so it saves time
715  * if we cache their address. Their address is also probably not in
716  * our cache, since ours is not in their cache.)
717  *
718  * Putting this another way, we only care about replies if they are to
719  * us, in which case we add them to the cache. For requests, we care
720  * about those for us and those for our proxies. We reply to both,
721  * and in the case of requests for us we add the requester to the arp
722  * cache.
723  */
724
725     /* Special case: IPv4 duplicate address detection packet (RFC2131) */
726     if (sip == 0) {
727         if (arp->ar_op == htons(ARPOP_REQUEST) &&
728             inet_addr_type(tip) == RTN_LOCAL)
729             arp_send(ARPOP_REPLY,ETH_P_ARP,tip,dev,tip,sha,dev->
730                     dev_addr,dev->dev_addr);
731         goto out;
732     }
733     if (arp->ar_op == htons(ARPOP_REQUEST) &&
734         ip_route_input(skb, tip, sip, 0, dev) == 0) {
735
736         rt = (struct rtable*)skb->dst;
737         addr_type = rt->rt_type;
738
739         if (addr_type == RTN_LOCAL) {
740             n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
741             if (n) {
742                 arp_send(ARPOP_REPLY,ETH_P_ARP,sip,dev,tip,sha,dev
743                         ->dev_addr,sha);
744                 neigh_release(n);
745             }
746             goto out;
747         } else if (IN_DEV_FORWARD(in_dev)) {
748             if ((rt->rt_flags&RTCF_DNAT) ||
749                 (addr_type == RTN_UNICAST && rt->u.dst.dev != dev &&
750                  (IN_DEV_PROXY_ARP(in_dev) || pneigh_lookup(&arp_tbl, &
751                    tip, dev, 0)))) {

```

Figure 367: ARP - Receive an ARP Message: linux/net/ipv4/arp.c

```

751         if (n)
752             neigh_release(n);
753
754         if (skb->stamp.tv_sec == 0 ||
755             skb->pkt_type == PACKET_HOST ||
756             in_dev->arp_parms->proxy_delay == 0) {
757             arp_send(ARPOP_REPLY,ETH_P_ARP,sip,dev,tip
758                     ,sha,dev->dev_addr,sha);
759         } else {
760             pneigh_enqueue(&arp_tbl, in_dev->arp_parms, skb
761                             );
762             in_dev_put(in_dev);
763             return 0;
764         }
765     }
766 }
767
768 /* Update our ARP tables */
769
770 n = _neigh_lookup(&arp_tbl, &sisip, dev, 0);
771
772 #ifdef CONFIG_IP_ACCEPT_UNSOLICITED_ARP
773     /* Unsolicited ARP is not accepted by default.
774        It is possible, that this option should be enabled for some
775        devices (strip is candidate)
776        */
777     if (n == NULL &&
778         arp->ar_op == _constant_htons(ARPOP_REPLY) &&
779         inet_addr_type(sip) == RTN_UNICAST)
780         n = _neigh_lookup(&arp_tbl, &sisip, dev, -1);
781 #endif
782
783     if (n) {
784         int state = NUD_REACHABLE;
785         int override = 0;
786
787         /* If several different ARP replies follows back-to-back,
788            use the FIRST one. It is possible, if several proxy
789            agents are active. Taking the first reply prevents
790            arp trashing and chooses the fastest router.
791            */
792         if ( jiffies - n->updated >= n->parms->locktime)
793             override = 1;

```

Figure 368: ARP - Receive an ARP Message: linux/net/ipv4/arp.c

```

798         if (arp->ar_op != _constant_htons(ARPOP_REPLY) ||
799             skb->pkt_type != PACKET_HOST)
800             state = NUD_STALE;
801         neigh_update(n, sha, state, override, 1);
802         neigh_release(n);
803     }
804
805 out:
806     if (in_dev)
807         in_dev_put(in_dev);
808 freeskb:
809     kfree_skb(skb);
810 out_of_mem:
811     return 0;
812 }
813
814
815
816 /*
817  *   User level interface (ioctl, /proc)
818  */
819
820 /*
821  *   Set (create) an ARP cache entry.
822  */
823
824 int arp_req_set(struct arpreq *r, struct net_device * dev)
825 {
826     u32 ip = ((struct sockaddr_in *) &r->arp_pa)->sin_addr.s_addr;
827     struct neighbour *neigh;
828     int err;
829
830     if (r->arp_flags & ATF_PUBL) {
831         u32 mask = ((struct sockaddr_in *) &r->arp_netmask)->sin_addr.s_addr;
832         if (mask && mask != 0xFFFFFFFF)
833             return -EINVAL;
834         if (!dev && (r->arp_flags & ATF_COM)) {
835             dev = dev_getbyhwaddr(r->arp_ha.sa_family, r->arp_ha.sa_data);

```

Figure 369: ARP - User Interface (proc, ioctl) - Create a cache entry:
linux/net/ipv4/arp.c


```

837         return -ENODEV;
838     }
839     if (mask) {
840         if (pneigh_lookup(&arp_tbl, &ip, dev, 1) == NULL)
841             return -ENOBUFS;
842         return 0;
843     }
844     if (dev == NULL) {
845         ipv4_devconf.proxy_arp = 1;
846         return 0;
847     }
848     if (_in_dev_get(dev)) {
849         _in_dev_get(dev)->cnf.proxy_arp = 1;
850         return 0;
851     }
852     return -ENXIO;
853 }
854
855 if (r->arp_flags & ATF_PERM)
856     r->arp_flags |= ATF_COM;
857 if (dev == NULL) {
858     struct rtable * rt;
859     if ((err = ip_route_output(&rt, ip, 0, RTO_ONLINK, 0)) != 0)
860         return err;
861     dev = rt->u.dst.dev;
862     ip_rt_put(rt);

```

Figure 370: ARP - User Interface (proc, ioctl) - Create a cache entry:
linux/net/ipv4/arp.c

```
864         return -EINVAL;
865     }
866     if (r->arp_ha.sa_family != dev->type)
867         return -EINVAL;
868
869     neigh = _neigh_lookup_errno(&arp_tbl, &ip, dev);
870     err = PTR_ERR(neigh);
871     if (!IS_ERR(neigh)) {
872         unsigned state = NUD_STALE;
```

Figure 371: ARP - User Interface - change ARP entry flags:
linux/net/ipv4/arp.c

```

878     }
879     return err;
880 }
881
882 static unsigned arp_state_to_flags (struct neighbour *neigh)
883 {
884     unsigned flags = 0;
885     if (neigh->nud_state&NUD_PERMANENT)
886         flags = ATF_PERM|ATF_COM;
887     else if (neigh->nud_state&NUD_VALID)
888         flags = ATF_COM;
889     return flags;
890 }
891
892 /*
893  *   Get an ARP cache entry.
894  */
895
896 static int arp_req_get(struct arpreq *r, struct net_device *dev)

```

Figure 372: ARP - User Interface - Get an ARP entry: linux/net/ipv4/arp.c

```

898     u32 ip = ((struct sockaddr_in *) &r->arp_pa)->sin_addr.s_addr;
899     struct neighbour *neigh;
900     int err = -ENXIO;
901
902     neigh = neigh_lookup(&arp_tbl, &ip, dev);
903     if (neigh) {
904         read_lock_bh(&neigh->lock);
905         memcpy(r->arp_ha.sa_data, neigh->ha, dev->addr_len);
906         r->arp_flags = arp_state_to_flags(neigh);
907         read_unlock_bh(&neigh->lock);
908         r->arp_ha.sa_family = dev->type;
909         strncpy(r->arp_dev, dev->name, sizeof(r->arp_dev));
910         neigh_release(neigh);
911         err = 0;
912     }
913     return err;
914 }
915
916 int arp_req_delete(struct arpreq *r, struct net_device * dev)
917 {
918     int err;
919     u32 ip = ((struct sockaddr_in *) &r->arp_pa)->sin_addr.s_addr;
920     struct neighbour *neigh;

```

Figure 373: ARP - User Interface - Delete an ARP entry:
linux/net/ipv4/arp.c

```

922     if (r->arp_flags & ATF_PUBL) {
923         u32 mask = ((struct sockaddr_in *) &r->arp_netmask)->sin_addr.s_addr;
924         if (mask == 0xFFFFFFFF)
925             return p neigh_delete(&arp_tbl, &ip, dev);
926         if (mask == 0) {
927             if (dev == NULL) {
928                 ipv4_devconf.proxy_arp = 0;
929                 return 0;
930             }
931             if (_in_dev_get(dev)) {
932                 _in_dev_get(dev)->cnf.proxy_arp = 0;
933                 return 0;
934             }
935             return -ENXIO;
936         }
937         return -EINVAL;
938     }

```

Figure 374: ARP - User Interface - Delete an ARP entry:
linux/net/ipv4/arp.c

```

945         ip_rt_put(rt);
946         if (!dev)
947             return -EINVAL;
948     }
949     err = -ENXIO;
950     neigh = neigh_lookup(&arp_tbl, &ip, dev);
951     if (neigh) {
952         if (neigh->nud_state & ~NUD_NOARP)
953             err = neigh_update(neigh, NULL, NUD_FAILED, 1, 0);
954         neigh_release(neigh);
955     }
956     return err;
957 }
958
959 /*
960  *   Handle an ARP layer I/O control request.
961  */
962
963 int arp_ioctl(unsigned int cmd, void *arg)
964 {
965     int err;
966     struct arpreq r;
967     struct net_device * dev = NULL;
968
969     switch(cmd) {
970         case SIOCDDARP:
971         case SIOCSARP:
972             if (!capable(CAP_NET_ADMIN))

```

Figure 375: ARP - User Interface - ioctl entry point: `linux/net/ipv4/arp.c`

```

974         case SIOCGARP:
975             err = copy_from_user(&r, arg, sizeof(struct arpreq));
976             if (err)
977                 return -EFAULT;
978             break;
979         default:
980             return -EINVAL;
981     }
982
983     if (r.arp_pa.sa_family != AF_INET)
984         return -EPFNOSUPPORT;
985
986     if (!(r.arp_flags & ATF_PUBL) &&
987         (r.arp_flags & (ATF_NETMASK|ATF_DONTPUB)))
988         return -EINVAL;
989     if (!(r.arp_flags & ATF_NETMASK))
990         ((struct sockaddr_in *)&r.arp_netmask)->sin_addr.s_addr=
991         __constant_htonl(0xFFFFFFFFUL);
992
993     rtnl_lock();
994     if (r.arp_dev[0]) {
995         err = -ENODEV;
996         if ((dev = __dev_get_by_name(r.arp_dev)) == NULL)
997             goto out;
998
999         /* Mmmm... It is wrong... ARPHRD_NETROM==0 */
1000        if (!r.arp_ha.sa_family)
1001            r.arp_ha.sa_family = dev->type;
1002        err = -EINVAL;
1003        if ((r.arp_flags & ATF_COM) && r.arp_ha.sa_family != dev->type)
1004            goto out;
1005    } else if (cmd == SIOCGARP) {
1006        err = -ENODEV;
1007        goto out;
1008    }

```

Figure 376: ARP - User Interface - ioctl entry point: `linux/net/ipv4/arp.c`

```

1009     switch(cmd) {
1010     case SIOCDDARP:
1011         err = arp_req_delete(&r, dev);
1012         break;
1013     case SIOCSARP:
1014         err = arp_req_set(&r, dev);
1015         break;
1016     case SIOCGARP:
1017         err = arp_req_get(&r, dev);
1018         if (!err && copy_to_user(arg, &r, sizeof(r)))
1019             err = -EFAULT;
1020         break;
1021     }
1022 out:
1023     rtnl_unlock();
1024     return err;
1025 }
1026
1027 /*
1028  *   Write the contents of the ARP cache to a PROCfs file.
1029  */
1030 #ifndef CONFIG_PROC_FS
1031 static int arp_get_info(char *buffer, char **start, off_t offset, int length) { return
1032     0; }
1033 #else
1034 #if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
1035 static char *ax2asc2(ax25_address *a, char *buf);
1036 #endif
1037 #define HBUFFERLEN 30
1038 static int arp_get_info(char *buffer, char **start, off_t offset, int length)
1039 {
1040     int len=0;
1041     off_t pos=0;
1042     int size;
1043     char hbuffer[HBUFFERLEN];
1044     int i,j,k;
1045     const char hexbuf[] = "0123456789ABCDEF";

```

Figure 377: ARP - User Interface - procfs entry point - write table:
linux/net/ipv4/arp.c


```

1047     size = sprintf(buffer, "IP_address_____HW_type_____Flags_____HW_address
1048         _____Mask_____Device\n");
1049
1049     pos+=size;
1050     len+=size;
1051
1052     for(i=0; i<=NEIGH_HASHMASK; i++) {
1053         struct neighbour *n;
1054         read_lock_bh(&arp_tbl.lock);
1055         for (n=arp_tbl.hash_buckets[i]; n; n=n->next) {
1056             struct net_device *dev = n->dev;
1057             int hatype = dev->type;
1058
1059             /* Do not confuse users "arp -a" with magic entries */
1060             if (!(n->nud_state&~NUD_NOARP))
1061                 continue;
1062
1063             read_lock(&n->lock);
1064
1065             /*
1066              * Convert hardware address to XX:XX:XX:XX ... form.
1067              */
1068             #if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
1069                 if (hatype == ARPHRD_AX25 || hatype ==
1070                     ARPHRD_NETROM)
1071                     ax2asc2((ax25_address *)n->ha, hbuffer);
1072             #endif
1073                 for (k=0, j=0; k<HBUFFERLEN-3 && j<dev->addr_len; j++) {
1074                     hbuffer[k++] = hexbuf[(n->ha[j]>>4)&15];
1075                     hbuffer[k++] = hexbuf[n->ha[j]&15];
1076                     hbuffer[k++] = ':';
1077                 }
1078                 hbuffer[--k] = 0;
1079
1080             #if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
1081                 }
1082             #endif
1083
1084             {
1085                 char tbuf[16];
1086                 sprintf(tbuf, "%u.%u.%u.%u", NIPQUAD(*(u32*)n->
1087                     primary_key));
1088                 size = sprintf(buffer+len, "%-16s, 0x%-10x, 0x%-10x, %s\n",
1089                     "_____*_____*_____*_____*",
1090                     tbuf,
1091                     hatype,
1092                     arp_state_to_flags(n),

```

Figure 378: ARP - User Interface - procs entry point - write table:
linux/net/ipv4/arp.c

```

1093         dev->name);
1094     }
1095
1096     read_unlock(&n->lock);
1097
1098     len += size;
1099     pos += size;
1100
1101     if (pos <= offset)
1102         len=0;
1103     if (pos >= offset+length) {
1104         read_unlock_bh(&arp_tbl.lock);
1105         goto done;
1106     }
1107 }
1108 read_unlock_bh(&arp_tbl.lock);
1109 }
1110
1111 for (i=0; i<=PNEIGH_HASHMASK; i++) {
1112     struct pneighbor *n;
1113     for (n=arp_tbl.phash_buckets[i]; n; n=n->next) {
1114         struct net_device *dev = n->dev;
1115         int hatype = dev ? dev->type : 0;
1116
1117         {
1118             char tbuf[16];
1119             sprintf(tbuf, "%u.%u.%u.%u", NIPQUAD(*(u32*)n->
1120                 key));
1121             size = sprintf(buffer+len, "%-16s_0x%-10x0x%-10x%s"
1122                 "_____*_____%s\n",
1123                 tbuf,
1124                 hatype,
1125                 ATF_PUBL|ATF_PERM,
1126                 "00:00:00:00:00:00",
1127                 dev ? dev->name : "*");
1128         }
1129         len += size;
1130         pos += size;

```

Figure 379: ARP - User Interface - procs entry point - write table:
linux/net/ipv4/arp.c

```
1133         len=0;
1134         if (pos >= offset+length)
1135             goto done;
1136     }
1137 }
1138
1139 done:
```

Figure 380: ARP - Interface Down: `linux/net/ipv4/arp.c`

```

1147     return len;
1148 }
1149 #endif
1150
1151 /* Note, that it is not on notifier chain.
1152    It is necessary, that this routine was called after route cache will be
1153    flushed.
1154 */
1155 void arp_ifdown(struct net_device *dev)
1156 {
1157     neigh_ifdown(&arp_tbl, dev);
1158 }
1159
1160
1161 /*
1162  *   Called once on startup.
1163  */
1164
1165 static struct packet_type arp_packet_type =
1166 {
1167     __constant_htons(ETH_P_ARP),

```

Figure 381: ARP - Initialise: linux/net/ipv4/arp.c

```

1170     (void*)1,
1171     NULL
1172 };
1173
1174 void _init arp_init (void)
1175 {
1176     neigh_table_init (&arp_tbl);
1177
1178     dev_add_pack(&arp_packet_type);
1179
1180     proc_net_create ("arp", 0, arp_get_info);
1181
1182 #ifdef CONFIG_SYSCTL
1183     neigh_sysctl_register (NULL, &arp_tbl.parms, NET_IPV4, NET_IPV4_NEIGH, "
1184         ipv4");
1185 #endif
1186 }
1187
1188 #ifdef CONFIG_PROC_FS
1189 #if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
1190
1191 /*
1192  *   ax25 -> ASCII conversion
1193  */
1194 char *ax2asc2(ax25_address *a, char *buf)
1195 {
1196     char c, *s;
1197     int n;
1198
1199     for (n = 0, s = buf; n < 6; n++) {
1200         c = (a->ax25_call[n] >> 1) & 0x7F;
1201
1202         if (c != '\0') *s++ = c;
1203     }
1204
1205     *s++ = '-';

```

Figure 382: ARP - Amateur Radio X.25 ARP String Address Conversion!: `linux/net/ipv4/arp.c`

```
61  */
62
63  #include <linux/config.h>
64  #include <linux/types.h>
65  #include <linux/sched.h>
66  #include <linux/kernel.h>
67  #include <linux/fcntl.h>
68  #include <linux/socket.h>
69  #include <linux/in.h>
70  #include <linux/inet.h>
71  #include <linux/netdevice.h>
72  #include <linux/string.h>
73  #include <linux/netfilter_ipv4.h>
74  #include <net/snmp.h>
75  #include <net/ip.h>
76  #include <net/route.h>
77  #include <net/protocol.h>
78  #include <net/icmp.h>
79  #include <net/tcp.h>
80  #include <net/udp.h>
81  #include <net/raw.h>
82  #include <linux/skbuff.h>
83  #include <net/sock.h>
```

Figure 383: ICMP, Hosts Requirement Conformance:-):
linux/net/ipv4/icmp.c

```

84 #include <linux/errno.h>
85 #include <linux/timer.h>
86 #include <linux/init.h>
87 #include <asm/system.h>
88 #include <asm/uaccess.h>
89 #include <net/checksum.h>
90
91 #define min(a,b)      ((a)<(b)?(a):(b))
92
93 /*
94  *   Build xmit assembly blocks
95  */
96
97 struct icmp_bxm
98 {
99     struct sk_buff *skb;
100     int offset ;
101     int data_len;
102
103     unsigned int csum;
104     struct {
105         struct icmphdr icmph;
106         _u32          times[3];
107     } data;
108     int head_len;
109     struct ip_options replyopts;
110     unsigned char  optbuf[40];
111 };
112
113 /*
114  *   Statistics
115  */
116
117 struct icmp_mib icmp_statistics[NR_CPUS*2];
118
119 /* An array of errno for error messages from dest unreachable. */
120 /* RFC 1122: 3.2.2.1 States that NET_UNREACH, HOS_UNREACH and SR_FAIELD MUST be
121    considered 'transient errs'. */
122
123 struct icmp_err icmp_err_convert[] = {

```

Figure 384: ICMP, Hosts Requirement Conformance:-);
linux/net/ipv4/icmp.c

```

125 { ENOPROTOOPT, 1 }, /* ICMP_PROT_UNREACH */
126 { ECONNREFUSED, 1 }, /* ICMP_PORT_UNREACH */
127 { EMSGSIZE, 0 }, /* ICMP_FRAG_NEEDED */
128 { EOPNOTSUPP, 0 }, /* ICMP_SR_FAILED */
129 { ENETUNREACH, 1 }, /* ICMP_NET_UNKNOWN */
130 { EHOSTDOWN, 1 }, /* ICMP_HOST_UNKNOWN */
131 { ENONET, 1 }, /* ICMP_HOST_ISOLATED */
132 { ENETUNREACH, 1 }, /* ICMP_NET_ANO */
133 { EHOSTUNREACH, 1 }, /* ICMP_HOST_ANO */
134 { ENETUNREACH, 0 }, /* ICMP_NET_UNR_TOS */
135 { EHOSTUNREACH, 0 }, /* ICMP_HOST_UNR_TOS */
136 { EHOSTUNREACH, 1 }, /* ICMP_PKT_FILTERED */
137 { EHOSTUNREACH, 1 }, /* ICMP_PREC_VIOLATION */
138 { EHOSTUNREACH, 1 } /* ICMP_PREC_CUTOFF */
139 };
140
141 /* Control parameters for ECHO relies. */
142 int sysctl_icmp_echo_ignore_all = 0;
143 int sysctl_icmp_echo_ignore_broadcasts = 0;
144
145 /* Control parameter - ignore bogus broadcast responses? */
146 int sysctl_icmp_ignore_bogus_error_responses = 0;
147
148 /*
149 * ICMP control array. This specifies what to do with each ICMP.
150 */
151
152 struct icmp_control
153 {
154     unsigned long *output; /* Address to increment on output */
155     unsigned long *input; /* Address to increment on input */
156     void (*handler)(struct sk_buff *skb);
157     short error; /* This ICMP is classed as an error message */
158     int *timeout; /* Rate limit */
159 };

```

Figure 385: ICMP, Router Requirement Conformance:-):
linux/net/ipv4/icmp.c


```

160
161 static struct icmp_control icmp_pointers[NR_ICMP_TYPES+1];
162
163 /*
164  *   The ICMP socket. This is the most convenient way to flow control
165  *   our ICMP output as well as maintain a clean interface throughout
166  *   all layers. All Socketless IP sends will soon be gone.
167  */
168
169 struct inode icmp_inode;
170 struct socket *icmp_socket=&icmp_inode.u.socket_i;
171
172 /* ICMPv4 socket is only a bit non-reentrantable (unlike ICMPv6,
173    which is strongly non-reentrantable). A bit later it will be made
174    reentrantable and the lock may be removed then.
175 */
176
177 static int icmp_xmit_holder = -1;

```

Figure 386: ICMP, Router Requirement Conformance:-):
linux/net/ipv4/icmp.c

```

178 static int icmp_xmit_lock_bh(void)
179 {
180     if (!spin_trylock(&icmp_socket->sk->lock.slock)) {
181         if (icmp_xmit_holder == smp_processor_id())
182             return -EAGAIN;
183         spin_lock(&icmp_socket->sk->lock.slock);
184     }
185     icmp_xmit_holder = smp_processor_id();
186     return 0;
187 }
188
189
190 static __inline__ int icmp_xmit_lock(void)
191 {
192     int ret;
193     local_bh_disable();
194     ret = icmp_xmit_lock_bh();
195     if (ret)
196         local_bh_enable();
197     return ret;
198 }
199
200 static void icmp_xmit_unlock_bh(void)
201 {
202     icmp_xmit_holder = -1;
203     spin_unlock(&icmp_socket->sk->lock.slock);
204 }
205
206 static __inline__ void icmp_xmit_unlock(void)
207 {
208     icmp_xmit_unlock_bh();
209     local_bh_enable();
210 }
211
212
213 /*
214  *   Send an ICMP frame.
215  */
216
217 /*
218  *   Check transmit rate limitation for given message.
219  *   The rate information is held in the destination cache now.
220  *   This function is generic and could be used for other purposes
221  *   too. It uses a Token bucket filter as suggested by Alexey Kuznetsov.
222  */

```

Figure 387: ICMP, Router Requirement Conformance:-):
linux/net/ipv4/icmp.c

```

223 *   Note that the same dst_entry fields are modified by functions in
224 *   route.c too, but these work for packet destinations while xrlim_allow
225 *   works for icmp destinations. This means the rate limiting information
226 *   for one "ip_object" is shared.
227 *
228 *   Note that the same dst_entry fields are modified by functions in
229 *   route.c too, but these work for packet destinations while xrlim_allow
230 *   works for icmp destinations. This means the rate limiting information
231 *   for one "ip_object" is shared – and these ICMPs are twice limited:
232 *   by source and by destination.
233 *
234 *   RFC 1812: 4.3.2.8 SHOULD be able to limit error message rate
235 *               SHOULD allow setting of rate limits
236 *
237 *   Shared between ICMPv4 and ICMPv6.
238 */
239 #define XRLIM_BURST_FACTOR 6
240 int xrlim_allow(struct dst_entry *dst, int timeout)
241 {
242     unsigned long now;
243
244     now = jiffies;
245     dst->rate_tokens += now - dst->rate_last;
246     dst->rate_last = now;
247     if (dst->rate_tokens > XRLIM_BURST_FACTOR*timeout)
248         dst->rate_tokens = XRLIM_BURST_FACTOR*timeout;
249     if (dst->rate_tokens >= timeout) {
250         dst->rate_tokens -= timeout;
251         return 1;

```

Figure 388: ICMP, Router Requirement Conformance:-):
linux/net/ipv4/icmp.c

```

288 }
289
290 /*
291  *   Checksum each fragment, and on the first include the headers and final
292   *   checksum.
293  */
294 static int icmp_glue_bits(const void *p, char *to, unsigned int offset, unsigned int
295   fraglen)
296 {
297     struct icmp_bxm *icmp_param = (struct icmp_bxm *)p;
298     struct icmp_hdr *icmph;
299     unsigned int csum;
300
301     if (offset) {
302         icmp_param->csum=skb_copy_and_csum_bits(icmp_param->skb,
303         icmp_param->offset+(offset-
304         icmp_param->head_len),
305         to, fraglen, icmp_param->csum);
306     }
307     return 0;
308 }
309
310 /*
311  *   First fragment includes header. Note that we've done
312  *   the other fragments first, so that we get the checksum
313  *   for the whole packet here.
314  */
315 csum = csum_partial_nocheck((void *)&icmp_param->data,
316   to, icmp_param->head_len,
317   icmp_param->csum);
318 csum=skb_copy_and_csum_bits(icmp_param->skb,
319   icmp_param->offset,
320   to+icmp_param->head_len,

```

Figure 389: ICMP Global Data: linux/net/ipv4/icmp.c

```

319         csum);
320     icmp_h=(struct icmp_hdr *)to;
321     icmp_h->checksum = csum_fold(csum);
322     return 0;
323 }
324
325 /*
326  *   Driving logic for building and sending ICMP messages.
327  */
328
329 static void icmp_reply(struct icmp_bxm *icmp_param, struct sk_buff *skb)
330 {
331     struct sock *sk=icmp_socket->sk;
332     struct ipcm_cookie ipc;
333     struct rtable *rt = (struct rtable*)skb->dst;
334     u32 daddr;
335
336     if (ip_options_echo(&icmp_param->replyopts, skb))
337         return;
338
339     if (icmp_xmit_lock_bh())
340         return;
341
342     icmp_param->data.icmp_h.checksum=0;
343     icmp_param->csum=0;
344     icmp_out_count(icmp_param->data.icmp_h.type);
345
346     sk->protinfo.af_inet.tos = skb->nh.iph->tos;
347     daddr = ipc.addr = rt->rt_src;
348     ipc.opt = NULL;

```

Figure 390: ICMP Global Data: `linux/net/ipv4/icmp.c`

```

350         ipc.opt = &icmp_param->replyopts;
351         if (ipc.opt->srr)
352             daddr = icmp_param->replyopts.faddr;
353     }
354     if (ip_route_output(&rt, daddr, rt->rt_spec_dst, RT_TOS(skb->nh.iph->tos), 0)
355         )
356         goto out;
357     if (icmpv4_xrlim_allow(rt, icmp_param->data.icmph.type,
358         icmp_param->data.icmph.code)) {
359         ip_build_xmit(sk, icmp_glue_bits, icmp_param,
360             icmp_param->data_len+icmp_param->head_len,
361             &ipc, rt, MSG_DONTWAIT);
362     }
363     ip_rt_put(rt);
364 out:
365     icmp_xmit_unlock_bh();
366 }
367
368 /*
369 *   Send an ICMP message in response to a situation
370 *
371 *   RFC 1122: 3.2.2 MUST send at least the IP header and 8 bytes of header.
372 *   MAY send more (we do).
373 *
374 *   MUST NOT change this header information.
375 *   MUST NOT reply to a multicast/broadcast IP address.
376 *   MUST NOT reply to a multicast/broadcast MAC address.
377 *   MUST reply to only the first fragment.
378 */
379 void icmp_send(struct sk_buff *skb_in, int type, int code, u32 info)
380 {
381     struct iphdr *iph;
382     int room;

```

Figure 391: ICMP - transmit locks: linux/net/ipv4/icmp.c

```

388     if (!rt)
389         return;
390
391     /*
392     *     Find the original header. It is expected to be valid, of course.
393     *     Check this, icmp_send is called from the most obscure devices
394     *     sometimes.
395     */
396     iph = skb_in->nh.iph;
397
398     if ((u8*)iph < skb_in->head || (u8*)(iph+1) > skb_in->tail)
399         return;
400
401     /*
402     *     No replies to physical multicast/broadcast
403     */
404     if (skb_in->pkt_type!=PACKET_HOST)
405         return;
406
407     /*
408     *     Now check at the protocol level
409     */
410     if (rt->rt_flags&(RTCF_BROADCAST|RTCF_MULTICAST))
411         return;
412
413     /*
414     *     Only reply to fragment 0. We byte re-order the constant
415     *     mask for efficiency.
416     */
417     if (iph->frag_off&htons(IP_OFFSET))
418         return;
419
420     /*
421     *     If we send an ICMP error to an ICMP error a mess would result..
422     */
423     if (icmp_pointers[type].error) {
424         /*
425         *     We are an error, check if we are replying to an ICMP error

```

Figure 392: ICMP transmit rate limiter - token bucket:
linux/net/ipv4/icmp.c

```

427         if (iph->protocol==IPPROTO_ICMP) {
428             u8 inner_type;
429
430             if (skb_copy_bits(skb_in,
431                             skb_in->nh.raw + (iph->ihl<<2)
432                             + offsetof(struct icmphdr, type)
433                             - skb_in->data,
434                             &inner_type, 1))
435                 return;
436
437             /*
438              *   Assume any unknown ICMP type is an error. This isn't
439              *   specified by the RFC, but think about it..
440              */
441             if (inner_type>NR_ICMP_TYPES || icmp_pointers[inner_type].
442                 error)
443                 return;
444         }
445     }
446     if (icmp_xmit_lock())
447         return;

```

Figure 393: ICMP transmit rate limit computation: `linux/net/ipv4/icmp.c`


```
449     /*
450      *   Construct source address and options.
451      */
452
453 #ifdef CONFIG_IP_ROUTE_NAT
454     /*
455      *   Restore original addresses if packet has been translated.
456      */
457     if (rt->rt_flags&RTCF_NAT && IPCB(skb_in)->flags&IPSKB_TRANSLATED)
458         {
459             iph->daddr = rt->key.dst;
460             iph->saddr = rt->key.src;
```

Figure 394: ICMP/SNMP statistics counters: `linux/net/ipv4/icmp.c`

```

451     */
452
453 #ifdef CONFIG_IP_ROUTE_NAT
454     /*
455      *   Restore original addresses if packet has been translated.
456      */
457     if (rt->rt_flags&RTCF_NAT && IPCB(skb_in)->flags&IPSKB_TRANSLATED)
458         {
459             iph->daddr = rt->key.dst;
460             iph->saddr = rt->key.src;
461         }
462 #endif
463
464     saddr = iph->daddr;
465     if (!(rt->rt_flags & RTCF_LOCAL))
466         saddr = 0;
467
468     tos = icmp_pointers[type].error ?
469         ((iph->tos & IPTOS_TOS_MASK) |
470          IPTOS_PREC_INTERNETCONTROL) :
471         iph->tos;
472
473     if (ip_route_output(&rt, iph->saddr, saddr, RT_TOS(tos), 0))
474         goto out;
475
476     if (ip_options_echo(&icmp_param.replyopts, skb_in))
477         goto ende;
478
479     /*
480      *   Prepare data for ICMP header.
481      */
482
483     icmp_param.data.icmph.type=type;
484     icmp_param.data.icmph.code=code;
485     icmp_param.data.icmph.un.gateway = info;
486     icmp_param.data.icmph.checksum=0;
487     icmp_param.csum=0;
488     icmp_param.skb=skb_in;
489     icmp_param.offset=skb_in->nh.raw - skb_in->data;
490     icmp_out_count(icmp_param.data.icmph.type);
491     icmp_socket->sk->prot_info.af_inet.tos = tos;
492     ipc.addr = iph->saddr;

```

Figure 395: ICMP Checksum: linux/net/ipv4/icmp.c

```

497     }
498
499     if (!icmpv4_xrlim_allow(rt, type, code))
500         goto ende;
501
502     /* RFC says return as much as we can without exceeding 576 bytes. */
503
504     room = rt->u.dst.pmtu;
505     if (room > 576)
506         room = 576;
507     room -= sizeof(struct iphdr) + icmp_param.replyopts.optlen;
508     room -= sizeof(struct icmphdr);
509
510     icmp_param.data_len = skb_in->len - icmp_param.offset;
511     if (icmp_param.data_len > room)
512         icmp_param.data_len = room;
513     icmp_param.head_len = sizeof(struct icmphdr);
514
515     ip_build_xmit(icmp_socket->sk, icmp_glue_bits, &icmp_param,
516                 icmp_param.data_len + sizeof(struct icmphdr),
517                 &ipc, rt, MSG_DONTWAIT);
518
519 ende:
520     ip_rt_put(rt);
521 out:
522     icmp_xmit_unlock();
523 }
524
525
526 /*
527  *   Handle ICMP_DEST_UNREACH, ICMP_TIME_EXCEED, and ICMP_QUENCH.
528  */
529
530 static void icmp_unreach(struct sk_buff *skb)
531 {
532     struct iphdr *iph;
533     struct icmphdr *icmph;

```

Figure 396: ICMP Reply: linux/net/ipv4/icmp.c

```

536 struct sock *raw_sk;
537 u32 info = 0;
538
539 /*
540  * Incomplete header ?
541  * Only checks for the IP header, there should be an
542  * additional check for longer headers in upper levels.
543  */
544
545 if (!pskb_may_pull(skb, sizeof(struct iphdr))) {
546     ICMP_INC_STATS_BH(IcmpInErrors);
547     return;
548 }
549
550 icmph = skb->h.icmph;
551 iph = (struct iphdr *) skb->data;
552
553 if (iph->ihl<5) {
554     /* Mangled header, drop. */
555     ICMP_INC_STATS_BH(IcmpInErrors);
556     return;
557 }
558
559 if (icmph->type==ICMP_DEST_UNREACH) {
560     switch(icmph->code & 15) {
561         case ICMP_NET_UNREACH:
562             break;
563         case ICMP_HOST_UNREACH:
564             break;
565         case ICMP_PROT_UNREACH:
566             break;
567         case ICMP_PORT_UNREACH:
568             break;
569         case ICMP_FRAG_NEEDED:
570             if (ipv4_config.no_pmtu_disc) {
571                 if (net_ratelimit())
572                     printk(KERN_INFO "ICMP: %u.%u.%u
573                             .%u: fragmentation needed and DF
574                             set.\n",
575                             NIPQUAD(iph->daddr));
576             } else {
577                 info = ip_rt_frag_needed(iph, ntohs(icmph->un.
578                     frag.mtu));
579                 if (!info)
580                     goto out;
581             }
582             break;
583         case ICMP_SR_FAILED:
584             if (net_ratelimit())
585                 printk(KERN_INFO "ICMP: %u.%u.%u.%u:
586                     Source Route Failed.\n", NIPQUAD(iph->
587                     daddr));

```

Figure 397: ICMP Send: linux/net/ipv4/icmp.c

```

584         default :
585             break;
586     }
587     if (icmph->code>NR_ICMP_UNREACH)
588         goto out;
589 } else if (icmph->type == ICMP_PARAMETERPROB) {
590     info = ntohl(icmph->un.gateway)>>24;
591 }
592
593 /*
594  *   Throw it at our lower layers
595  *
596  *   RFC 1122: 3.2.2 MUST extract the protocol ID from the passed
597  *   header.
598  *   RFC 1122: 3.2.2.1 MUST pass ICMP unreachable messages to the transport
599  *   layer.
600  *   RFC 1122: 3.2.2.2 MUST pass ICMP time expired messages to
601  *   transport layer.
602  */
603
604 /*
605  *   Check the other end isnt violating RFC 1122. Some routers send
606  *   bogus responses to broadcast frames. If you see this message
607  *   first check your netmask matches at both ends, if it does then
608  *   get the other vendor to fix their kit.
609  */
610
611 if (!sysctl_icmp_ignore_bogus_error_responses)
612 {
613     if (inet_addr_type(iph->daddr) == RTN_BROADCAST)
614     {
615         if (net_ratelimit())
616             printk(KERN_WARNING "%u.%u.%u.%u sent an invalid
617             ICMP error to a broadcast.\n",
618             NIPQUAD(skb->nh.iph->saddr));
619         goto out;
620     }
621 }
622
623 /* Checkin full IP header plus 8 bytes of protocol to
624  * avoid additional coding at protocol handlers.
625  */
626 if (!pskb_may_pull(skb, iph->ihl*4+8))
627     goto out;
628
629 iph = (struct iphdr *) skb->data;

```

Figure 398: ICMP Send: linux/net/ipv4/icmp.c

```

628
629      /*
630      *    Deliver ICMP message to raw sockets. Pretty useless feature?
631      */
632
633      /* Note: See raw.c and net/raw.h, RAWV4_HTABLE_SIZE==MAX_INET_PROTOS */
634      hash = protocol & (MAX_INET_PROTOS - 1);
635      read_lock(&raw_v4_lock);
636      if ((raw_sk = raw_v4_htable[hash]) != NULL)
637      {
638          while ((raw_sk = __raw_v4_lookup(raw_sk, protocol, iph->saddr,
639                                         iph->daddr, skb->dev->ifindex)) !=
640                NULL) {
641              raw_err(raw_sk, skb, info);
642              raw_sk = raw_sk->next;
643              iph = (struct iphdr *)skb->data;
644          }
645      }
646      read_unlock(&raw_v4_lock);
647
648      /*
649      *    This can't change while we are doing it.
650      *    Callers have obtained BR_NETPROTO_LOCK so
651      *    we are OK.
652      */
653      ipprot = (struct inet_protocol *) inet_protos[hash];
654      while(ipprot != NULL) {
655          struct inet_protocol *nextip;
656
657          nextip = (struct inet_protocol *) ipprot->next;
658
659          /*

```

Figure 399: ICMP Send: linux/net/ipv4/icmp.c

```

661         */
662
663         /* RFC1122: OK. Passes appropriate ICMP errors to the */
664         /* appropriate protocol layer (MUST), as per 3.2.2. */
665
666         if (protocol == ipprot->protocol && ipprot->err_handler)
667             ipprot->err_handler(skb, info);
668
669         ipprot = nextip;
670     }
671 out ;;
672 }
673
674
675 /*
676  *   Handle ICMP_REDIRECT.
677  */
678
679 static void icmp_redirect(struct sk_buff *skb)
680 {
681     struct iphdr *iph;
682     unsigned long ip;
683
684     if (skb->len < sizeof(struct iphdr)) {

```

Figure 400: ICMP Send: `linux/net/ipv4/icmp.c`

```

687     }
688
689     /*
690     *   Get the copied header of the packet that caused the redirect
691     */
692     if (!pskb_may_pull(skb, sizeof(struct iphdr)))
693         return;
694
695     iph = (struct iphdr *) skb->data;
696     ip = iph->daddr;
697
698     switch(skb->h.icmph->code & 7) {
699     case ICMP_REDIRECT_NET:
700     case ICMP_REDIRECT_NETTOS:
701         /*
702         *   As per RFC recommendations now handle it as
703         *   a host redirect.
704         */
705
706     case ICMP_REDIRECT_HOST:
707     case ICMP_REDIRECT_HOSTTOS:
708         ip_rt_redirect (skb->nh.iph->saddr, ip, skb->h.icmph->un.
709             gateway, iph->saddr, iph->tos, skb->dev);
710         break;
711     default:
712         break;

```

Figure 401: ICMP Unreachables: linux/net/ipv4/icmp.c


```

713 }
714
715 /*
716 *   Handle ICMP_ECHO ("ping") requests.
717 *
718 *   RFC 1122: 3.2.2.6 MUST have an echo server that answers ICMP echo
719 *   requests.
720 *   RFC 1122: 3.2.2.6 Data received in the ICMP_ECHO request MUST be included
721 *   in the reply.
722 *   RFC 1812: 4.3.3.6 SHOULD have a config option for silently ignoring echo
723 *   requests, MUST have default=NOT.
724 *   See also WRT handling of options once they are done and working.
725 */
726
727 static void icmp_echo(struct sk_buff *skb)
728 {
729     if (! sysctl_icmp_echo_ignore_all ) {
730         struct icmp_bxm icmp_param;
731
732         icmp_param.data.icmph=*skb->h.icmph;
733         icmp_param.data.icmph.type=ICMP_ECHOREPLY;
734         icmp_param.skbn=skb;
735         icmp_param.offset=0;
736         icmp_param.data_len=skb->len;
737         icmp_param.head_len=sizeof(struct icmphdr);
738         icmp_reply(&icmp_param, skb);
739     }
740 }
741
742 /*
743 *   Handle ICMP Timestamp requests.
744 *   RFC 1122: 3.2.2.8 MAY implement ICMP timestamp requests.
745 *   SHOULD be in the kernel for minimum random latency.
746 *   MUST be accurate to a few minutes.
747 *   MUST be updated at least at 15Hz.
748 */

```

Figure 402: ICMP Unreachables: linux/net/ipv4/icmp.c

```

747 static void icmp_timestamp(struct sk_buff *skb)
748 {
749     struct timeval tv;
750     struct icmp_bxm icmp_param;
751
752     /*
753      *    Too short.
754      */
755
756     if (skb->len < 4) {
757         ICMP_INC_STATS_BH(IcmpInErrors);
758         return;
759     }
760
761     /*
762      *    Fill in the current time as ms since midnight UT:
763      */
764     do_gettimeofday(&tv);
765     icmp_param.data.times[1] = htonl((tv.tv_sec % 86400) * 1000 + tv.tv_usec / 1000);
766     icmp_param.data.times[2] = icmp_param.data.times[1];
767     if (skb_copy_bits(skb, 0, &icmp_param.data.times[0], 4))
768         BUG();
769     icmp_param.data.icmph = *skb->h.icmph;
770     icmp_param.data.icmph.type = ICMP_TIMESTAMPREPLY;
771     icmp_param.data.icmph.code = 0;
772     icmp_param.skb = skb;
773     icmp_param.offset = 0;
774     icmp_param.data_len = 0;
775     icmp_param.head_len = sizeof(struct icmphdr) + 12;
776     icmp_reply(&icmp_param, skb);
777 }
778
779
780 /*
781 *    Handle ICMP_ADDRESS_MASK requests. (RFC950)
782 *
783 * RFC1122 (3.2.2.9). A host MUST only send replies to
784 * ADDRESS_MASK requests if it's been configured as an address mask
785 * agent. Receiving a request doesn't constitute implicit permission to
786 * act as one. Of course, implementing this correctly requires (SHOULD)
787 * a way to turn the functionality on and off. Another one for sysctl(),
788 * I guess. -- MS
789 *

```

Figure 403: ICMP Unreachables: linux/net/ipv4/icmp.c

```

791 *           A router SHOULD have switch turning it on/off.
792 *           This switch MUST be ON by default.
793 *
794 * Gratuitous replies , zero-source replies are not implemented,
795 * that complies with RFC. DO NOT implement them!!! All the idea
796 * of broadcast addrmask replies as specified in RFC950 is broken.
797 * The problem is that it is not uncommon to have several prefixes
798 * on one physical interface . Moreover, addrmask agent can even be
799 * not aware of existing another prefixes .
800 * If source is zero , addrmask agent cannot choose correct prefix.
801 * Gratuitous mask announcements suffer from the same problem.
802 * RFC1812 explains it, but still allows to use ADDRMASK,
803 * that is pretty silly . --ANK
804 *
805 * All these rules are so bizarre , that I removed kernel addrmask
806 * support at all . It is wrong, it is obsolete , nobody uses it in
807 * any case. --ANK
808 *
809 * Furthermore you can do it with a usermode address agent program
810 * anyway...
811 */
812
813 static void icmp_address(struct sk_buff *skb)
814 {
815 #if 0

```

Figure 404: ICMP Unreachables: `linux/net/ipv4/icmp.c`

```

822  * RFC1812 (4.3.3.9).  A router SHOULD listen all replies, and complain
823  *                      loudly if an inconsistency is found.
824  */
825
826  static void icmp_address_reply(struct sk_buff *skb)
827  {
828      struct rtable *rt = (struct rtable*)skb->dst;
829      struct net_device *dev = skb->dev;
830      struct in_device *in_dev;
831      struct in_ifaddr *ifa;
832      u32 mask;
833
834      if (skb->len < 4 || !(rt->rt_flags&RTCF_DIRECTSRC))
835          return;
836
837      in_dev = in_dev_get(dev);
838      if (!in_dev)
839          return;
840      read_lock(&in_dev->lock);
841      if (in_dev->ifa_list &&
842          IN_DEV_LOG_MARTIANS(in_dev) &&
843          IN_DEV_FORWARD(in_dev)) {
844          if (skb_copy_bits(skb, 0, &mask, 4))
845              BUG();
846          for (ifa=in_dev->ifa_list; ifa; ifa = ifa->ifa_next) {
847              if (mask == ifa->ifa_mask && inet_ifa_match(rt->rt_src, ifa))
848                  break;
849          }
850          if (!ifa && net_ratelimit()) {
851              printk(KERN_INFO "Wrong address mask %u.%u.%u.%u from %u.%u.%u.%u\n",
852                     NIPQUAD(mask), dev->name, NIPQUAD(rt->rt_src));
853          }
854      }

```

Figure 405: ICMP Redirects: linux/net/ipv4/icmp.c

```

856     in_dev_put(in_dev);
857 }
858
859 static void icmp_discard(struct sk_buff *skb)
860 {
861 }
862
863 /*
864  *   Deal with incoming ICMP packets.
865  */
866
867 int icmp_rcv(struct sk_buff *skb)
868 {
869     struct icmphdr *icmph = skb->h.icmph;
870     struct rtable *rt = (struct rtable*)skb->dst;
871
872     ICMP_INC_STATS_BH(IcmpInMsgs);
873
874     switch (skb->ip_summed) {
875     case CHECKSUM_HW:
876         if ((u16)csum_fold(skb->csum) == 0)

```

Figure 406: ICMP Echo: `linux/net/ipv4/icmp.c`

```

878     NETDEBUG(printk(KERN_DEBUG "icmp_v4_hw_csum_failure\n"));
879 case CHECKSUM_NONE:
880     if ((u16)csum_fold(skb_checksum(skb, 0, skb->len, 0)))
881         goto error;
882     default ;;
883     }
884
885     if (!pskb_pull(skb, sizeof(struct icmphdr)))
886         goto error;
887
888     /*
889     *    18 is the highest 'known' ICMP type. Anything else is a mystery
890     *
891     *    RFC 1122: 3.2.2 Unknown ICMP messages types MUST be silently
892     *    discarded.
893     */
894     if (icmph->type > NR_ICMP_TYPES)
895         goto error;
896
897     /*
898     *    Parse the ICMP message
899     */
900
901     if (rt->rt_flags&(RTCF_BROADCAST|RTCF_MULTICAST)) {
902         /*
903         *    RFC 1122: 3.2.2.6 An ICMP_ECHO to broadcast MAY be
904         *    silently ignored (we let user decide with a sysctl).
905         *    RFC 1122: 3.2.2.8 An ICMP_TIMESTAMP MAY be silently
906         *    discarded if to broadcast/multicast.
907         */
908         if (icmph->type == ICMP_ECHO &&
909             sysctl_icmp_echo_ignore_broadcasts) {
910             goto error;
911         }
912         if (icmph->type != ICMP_ECHO &&
913             icmph->type != ICMP_TIMESTAMP &&
914             icmph->type != ICMP_ADDRESS &&
915             icmph->type != ICMP_ADDRESSREPLY) {

```

Figure 407: ICMP Timestamp: linux/net/ipv4/icmp.c

```

918     }
919
920     icmp_pointers[icmph->type].input[smp_processor_id()*2*sizeof(struct icmp_mib)/
921     sizeof(unsigned long)]++;
922     (icmp_pointers[icmph->type].handler)(skb);
923
924 drop:
925     kfree_skb(skb);
926     return 0;
927
928 error:
929     ICMP_INC_STATS_BH(IcmpInErrors);
930     goto drop;
931 }
932
933 /*
934  *   A spare long used to speed up statistics updating
935  */
936
937 static unsigned long dummy;
938
939 /*
940  *   Configurable rate limits.
941  *   Someone should check if these default values are correct.
942  *   Note that these values interact with the routing cache GC timeout.
943  *   If you chose them too high they won't take effect, because the
944  *   dst_entry gets expired too early. The same should happen when
945  *   the cache grows too big.
946  */
947
948 int sysctl_icmp_destunreach_time = 1*HZ;
949 int sysctl_icmp_timeexceed_time = 1*HZ;
950 int sysctl_icmp_paramprob_time = 1*HZ;
951 int sysctl_icmp_echoreply_time = 0; /* don't limit it per default. */
952
953 /*
954  *   This table is the definition of how we handle ICMP.
955  */
956
957 static struct icmp_control icmp_pointers[NR_ICMP_TYPES+1] = {
958     /* ECHO REPLY (0) */
959     { &icmp_statistics[0].IcmpOutEchoReps, &icmp_statistics[0].IcmpInEchoReps,
960       icmp_discard, 0, &sysctl_icmp_echoreply_time},
961     { &dummy, &icmp_statistics[0].IcmpInErrors, icmp_discard, 1, },
962 }

```

Figure 408: ICMP Address Mask (not!): `linux/net/ipv4/icmp.c`

```

959 /* DEST UNREACH (3) */
960 { &icmp_statistics [0].IcmpOutDestUnreachs, &icmp_statistics[0].IcmpInDestUnreachs,
    icmp_unreach, 1, &sysctl_icmp_destunreach_time },
961 /* SOURCE QUENCH (4) */
962 { &icmp_statistics [0].IcmpOutSrcQuenchs, &icmp_statistics[0].IcmpInSrcQuenchs,
    icmp_unreach, 1, },
963 /* REDIRECT (5) */
964 { &icmp_statistics [0].IcmpOutRedirects, &icmp_statistics[0].IcmpInRedirects,
    icmp_redirect, 1, },
965 { &dummy, &icmp_statistics[0].IcmpInErrors, icmp_discard, 1, },
966 { &dummy, &icmp_statistics[0].IcmpInErrors, icmp_discard, 1, },
967 /* ECHO (8) */
968 { &icmp_statistics [0].IcmpOutEchos, &icmp_statistics[0].IcmpInEchos, icmp_echo, 0, },
969 { &dummy, &icmp_statistics[0].IcmpInErrors, icmp_discard, 1, },
970 { &dummy, &icmp_statistics[0].IcmpInErrors, icmp_discard, 1, },
971 /* TIME EXCEEDED (11) */
972 { &icmp_statistics [0].IcmpOutTimeExcds, &icmp_statistics[0].IcmpInTimeExcds,
    icmp_unreach, 1, &sysctl_icmp_timeexceed_time },
973 /* PARAMETER PROBLEM (12) */
974 { &icmp_statistics [0].IcmpOutParmProbs, &icmp_statistics[0].IcmpInParmProbs,
    icmp_unreach, 1, &sysctl_icmp_paramprob_time },
975 /* TIMESTAMP (13) */
976 { &icmp_statistics [0].IcmpOutTimestamps, &icmp_statistics[0].IcmpInTimestamps,
    icmp_timestamp, 0, },
977 /* TIMESTAMP REPLY (14) */
978 { &icmp_statistics [0].IcmpOutTimestampReps, &icmp_statistics[0].IcmpInTimestampReps,
    icmp_discard, 0, },
979 /* INFO (15) */
980 { &dummy, &dummy, icmp_discard, 0, },
981 /* INFO REPLY (16) */
982 { &dummy, &dummy, icmp_discard, 0, },
983 /* ADDR MASK (17) */
984 { &icmp_statistics [0].IcmpOutAddrMasks, &icmp_statistics[0].IcmpInAddrMasks,
    icmp_address, 0, },
985 /* ADDR MASK REPLY (18) */
986 { &icmp_statistics [0].IcmpOutAddrMaskReps, &icmp_statistics[0].IcmpInAddrMaskReps,
    icmp_address_reply, 0, }
987 };
988
989 void _init_icmp_init(struct net_proto_family *ops)
990 {
991     int err;
992
993     icmp_inode.i_mode = S_IFSOCK;
994     icmp_inode.i_sock = 1;
995     icmp_inode.i_uid = 0;

```

Figure 409: ICMP Address Reply: linux/net/ipv4/icmp.c


```
997     init_waitqueue_head(&icmp_inode.i_wait);  
998     init_waitqueue_head(&icmp_inode.u.socket_i_wait);
```

Figure 410: ICMP discard: `linux/net/ipv4/icmp.c`

```

1005         panic("Failed to create the ICMP control socket.\n");
1006         icmp_socket->sk->allocation=GFP_ATOMIC;
1007         icmp_socket->sk->sndbuf = SK_WMEM_MAX*2;
1008         icmp_socket->sk->protinfo.af_inet.ttl = MAXTTL;
1009
1010         /* Unhash it so that IP input processing does not even
1011          * see it, we do not wish this socket to see incoming
1012          * packets.
1013          */
1014         icmp_socket->sk->prot->unhash(icmp_socket->sk);
1015     }

```

Figure 411: ICMP Receive General Packet: `linux/net/ipv4/icmp.c`



Figure 412: ICMP Statistics: `linux/net/ipv4/icmp.c`



Figure 413: ICMP Statistics: `linux/net/ipv4/icmp.c`



Figure 414: ICMP Initialisation: `linux/net/ipv4/icmp.c`

```

102 #define IP_MAX_MEMBERSHIPS 20
103
104 #ifdef CONFIG_IP_MULTICAST
105
106
107 /* Parameter names and values are taken from igmp-v2-06 draft */
108
109 #define IGMP_V1_Router_Present_Timeout      (400*HZ)
110 #define IGMP_Unsolicited_Report_Interval    (10*HZ)
111 #define IGMP_Query_Response_Interval        (10*HZ)
112 #define IGMP_Unsolicited_Report_Count      2
113
114
115 #define IGMP_Initial_Report_Delay           (1*HZ)
116
117 /* IGMP_Initial_Report_Delay is not from IGMP specs!
118  * IGMP specs require to report membership immediately after
119  * joining a group, but we delay the first report by a
120  * small interval. It seems more natural and still does not
121  * contradict to specs provided this delay is small enough.
122  */
123
124 #define IGMP_V1_SEEN(in_dev) ((in_dev)->mr_v1_seen && (long)(jiffies - (in_dev)->
    mr_v1_seen) < 0)
125
126 #endif

```

Figure 415: Internet Group Management Protocol - Data:
linux/net/ipv4/igmp.c

```
128 static void ip_ma_put(struct ip_mc_list *im)
129 {
130     if (atomic_dec_and_test(&im->refcnt)) {
131         in_dev_put(im->interface);
132         kfree(im);
133     }
134 }
```

Figure 416: IGMP Configure multicast interface: `linux/net/ipv4/igmp.c`

```

142 static __inline__ void igmp_stop_timer(struct ip_mc_list *im)
143 {
144     spin_lock_bh(&im->lock);
145     if (del_timer(&im->timer))
146         atomic_dec(&im->refcnt);
147     im->tm_running=0;
148     im->reporter = 0;
149     im->unsolicit_count = 0;
150     spin_unlock_bh(&im->lock);
151 }
152
153 /* It must be called with locked im->lock */
154 static void igmp_start_timer(struct ip_mc_list *im, int max_delay)
155 {
156     int tv=net_random() % max_delay;
157
158     im->tm_running=1;
159     if (!mod_timer(&im->timer, jiffies+tv+2))
160         atomic_inc(&im->refcnt);
161 }
162
163 static void igmp_mod_timer(struct ip_mc_list *im, int max_delay)
164 {
165     spin_lock_bh(&im->lock);
166     im->unsolicit_count = 0;
167     if (del_timer(&im->timer)) {
168         if ((long)(im->timer.expires-jiffies) < max_delay) {
169             add_timer(&im->timer);
170             im->tm_running=1;
171             spin_unlock_bh(&im->lock);
172             return;
173         }
174         atomic_dec(&im->refcnt);
175     }
176     igmp_start_timer(im, max_delay);
177     spin_unlock_bh(&im->lock);
178 }

```

Figure 417: IGMP Timer Management: `linux/net/ipv4/igmp.c`


```

257 static void igmp_timer_expire(unsigned long data)
258 {
259     struct ip_mc_list *im=(struct ip_mc_list *)data;
260     struct in_device *in_dev = im->interface;
261     int err;
262
263     spin_lock(&im->lock);
264     im->tm_running=0;
265
266     if (IGMP_V1_SEEN(in_dev))
267         err = igmp_send_report(in_dev->dev, im->multiaddr,
268                               IGMP_HOST_MEMBERSHIP_REPORT);
269     else
270         err = igmp_send_report(in_dev->dev, im->multiaddr,
271                               IGMP_HOST_NEW_MEMBERSHIP_REPORT);
272
273     /* Failed. Retry later. */
274     if (err) {
275         if (!in_dev->dead)
276             igmp_start_timer(im, IGMP_Unsolicited_Report_Interval);
277         goto out;
278     }
279
280     if (im->unsolicit_count) {
281         im->unsolicit_count--;
282         igmp_start_timer(im, IGMP_Unsolicited_Report_Interval);
283     }
284     im->reporter = 1;
285 out:
286     spin_unlock(&im->lock);
287     ip_ma_put(im);
288 }

```

Figure 418: IGMP Timer Expired: linux/net/ipv4/igmp.c

```

185 #define IGMP_SIZE (sizeof(struct igmp_hdr)+sizeof(struct ip_hdr)+4)
186
187 /* Don't just hand NF_HOOK skb->dst->output, in case netfilter hook
188    changes route */
189 static inline int
190 output_maybe_reroute(struct sk_buff *skb)
191 {
192     return skb->dst->output(skb);
193 }
194
195 static int igmp_send_report(struct net_device *dev, u32 group, int type)
196 {
197     struct sk_buff *skb;
198     struct ip_hdr *iph;
199     struct igmp_hdr *ih;
200     struct rtable *rt;
201     u32 dst;
202
203     /* According to IGMPv2 specs, LEAVE messages are
204        * sent to all-routers group.
205        */
206     dst = group;
207     if (type == IGMP_HOST_LEAVE_MESSAGE)
208         dst = IGMP_ALL_ROUTER;
209
210     if (ip_route_output(&rt, dst, 0, 0, dev->ifindex))
211         return -1;
212     if (rt->rt_src == 0) {
213         ip_rt_put(rt);
214         return -1;
215     }
216
217     skb=alloc_skb(IGMP_SIZE+dev->hard_header_len+15, GFP_ATOMIC);
218     if (skb == NULL) {
219         ip_rt_put(rt);
220         return -1;
221     }

```

Figure 419: IGMP Send Report: `linux/net/ipv4/igmp.c`

```

223     skb->dst = &rt->u.dst;
224
225     skb_reserve(skb, (dev->hard_header_len+15)&~15);
226
227     skb->nh.iph = iph = (struct iphdr *)skb_put(skb, sizeof(struct iphdr)+4);
228
229     iph->version = 4;
230     iph->ihl     = (sizeof(struct iphdr)+4)>>2;
231     iph->tos     = 0;
232     iph->frag_off = htons(IP_DF);
233     iph->ttl     = 1;
234     iph->daddr   = dst;
235     iph->saddr   = rt->rt_src;
236     iph->protocol = IPPROTO_IGMP;
237     iph->tot_len = htons(IGMP_SIZE);
238     ip_select_ident(iph, &rt->u.dst, NULL);
239     ((u8*)&iph[1])[0] = IPOPT_RA;
240     ((u8*)&iph[1])[1] = 4;
241     ((u8*)&iph[1])[2] = 0;
242     ((u8*)&iph[1])[3] = 0;
243     ip_send_check(iph);
244
245     ih = (struct igmp_hdr *)skb_put(skb, sizeof(struct igmp_hdr));
246     ih->type=type;
247     ih->code=0;
248     ih->csum=0;
249     ih->group=group;
250     ih->csum=ip_compute_csum((void *)ih, sizeof(struct igmp_hdr));
251
252     return NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev,
253                  output_maybe_reroute);
254 }

```

Figure 420: IGMP Send Report: `linux/net/ipv4/igmp.c`

```

288 static void igmp_heard_report(struct in_device *in_dev, u32 group)
289 {
290     struct ip_mc_list *im;
291
292     /* Timers are only set for non-local groups */
293
294     if (group == IGMP_ALL_HOSTS)
295         return;
296
297     read_lock(&in_dev->lock);
298     for (im=in_dev->mc_list; im!=NULL; im=im->next) {
299         if (im->multiaddr == group) {
300             igmp_stop_timer(im);
301             break;
302         }
303     }
304     read_unlock(&in_dev->lock);
305 }

```

Figure 421: IGMP Heard a Report: `linux/net/ipv4/igmp.c`

```

307 static void igmp_heard_query(struct in_device *in_dev, unsigned char max_resp_time,
308                             u32 group)
309 {
310     struct ip_mc_list *im;
311     int max_delay;
312
313     max_delay = max_resp_time*(HZ/IGMP_TIMER_SCALE);
314
315     if (max_resp_time == 0) {
316         /* Alas, old v1 router presents here. */
317
318         max_delay = IGMP_Query_Response.Interval;
319         in_dev->mr_v1_seen = jiffies + IGMP_V1_Router_Present_Timeout;
320         group = 0;
321     }
322
323     /*
324     * - Start the timers in all of our membership records
325     *   that the query applies to for the interface on
326     *   which the query arrived excl. those that belong
327     *   to a "local" group (224.0.0.X)
328     * - For timers already running check if they need to
329     *   be reset.
330     * - Use the igmp->igmp_code field as the maximum
331     *   delay possible
332     */
333     read_lock(&in_dev->lock);
334     for (im=in_dev->mc_list; im!=NULL; im=im->next) {
335         if (group && group != im->multiaddr)
336             continue;
337         if (im->multiaddr == IGMP_ALL_HOSTS)
338             continue;
339         igmp_mod_timer(im, max_delay);
340     }
341     read_unlock(&in_dev->lock);
342 }

```

Figure 422: IGMP Heard a Query: linux/net/ipv4/igmp.c

```

344 int igmp_rcv(struct sk_buff *skb)
345 {
346     /* This basically follows the spec line by line -- see RFC1112 */
347     struct igmp_hdr *ih = skb->h.igmp;
348     struct in_device *in_dev = in_dev_get(skb->dev);
349     int len = skb->len;
350
351     if (in_dev==NULL) {
352         kfree_skb(skb);
353         return 0;
354     }
355
356     if (skb_is_nonlinear(skb)) {
357         if (skb_linearize(skb, GFP_ATOMIC) != 0) {
358             kfree_skb(skb);
359             return -ENOMEM;
360         }
361         ih = skb->h.igmp;
362     }
363
364     if (len < sizeof(struct igmp_hdr) || ip_compute_csum((void *)ih, len)) {
365         in_dev_put(in_dev);
366         kfree_skb(skb);
367         return 0;
368     }
369
370     switch (ih->type) {
371     case IGMP_HOST_MEMBERSHIP_QUERY:
372         igmp_heard_query(in_dev, ih->code, ih->group);
373         break;
374     case IGMP_HOST_MEMBERSHIP_REPORT:
375     case IGMP_HOST_NEW_MEMBERSHIP_REPORT:
376         /* Is it our report looped back? */
377         if (((struct rtable*)skb->dst)->key.iif == 0)
378             break;
379         igmp_heard_report(in_dev, ih->group);
380         break;
381     case IGMP_PIM:
382     #ifdef CONFIG_IP_PIMSM_V1
383         in_dev_put(in_dev);
384         return pim_rcv_v1(skb);
385     #endif
386     case IGMP_DVMRP:
387     case IGMP_TRACE:
388     case IGMP_HOST_LEAVE_MESSAGE:
389     case IGMP_MTRACE:

```

Figure 423: IGMP Receive: linux/net/ipv4/igmp.c

```

398 }
399
400 #endif
401
402
403 /*
404  *   Add a filter to a device
405  */
406
407 static void ip_mc_filter_add(struct in_device *in_dev, u32 addr)
408 {
409     char buf[MAX_ADDR_LEN];
410     struct net_device *dev = in_dev->dev;
411
412     /* Checking for IFF_MULTICAST here is WRONG-WRONG-WRONG.
413        We will get multicast token leakage, when IFF_MULTICAST
414        is changed. This check should be done in dev->set_multicast_list
415        routine. Something sort of:
416        if (dev->mc_list && dev->flags&IFF_MULTICAST) { do it; }
417        --ANK
418        */
419     if (arp_mc_map(addr, buf, dev, 0) == 0)
420         dev_mc_add(dev, buf, dev->addr_len, 0);
421 }
422
423 /*
424  *   Remove a filter from a device
425  */

```

Figure 424: IGMP Add and Delete Filter from interface:
linux/net/ipv4/igmp.c

```

427 static void ip_mc_filter_del (struct in_device *in_dev, u32 addr)
428 {
429     char buf[MAX_ADDR_LEN];
430     struct net_device *dev = in_dev->dev;
431
432     if (arp_mc_map(addr, buf, dev, 0) == 0)
433         dev_mc_delete(dev, buf, dev->addr_len, 0);
434 }
435
436 static void igmp_group_dropped(struct ip_mc_list *im)
437 {
438     #ifdef CONFIG_IP_MULTICAST
439         int reporter;
440     #endif
441
442     if (im->loaded) {
443         im->loaded = 0;
444         ip_mc_filter_del (im->interface, im->multiaddr);
445     }
446
447     #ifdef CONFIG_IP_MULTICAST
448         if (im->multiaddr == IGMP_ALL_HOSTS)
449             return;
450
451         reporter = im->reporter;
452         igmp_stop_timer(im);
453
454         if (reporter && !IGMP_V1_SEEN(im->interface))
455             igmp_send_report(im->interface->dev, im->multiaddr,
456                             IGMP_HOST_LEAVE_MESSAGE);
457     #endif
458 }
459
460 static void igmp_group_added(struct ip_mc_list *im)
461 {
462     if (im->loaded == 0) {
463         im->loaded = 1;
464         ip_mc_filter_add (im->interface, im->multiaddr);
465     }
466 }

```

Figure 425: IGMP Group added or dropped: `linux/net/ipv4/igmp.c`


```

477  /*
478  *   Multicast list managers
479  */
480
481
482  /*
483  *   A socket has joined a multicast group on device dev.
484  */
485
486  void ip_mc_inc_group(struct in_device *in_dev, u32 addr)
487  {
488      struct ip_mc_list *im;
489
490      ASSERT_RTNL();
491
492      for (im=in_dev->mc_list; im; im=im->next) {
493          if (im->multiaddr == addr) {
494              im->users++;
495              goto out;
496          }
497      }
498
499      im = (struct ip_mc_list *)kmalloc(sizeof(*im), GFP_KERNEL);
500      if (!im)
501          goto out;
502
503      im->users=1;
504      im->interface=in_dev;
505      in_dev_hold(in_dev);
506      im->multiaddr=addr;
507      atomic_set(&im->refcnt, 1);
508      spin_lock_init (&im->lock);
509  #ifdef CONFIG_IP_MULTICAST
510      im->tm_running=0;
511      init_timer (&im->timer);
512      im->timer.data=(unsigned long)im;
513      im->timer.function=&igmp_timer_expire;
514      im->unsolicit_count = IGMP_Unsolicited_Report_Count;
515      im->reporter = 0;
516  #endif
517      im->loaded = 0;
518      write_lock_bh(&in_dev->lock);

```

Figure 426: IGMP - socket joins group: `linux/net/ipv4/igmp.c`

```

524         ip_rt_multicast_event (in_dev);
525 out:
526     return;
527 }
528
529 /*
530  *   A socket has left a multicast group on device dev
531  */
532
533 int ip_mc_dec_group(struct in_device *in_dev, u32 addr)
534 {
535     int err = -ESRCH;
536     struct ip_mc_list *i, **ip;
537
538     ASSERT_RTNL();
539
540     for (ip=&in_dev->mc_list; (i=*ip)!=NULL; ip=&i->next) {
541         if (i->multiaddr==addr) {
542             if (--i->users == 0) {
543                 write_lock_bh(&in_dev->lock);
544                 *ip = i->next;
545                 write_unlock_bh(&in_dev->lock);
546                 igmp_group_dropped(i);
547
548                 if (in_dev->dev->flags & IFF_UP)
549                     ip_rt_multicast_event (in_dev);

```

Figure 427: IGMP Socket leaves group: linux/net/ipv4/igmp.c

```

630         imr->imr_ifindex = dev->ifindex;
631         idev = _in_dev_get(dev);
632     }
633     return idev;
634 }
635
636 /*
637  *   Join a socket to a group
638  */
639 int sysctl_igmp_max_memberships = IP_MAX_MEMBERSHIPS;
640
641 int ip_mc_join_group(struct sock *sk , struct ip_mreqn *imr)
642 {
643     int err;
644     u32 addr = imr->imr_multiaddr.s_addr;
645     struct ip_mc_socklist *iml, *i;
646     struct in_device *in_dev;
647     int count = 0;
648
649     if (!MULTICAST(addr))
650         return -EINVAL;
651
652     rtnl_shlock();
653
654     if (!imr->imr_ifindex)
655         in_dev = ip_mc_find_dev(imr);
656     else {
657         in_dev = inetdev_by_index(imr->imr_ifindex);

```

Figure 428: IGMP join socket to group: `linux/net/ipv4/igmp.c`

```

659         _in_dev_put(in_dev);
660     }
661
662     if (!in_dev) {
663         iml = NULL;
664         err = -ENODEV;
665         goto done;
666     }
667
668     iml = (struct ip_mc_socklist *)sock_kmalloc(sk, sizeof(*iml), GFP_KERNEL);
669
670     err = -EADDRINUSE;
671     for (i=sk->proto.af_inet.mc_list; i; i=i->next) {
672         if (memcmp(&i->multi, imr, sizeof(*imr)) == 0) {
673             /* New style additions are reference counted */
674             if (imr->imr_address.s_addr == 0) {
675                 i->count++;
676                 err = 0;
677             }
678             goto done;
679         }
680         count++;
681     }
682     err = -ENOBUFFS;
683     if (iml == NULL || count >= sysctl_igmp_max_memberships)
684         goto done;
685     memcpy(&iml->multi, imr, sizeof(*imr));
686     iml->next = sk->proto.af_inet.mc_list;
687     iml->count = 1;
688     sk->proto.af_inet.mc_list = iml;
689     ip_mc_inc_group(in_dev, addr);

```

Figure 429: IGMP join socket to group: `linux/net/ipv4/igmp.c`

```

695     if (iml)
696         sock_kfree_s(sk, iml, sizeof(*iml));
697     return err;
698 }
699
700 /*
701  *   Ask a socket to leave a group.
702  */
703
704 int ip_mc_leave_group(struct sock *sk, struct ip_mreqn *imr)
705 {
706     struct ip_mc_socklist *iml, **imlp;
707
708     rtnl_lock();
709     for (imlp=&sk->protinfo.af_inet.mc_list; (iml=*imlp)!=NULL; imlp=&iml->next
710         ) {
711         if (iml->multi.imr_multiaddr.s_addr==imr->imr_multiaddr.s_addr &&
712             iml->multi.imr_address.s_addr==imr->imr_address.s_addr &&
713             (!imr->imr_ifindex || iml->multi.imr_ifindex==imr->imr_ifindex)) {
714             struct in_device *in_dev;
715             if (--iml->count) {
716                 rtnl_unlock();
717                 return 0;
718             }
719             *imlp = iml->next;
720
721             in_dev = inetdev_by_index(iml->multi.imr_ifindex);
722             if (in_dev) {
723                 ip_mc_dec_group(in_dev, imr->imr_multiaddr.s_addr);
724                 in_dev_put(in_dev);

```

Figure 430: IGMP: linux/net/ipv4/igmp.c

```

730     }
731     rtnl_unlock();
732     return -EADDRNOTAVAIL;
733 }
734
735 /*
736  *   A socket is closing.
737  */
738
739 void ip_mc_drop_socket(struct sock *sk)
740 {
741     struct ip_mc_socklist *iml;
742
743     if (sk->protinfo.af_inet.mc_list == NULL)
744         return;
745
746     rtnl_lock();
747     while ((iml=sk->protinfo.af_inet.mc_list) != NULL) {
748         struct in_device *in_dev;
749         sk->protinfo.af_inet.mc_list = iml->next;

```

Figure 431: IGMP - socket closing: `linux/net/ipv4/igmp.c`

```

554             err = 0;
555             break;
556         }
557     }
558     return -ESRCH;
559 }
560
561 /* Device going down */
562
563 void ip_mc_down(struct in_device *in_dev)
564 {
565     struct ip_mc_list *i;
566
567     ASSERT_RTNL();
568
569     for (i=in_dev->mc_list; i; i=i->next)
570         igmp_group_dropped(i);
571
572     ip_mc_dec_group(in_dev, IGMP_ALL_HOSTS);
573 }
574
575 /* Device going up */
576
577 void ip_mc_up(struct in_device *in_dev)
578 {

```

Figure 432: IGMP - Device Goes Down or Comes Up: `linux/net/ipv4/igmp.c`

```

584     for (i=in_dev->mc_list; i; i=i->next)
585         igmp_group_added(i);
586     }
587 }
588
589 /*
590  *   Device is about to be destroyed: clean up.
591  */
592
593 void ip_mc_destroy_dev(struct in_device *in_dev)
594 {
595     struct ip_mc_list *i;
596
597     ASSERT_RTNL();
598
599     write_lock_bh(&in_dev->lock);
600     while ((i = in_dev->mc_list) != NULL) {
601         in_dev->mc_list = i->next;

```

Figure 433: IGMP - Destroy Device: `linux/net/ipv4/igmp.c`


```

603         igmp_group_dropped(i);
604         ip_ma_put(i);
605
606         write_lock_bh(&in_dev->lock);
607     }
608     write_unlock_bh(&in_dev->lock);
609 }
610
611 static struct in_device * ip_mc_find_dev(struct ip_mreqn *imr)
612 {
613     struct rtable *rt;
614     struct net_device *dev = NULL;
615     struct in_device *idev = NULL;
616
617     if (imr->imr_address.s_addr) {
618         dev = ip_dev_find(imr->imr_address.s_addr);
619         if (!dev)
620             return NULL;
621         _dev_put(dev);
622     }
623
624     if (!dev && !ip_route_output(&rt, imr->imr_multiaddr.s_addr, 0, 0, 0)) {
625

```

Figure 434: IGMP Find Device: `linux/net/ipv4/igmp.c`

```

1055 }
1056
1057
1058 static int ipmr_device_event(struct notifier_block *this, unsigned long event, void *ptr)
1059 {
1060     struct vif_device *v;
1061     int ct;
1062     if (event != NETDEV_UNREGISTER)
1063         return NOTIFY_DONE;
1064     v=&vif_table[0];
1065     for(ct=0;ct<maxvif;ct++,v++) {
1066         if (v->dev==ptr)
1067             vif_delete(ct);
1068     }
1069     return NOTIFY_DONE;
1070 }
1071
1072
1073 static struct notifier_block ip_mr_notifier={
1074     ipmr_device_event,

```

Figure 435: IPMR - device notifier: `linux/net/ipv4/ipmr.c`

```

752         ip_mc_dec_group(in_dev, iml->multi.imr_multiaddr.s_addr);
753         in_dev_put(in_dev);
754     }
755     sock_kfree_s(sk, iml, sizeof(*iml));
756
757 }
758 rtnl_unlock();
759 }
760
761 int ip_check_mc(struct in_device *in_dev, u32 mc_addr)
762 {
763     struct ip_mc_list *im;
764
765     read_lock(&in_dev->lock);

```

Figure 436: IGMP Check if address is in use on any interface:
linux/net/ipv4/igmp.c

```

770     }
771     }
772     read_unlock(&in_dev->lock);
773     return 0;
774 }
775
776
777 #ifdef CONFIG_IP_MULTICAST
778
779 int ip_mc_procinfo(char *buffer, char **start, off_t offset, int length)
780 {
781     off_t pos=0, begin=0;
782     struct ip_mc_list *im;
783     int len=0;
784     struct net_device *dev;
785
786     len=sprintf(buffer, "Idx\tDevice\tCount\tQuerier\tGroup\tUsers\tTimer\t
787                 tReporter\n");
788
789     read_lock(&dev_base_lock);
790     for(dev = dev_base; dev; dev = dev->next) {
791         struct in_device *in_dev = in_dev_get(dev);
792         char *querier = "NONE";

```

Figure 437: IGMP - procfs - report membership etc: `linux/net/ipv4/igmp.c`

```

793         if (in_dev == NULL)
794             continue;
795
796         querier = IGMP_V1_SEEN(in_dev) ? "V1" : "V2";
797
798         len+=sprintf(buffer+len,"%d\t%-10s:_%5d_%7s\n",
799                     dev->ifindex, dev->name, dev->mc_count, querier);
800
801         read_lock(&in_dev->lock);
802         for (im = in_dev->mc_list; im; im = im->next) {
803             len+=sprintf(buffer+len,
804                          "\t\t\t\t\t%08X_%5d_%d:%08X\t\t%d\n",
805                          im->multiaddr, im->users,
806                          im->t.m_running, im->timer.expires-jiffies, im->
807                          reporter);
808
809             pos=begin+len;
810             if (pos<offset)
811             {
812                 len=0;
813                 begin=pos;
814             }
815             if (pos>offset+length) {
816                 read_unlock(&in_dev->lock);
817                 in_dev_put(in_dev);
818                 goto done;
819             }
820             read_unlock(&in_dev->lock);
821             in_dev_put(in_dev);
822         }
823     done:
824         read_unlock(&dev_base_lock);

```

Figure 438: IGMP - procs - report membership etc: `linux/net/ipv4/igmp.c`

```

64 #if defined(CONFIG_IP_PIMSM_V1) || defined(CONFIG_IP_PIMSM_V2)
65 #define CONFIG_IP_PIMSM 1
66 #endif
67
68 static struct sock *mroute_socket;
69
70
71 /* Big lock, protecting vif table, mrt cache and mroute socket state.
72    Note that the changes are semaphored via rtnl_lock.
73    */
74
75 static rwlock_t mrt_lock = RW_LOCK_UNLOCKED;
76
77 /*
78    *      Multicast router control variables
79    */
80
81 static struct vif_device vif_table [MAXVIFS];          /* Devices          */
82 static int maxvif;
83
84 #define VIF_EXISTS(idx) (vif_table[idx].dev != NULL)
85
86 int mroute_do_assert = 0;                             /* Set in PIM assert */
87 int mroute_do_pim = 0;
88
89 static struct mfc_cache *mfc_cache_array[MFC_LINES]; /* Forwarding cache */
90
91 static struct mfc_cache *mfc_unres_queue;             /* Queue of unresolved
92    */
93 atomic_t cache_resolve_queue_len;                    /* Size of unresolved */
94
95 /* Special spinlock for queue of unresolved entries */
96 static spinlock_t mfc_unres_lock = SPIN_LOCK_UNLOCKED;

```

Figure 439: Internet Multicast Routing Support - Data: linux/net/ipv4/ipmr.c

```

97  /* We return to original Alan's scheme. Hash table of resolved
98     entries is changed only in process context and protected
99     with weak lock mrt_lock. Queue of unresolved entries is protected
100    with strong spinlock mfc_unres_lock.
101
102    In this case data path is free of exclusive locks at all.
103    */
104
105    kmem_cache_t *mrt_cachep;
106
107    static int ip_mr_forward(struct sk_buff *skb, struct mfc_cache *cache, int local);
108    static int ipmr_cache_report(struct sk_buff *pkt, vifi_t vifi, int assert);
109    static int ipmr_fill_mroute(struct sk_buff *skb, struct mfc_cache *c, struct rtmsg *rtm);
110
111    extern struct inet_protocol pim_protocol;
112
113    static struct timer_list ipmr_expire_timer;

```

Figure 440: Internet Multicast Routing Support - Data:
linux/net/ipv4/ipmr.c

```

117 static
118 struct net_device *ipmr_new_tunnel(struct vifctl *v)
119 {
120     struct net_device *dev;
121
122     dev = __dev_get_by_name("tunl0");
123
124     if (dev) {
125         int err;
126         struct ifreq ifr;
127         mm_segment_t oldfs;
128         struct ip_tunnel_parm p;
129         struct in_device *in_dev;
130
131         memset(&p, 0, sizeof(p));
132         p.iph.daddr = v->vifc_rmt_addr.s_addr;
133         p.iph.saddr = v->vifc_lcl_addr.s_addr;
134         p.iph.version = 4;
135         p.iph.ihl = 5;
136         p.iph.protocol = IPPROTO_IP;
137         sprintf(p.name, "dvmrp%d", v->vifc_vifi);
138         ifr.ifr_ifru.ifru_data = (void*)&p;
139
140         oldfs = get_fs(); set_fs(KERNEL_DS);
141         err = dev->do_ioctl(dev, &ifr, SIOCADDTUNNEL);
142         set_fs(oldfs);
143
144         dev = NULL;
145
146         if (err == 0 && (dev = __dev_get_by_name(p.name)) != NULL) {
147             dev->flags |= IFF_MULTICAST;
148
149             in_dev = __in_dev_get(dev);
150             if (in_dev == NULL && (in_dev = inetdev_init(dev)) == NULL)
151                 goto failure;
152             in_dev->cnf.rp_filter = 0;
153
154             if (dev_open(dev))
155                 goto failure;
156         }
157     }
158     return dev;
159
160 failure:
161     unregister_netdevice(dev);
162     return NULL;
163 }

```

Figure 441: IPMR - new vif/tunnel: linux/net/ipv4/ipmr.c


```

165 #ifdef CONFIG_IP_PIMSM
166
167 static int reg_vif_num = -1;
168
169 static int reg_vif_xmit(struct sk_buff *skb, struct net_device *dev)
170 {
171     read_lock(&mrt_lock);
172     ((struct net_device_stats *)dev->priv)->tx_bytes += skb->len;
173     ((struct net_device_stats *)dev->priv)->tx_packets++;
174     ipmr_cache_report(skb, reg_vif_num, IGMPMSG_WHOLEPKT);
175     read_unlock(&mrt_lock);
176     kfree_skb(skb);
177     return 0;
178 }
179
180 static struct net_device_stats * reg_vif_get_stats (struct net_device *dev)
181 {
182     return (struct net_device_stats *)dev->priv;
183 }

```

Figure 442: IPMR - register a vif: linux/net/ipv4/ipmr.c

```

185 static
186 struct net_device *ipmr_reg_vif(struct vifctl *v)
187 {
188     struct net_device *dev;
189     struct in_device *in_dev;
190     int size;
191
192     size = sizeof(*dev) + sizeof(struct net_device_stats);
193     dev = kmalloc(size, GFP_KERNEL);
194     if (!dev)
195         return NULL;
196
197     memset(dev, 0, size);
198
199     dev->priv = dev + 1;
200
201     strcpy(dev->name, "pimreg");
202
203     dev->type           = ARPHRD_PIMREG;
204     dev->mtu           = 1500 - sizeof(struct iphdr) - 8;
205     dev->flags         = IFF_NOARP;
206     dev->hard_start_xmit = reg_vif_xmit;
207     dev->get_stats     = reg_vif_get_stats;
208     dev->features      |= NETIF_F_DYNALLOC;
209
210     if (register_netdevice(dev)) {
211         kfree(dev);
212         return NULL;
213     }
214     dev->iflink = 0;
215
216     if ((in_dev = inetdev_init(dev)) == NULL)
217         goto failure;
218
219     in_dev->cnf.rp_filter = 0;
220
221     if (dev_open(dev))
222         goto failure;
223
224     return dev;
225
226 failure:
227     unregister_netdevice(dev);
228     return NULL;
229 }
230 #endif

```

Figure 443: IPMR - register a vif: linux/net/ipv4/ipmr.c

```

236 static int vif_delete(int vifi)
237 {
238     struct vif_device *v;
239     struct net_device *dev;
240     struct in_device *in_dev;
241
242     if (vifi < 0 || vifi >= maxvif)
243         return -EADDRNOTAVAIL;
244
245     v = &vif_table[vifi];
246
247     write_lock_bh(&mrt_lock);
248     dev = v->dev;
249     v->dev = NULL;
250
251     if (!dev) {
252         write_unlock_bh(&mrt_lock);
253         return -EADDRNOTAVAIL;
254     }
255
256 #ifdef CONFIG_IP_PIMSM
257     if (vifi == reg_vif_num)
258         reg_vif_num = -1;
259 #endif
260
261     if (vifi + 1 == maxvif) {
262         int tmp;
263         for (tmp=vifi-1; tmp>=0; tmp--) {
264             if (VIF_EXISTS(tmp))
265                 break;
266         }
267         maxvif = tmp+1;
268     }
269
270     write_unlock_bh(&mrt_lock);
271
272     dev_set_allmulti(dev, -1);
273
274     if ((in_dev = __in_dev_get(dev)) != NULL) {
275         in_dev->cnf.mc_forwarding--;
276         ip_rt_multicast_event(in_dev);
277     }
278
279     if (v->flags&(VIFF_TUNNEL|VIFF_REGISTER))
280         unregister_netdevice(dev);
281
282     dev_put(dev);
283     return 0;
284 }

```

Figure 444: IPMR - delete a vif: `linux/net/ipv4/ipmr.c`

```

377 static int vif_add(struct vifctl *vifc, int mrtsock)
378 {
379     int vifi = vifc->vifc_vifi;
380     struct vif_device *v = &vifc_table[ vifi ];
381     struct net_device *dev;
382     struct in_device *in_dev;
383
384     /* Is vif busy ? */
385     if (VIF_EXISTS(vifi))
386         return -EADDRINUSE;
387
388     switch (vifc->vifc_flags) {
389 #ifdef CONFIG_IP_PIMSM
390     case VIFF_REGISTER:
391         /*
392          * Special Purpose VIF in PIM
393          * All the packets will be sent to the daemon
394          */
395         if (reg_vif_num >= 0)
396             return -EADDRINUSE;
397         dev = ipmr_reg_vif(vifc);
398         if (!dev)
399             return -ENOBUFS;
400         break;
401 #endif
402     case VIFF_TUNNEL:
403         dev = ipmr_new_tunnel(vifc);
404         if (!dev)
405             return -ENOBUFS;
406         break;
407     case 0:
408         dev = ip_dev_find(vifc->vifc_lcl_addr.s_addr);
409         if (!dev)
410             return -EADDRNOTAVAIL;
411         _dev_put(dev);
412         break;
413     default:
414         return -EINVAL;
415     }

```

Figure 445: IPMR - Add a VIF: linux/net/ipv4/ipmr.c

```

417     if ((in_dev = __in_dev_get(dev)) == NULL)
418         return -EADDRNOTAVAIL;
419     in_dev->cnf.mc_forwarding++;
420     dev_set_allmulti(dev, +1);
421     ip_rt_multicast_event(in_dev);
422
423     /*
424     *   Fill in the VIF structures
425     */
426     v->rate_limit=vifc->vifc_rate_limit;
427     v->local=vifc->vifc_lcl_addr.s_addr;
428     v->remote=vifc->vifc_rmt_addr.s_addr;
429     v->flags=vifc->vifc_flags;
430     if (!mrtsock)
431         v->flags |= VIFF_STATIC;
432     v->threshold=vifc->vifc_threshold;
433     v->bytes_in = 0;
434     v->bytes_out = 0;
435     v->pkt_in = 0;
436     v->pkt_out = 0;
437     v->link = dev->ifindex;
438     if (v->flags&(VIFF_TUNNEL|VIFF_REGISTER))
439         v->link = dev->iflink;
440
441     /* And finish update writing critical data */
442     write_lock_bh(&mrt_lock);
443     dev_hold(dev);
444     v->dev=dev;
445 #ifdef CONFIG_IP_PIMSM
446     if (v->flags&VIFF_REGISTER)
447         reg_vif_num = vifi;
448 #endif
449     if (vifi + 1 > maxvif)
450         maxvif = vifi + 1;
451     write_unlock_bh(&mrt_lock);
452     return 0;
453 }

```

Figure 446: IPMR - Add a VIF: linux/net/ipv4/ipmr.c

```

776 static void mroute_clean_tables(struct sock *sk)
777 {
778     int i;
779
780     /*
781      *   Shut down all active vif entries
782      */
783     for(i=0; i<maxvif; i++) {
784         if (!(vif_table[i]. flags&VIFF_STATIC))
785             vif_delete(i);
786     }
787
788     /*
789      *   Wipe the cache
790      */
791     for (i=0;i<MFC_LINES;i++) {
792         struct mfc_cache *c, **cp;
793
794         cp = &mfc_cache_array[i];
795         while ((c = *cp) != NULL) {
796             if (c->mfc_flags&MFC_STATIC) {
797                 cp = &c->next;
798                 continue;
799             }
800             write_lock_bh(&mrt_lock);
801             *cp = c->next;
802             write_unlock_bh(&mrt_lock);
803
804             kmem_cache_free(mrt_cache, c);
805         }
806     }
807
808     if (atomic_read(&cache_resolve_queue_len) != 0) {
809         struct mfc_cache *c;
810
811         spin_lock_bh(&mfc_unres_lock);
812         while (mfc_unres_queue != NULL) {
813             c = mfc_unres_queue;
814             mfc_unres_queue = c->next;
815             spin_unlock_bh(&mfc_unres_lock);
816
817             ipmr_destroy_unres(c);
818
819             spin_lock_bh(&mfc_unres_lock);
820         }
821         spin_unlock_bh(&mfc_unres_lock);
822     }
823 }

```

Figure 447: IPMR - cleanup vifs after socket close: `linux/net/ipv4/ipmr.c`

```

286 /* Destroy an unresolved cache entry, killing queued skbs
287    and reporting error to netlink readers.
288 */
289
290 static void ipmr_destroy_unres(struct mfc_cache *c)
291 {
292     struct sk_buff *skb;
293
294     atomic_dec(&cache_resolve_queue_len);
295
296     while((skb=skb_dequeue(&c->mfc_un.unres.unresolved))) {
297 #ifdef CONFIG_RTNETLINK
298         if (skb->nh.iph->version == 0) {
299             struct nlmsg_hdr *nlh = (struct nlmsg_hdr *)skb_pull(skb, sizeof(
300                 struct ip_hdr));
301             nlh->nlmsg_type = NLMSG_ERROR;
302             nlh->nlmsg_len = NLMSG_LENGTH(sizeof(struct nlmsgerr));
303             skb_trim(skb, nlh->nlmsg_len);
304             ((struct nlmsgerr*)NLMSG_DATA(nlh))->error = -
305                 ETIMEDOUT;
306             netlink_unicast(rtnl, skb, NETLINK_CB(skb).dst_pid,
307                 MSG_DONTWAIT);
308         } else
309 #endif
310         kfree_skb(skb);
311     }
312
313     kmem_cache_free(mrt_cachep, c);
314 }

```

Figure 448: IPMR - Destroy stale cache entry: `linux/net/ipv4/ipmr.c`

```

314 /* Single timer process for all the unresolved queue. */
315
316 void ipmr_expire_process(unsigned long dummy)
317 {
318     unsigned long now;
319     unsigned long expires;
320     struct mfc_cache *c, **cp;
321
322     if (!spin_trylock(&mfc_unres_lock)) {
323         mod_timer(&ipmr_expire_timer, jiffies+HZ/10);
324         return;
325     }
326
327     if (atomic_read(&cache_resolve_queue_len) == 0)
328         goto out;
329
330     now = jiffies;
331     expires = 10*HZ;
332     cp = &mfc_unres_queue;
333
334     while ((c=*cp) != NULL) {
335         long interval = c->mfc_un.unres.expires - now;
336
337         if (interval > 0) {
338             if (interval < expires)
339                 expires = interval;
340             cp = &c->next;
341             continue;
342         }
343
344         *cp = c->next;
345
346         ipmr_destroy_unres(c);
347     }
348
349     if (atomic_read(&cache_resolve_queue_len))
350         mod_timer(&ipmr_expire_timer, jiffies + expires);
351
352 out:
353     spin_unlock(&mfc_unres_lock);
354 }

```

Figure 449: IPMR - Timer process: `linux/net/ipv4/ipmr.c`


```

356 /* Fill oifs list. It is called under write locked mrt_lock. */
357
358 static void ipmr_update_thresholds(struct mfc_cache *cache, unsigned char *ttls)
359 {
360     int vifi;
361
362     cache->mfc_un.res.minvif = MAXVIFS;
363     cache->mfc_un.res.maxvif = 0;
364     memset(cache->mfc_un.res.ttls, 255, MAXVIFS);
365
366     for ( vifi =0; vifi <maxvif; vifi++) {
367         if (VIF_EXISTS(vifi) && ttls[vifi] && ttls[ vifi] < 255) {
368             cache->mfc_un.res.ttls[ vifi] = ttls[ vifi ];
369             if (cache->mfc_un.res.minvif > vifi)
370                 cache->mfc_un.res.minvif = vifi;
371             if (cache->mfc_un.res.maxvif <= vifi)
372                 cache->mfc_un.res.maxvif = vifi + 1;
373         }
374     }
375 }

```

Figure 450: IPMR - update output interface lists: `linux/net/ipv4/ipmr.c`

```

455 static struct mfc_cache *ipmr_cache_find(_u32 origin, _u32 mcastgrp)
456 {
457     int line=MFC_HASH(mcastgrp,origin);
458     struct mfc_cache *c;
459
460     for (c=mfc_cache_array[line]; c; c = c->next) {
461         if (c->mfc_origin==origin && c->mfc_mcastgrp==mcastgrp)
462             break;
463     }
464     return c;
465 }
466
467 /*
468  *   Allocate a multicast cache entry
469  */
470 static struct mfc_cache *ipmr_cache_alloc(void)
471 {
472     struct mfc_cache *c=kmem_cache_alloc(mrt_cache, GFP_KERNEL);
473     if (c==NULL)
474         return NULL;
475     memset(c, 0, sizeof(*c));
476     c->mfc_un.res.minvif = MAXVIFS;
477     return c;
478 }
479
480 static struct mfc_cache *ipmr_cache_alloc_unres(void)
481 {
482     struct mfc_cache *c=kmem_cache_alloc(mrt_cache, GFP_ATOMIC);
483     if (c==NULL)
484         return NULL;
485     memset(c, 0, sizeof(*c));
486     skb_queue_head_init(&c->mfc_un.unres.unresolved);
487     c->mfc_un.unres.expires = jiffies + 10*HZ;
488     return c;
489 }

```

Figure 451: IPMR - allocate cache entries: `linux/net/ipv4/ipmr.c`

```

491  /*
492  *   A cache entry has gone into a resolved state from queued
493  */
494
495  static void ipmr_cache_resolve(struct mfc_cache *uc, struct mfc_cache *c)
496  {
497      struct sk_buff *skb;
498
499      /*
500       *   Play the pending entries through our router
501       */
502
503      while((skb=__skb_dequeue(&uc->mfc_un.unres.unresolved))) {
504  #ifdef CONFIG_RTNETLINK
505          if (skb->nh.iph->version == 0) {
506              int err;
507              struct nlmsg_hdr *nlh = (struct nlmsg_hdr *)skb_pull(skb, sizeof(
508                  struct iphdr));
509
510              if (ipmr_fill_mroute(skb, c, NLMSG_DATA(nlh)) > 0) {
511                  nlh->nmsg_len = skb->tail - (u8*)nlh;
512              } else {
513                  nlh->nmsg_type = NLMSG_ERROR;
514                  nlh->nmsg_len = NLMSG_LENGTH(sizeof(struct
515                      nlmsgerr));
516                  skb_trim(skb, nlh->nmsg_len);
517                  ((struct nlmsgerr*)NLMSG_DATA(nlh))->error = -
518                      EMSGSIZE;
519              }
520              err = netlink_unicast(rtnl, skb, NETLINK_CB(skb).dst_pid,
521                  MSG_DONTWAIT);
522          } else
523              ip_mr_forward(skb, c, 0);
524      }
525  #endif
526  }

```

Figure 452: IPMR - resolve cache entries: `linux/net/ipv4/ipmr.c`

```

524  /*
525  *   Bounce a cache query up to mrouterd. We could use netlink for this but
526  *   mrouterd
527  *   expects the following bizarre scheme.
528  *   Called under mrt_lock.
529  */
530
531  static int ipmr_cache_report(struct sk_buff *pkt, vifi_t vifi, int assert)
532  {
533      struct sk_buff *skb;
534      int ihl = pkt->nh.iph->ihl<<2;
535      struct igmp_hdr *igmp;
536      struct igmpmsg *msg;
537      int ret;
538
539  #ifdef CONFIG_IP_PIMSM
540      if (assert == IGMPMSG_WHOLEPKT)
541          skb = skb_realloc_headroom(pkt, sizeof(struct iphdr));
542      else
543  #endif
544          skb = alloc_skb(128, GFP_ATOMIC);
545
546      if (!skb)
547          return -ENOBUFS;
548
549  #ifdef CONFIG_IP_PIMSM
550      if (assert == IGMPMSG_WHOLEPKT) {
551          /* Ugly, but we have no choice with this interface.
552             Duplicate old header, fix ihl, length etc.
553             And all this only to mangle msg->im_msgtype and
554             to set msg->im_mbz to "mbz" :-)
555          */
556          msg = (struct igmpmsg*)skb_push(skb, sizeof(struct iphdr));
557          skb->nh.raw = skb->h.raw = (u8*)msg;
558          memcpy(msg, pkt->nh.raw, sizeof(struct iphdr));
559          msg->im_msgtype = IGMPMSG_WHOLEPKT;
560          msg->im_mbz = 0;
561          msg->im_vif = reg_vif_num;
562          skb->nh.iph->ihl = sizeof(struct iphdr) >> 2;
563          skb->nh.iph->tot_len = htons(ntohs(pkt->nh.iph->tot_len) + sizeof(
564              struct iphdr));
565      } else
566  #endif

```

Figure 453: IPMR - multicast route cache query passed up to mrouterd:
linux/net/ipv4/ipmr.c

```

566     {
567
568     /*
569     *   Copy the IP header
570     */
571
572     skb->nh.iph = (struct iphdr *)skb_put(skb, ihl);
573     memcpy(skb->data, pkt->data, ihl);
574     skb->nh.iph->protocol = 0;          /* Flag to the kernel this is
575     a route add */
576     msg = (struct igmpmsg*)skb->nh.iph;
577     msg->im_vif = vif;
578     skb->dst = dst_clone(pkt->dst);
579
580     /*
581     *   Add our header
582     */
583     igmp=(struct igmphdr *)skb_put(skb, sizeof(struct igmphdr));
584     igmp->type =
585     msg->im_msgtype = assert;
586     igmp->code = 0;
587     skb->nh.iph->tot_len=htons(skb->len);    /* Fix the length */
588     skb->h.raw = skb->nh.raw;
589     }
590
591     if (mroute_socket == NULL) {
592         kfree_skb(skb);
593         return -EINVAL;
594     }
595
596     /*
597     *   Deliver to mrouterd
598     */
599     if ((ret=sock_queue_rcv_skb(mroute_socket,skb))<0) {
600         if ( net_ratelimit ()
601             printk(KERN_WARNING "mroute: pending queue full, dropping
602             entries.\n");
603             kfree_skb(skb);
604         }
605     return ret;
606 }

```

Figure 454: IPMR - multicast route cache query passed up to mrouterd:
linux/net/ipv4/ipmr.c

```

612 static int
613 ipmr_cache_unresolved(vifi_t vifi, struct sk_buff *skb)
614 {
615     int err;
616     struct mfc_cache *c;
617
618     spin_lock_bh(&mfc_unres_lock);
619     for (c=mfc_unres_queue; c; c=c->next) {
620         if (c->mfc_mcastgrp == skb->nh.iph->daddr &&
621             c->mfc_origin == skb->nh.iph->saddr)
622             break;
623     }
624
625     if (c == NULL) {
626         /*
627          *   Create a new entry if allowable
628          */
629
630         if (atomic_read(&cache_resolve_queue_len) >= 10 ||
631             (c=ipmr_cache_alloc_unres()) == NULL) {
632             spin_unlock_bh(&mfc_unres_lock);
633
634             kfree_skb(skb);
635             return -ENOBUFS;
636         }
637
638         /*
639          *   Fill in the new cache entry
640          */
641         c->mfc_parent = -1;
642         c->mfc_origin = skb->nh.iph->saddr;
643         c->mfc_mcastgrp = skb->nh.iph->daddr;

```

Figure 455: IPMR - queue a packet to send awaiting resolution:
linux/net/ipv4/ipmr.c

```

645     /*
646     *       Reflect first query at mouted.
647     */
648     if ((err = ipmr_cache_report(skb, vifi, IGMPMSG_NOCACHE)) < 0) {
649         /* If the report failed throw the cache entry
650          * out - Brad Parker
651          */
652         spin_unlock_bh(&mfc_unres_lock);
653
654         kmem_cache_free(mrt_cache, c);
655         kfree_skb(skb);
656         return err;
657     }
658
659     atomic_inc(&cache_resolve_queue_len);
660     c->next = mfc_unres_queue;
661     mfc_unres_queue = c;
662
663     mod_timer(&ipmr_expire_timer, c->mfc_un.unres.expires);
664 }

```

Figure 456: IPMR - queue a packet to send awaiting resolution:
linux/net/ipv4/ipmr.c

```

666     /*
667     *      See if we can append the packet
668     */
669     if (c->mfc_un.unres.unresolved.qlen>3) {
670         kfree_skb(skb);
671         err = -ENOBUFS;
672     } else {
673         skb_queue_tail(&c->mfc_un.unres.unresolved,skb);
674         err = 0;
675     }
676     spin_unlock_bh(&mfc_unres_lock);
677     return err;
678 }
679

```

Figure 457: IPMR - queue a packet to send awaiting resolution:
linux/net/ipv4/ipmr.c


```

685 int ipmr_mfc_delete(struct mfcctl *mfc)
686 {
687     int line;
688     struct mfc_cache *c, **cp;
689
690     line=MFC_HASH(mfc->mfcc_mcastgrp.s_addr, mfc->mfcc_origin.s_addr);
691
692     for (cp=&mfc_cache_array[line]; (c=*cp) != NULL; cp = &c->next) {
693         if (c->mfc_origin == mfc->mfcc_origin.s_addr &&
694             c->mfc_mcastgrp == mfc->mfcc_mcastgrp.s_addr) {
695             write_lock_bh(&mrt_lock);
696             *cp = c->next;
697             write_unlock_bh(&mrt_lock);
698
699             kmem_cache_free(mrt_cache, c);
700             return 0;
701         }
702     }
703     return -ENOENT;
704 }

```

Figure 458: IPMR - user space calls MFC delete: `linux/net/ipv4/ipmr.c`

```

706 int ipmr_mfc_add(struct mfcctl *mfc, int mrtsock)
707 {
708     int line;
709     struct mfc_cache *uc, *c, **cp;
710
711     line=MFC_HASH(mfc->mfcc_mcastgrp.s_addr, mfc->mfcc_origin.s_addr);
712
713     for (cp=&mfc_cache_array[line]; (c=*cp) != NULL; cp = &c->next) {
714         if (c->mfc_origin == mfc->mfcc_origin.s_addr &&
715             c->mfc_mcastgrp == mfc->mfcc_mcastgrp.s_addr)
716             break;
717     }
718
719     if (c != NULL) {
720         write_lock_bh(&mrt_lock);
721         c->mfc_parent = mfc->mfcc_parent;
722         ipmr_update_thresholds(c, mfc->mfcc_ttls);
723         if (!mrtsock)
724             c->mfc_flags |= MFC_STATIC;
725         write_unlock_bh(&mrt_lock);
726         return 0;
727     }
728
729     if (!MULTICAST(mfc->mfcc_mcastgrp.s_addr))
730         return -EINVAL;
731
732     c=ipmr_cache_alloc();
733     if (c==NULL)
734         return -ENOMEM;
735
736     c->mfc_origin=mfc->mfcc_origin.s_addr;
737     c->mfc_mcastgrp=mfc->mfcc_mcastgrp.s_addr;
738     c->mfc_parent=mfc->mfcc_parent;
739     ipmr_update_thresholds(c, mfc->mfcc_ttls);
740     if (!mrtsock)
741         c->mfc_flags |= MFC_STATIC;
742
743     write_lock_bh(&mrt_lock);
744     c->next = mfc_cache_array[line];
745     mfc_cache_array[line] = c;
746     write_unlock_bh(&mrt_lock);

```

Figure 459: IPMR - user space calls MFC add: `linux/net/ipv4/ipmr.c`

```

748     /*
749     *      Check to see if we resolved a queued list. If so we
750     *      need to send on the frames and tidy up.
751     */
752     spin_lock_bh(&mfc_unres_lock);
753     for (cp = &mfc_unres_queue; (uc=*cp) != NULL;
754          cp = &uc->next) {
755         if (uc->mfc_origin == c->mfc_origin &&
756             uc->mfc_mcastgrp == c->mfc_mcastgrp) {
757             *cp = uc->next;
758             if (atomic_dec_and_test(&cache_resolve_queue_len))
759                 del_timer(&ipmr_expire_timer);
760             break;
761         }
762     }
763     spin_unlock_bh(&mfc_unres_lock);
764
765     if (uc) {
766         ipmr_cache_resolve(uc, c);
767         kmem_cache_free(mrt_cache, uc);
768     }
769     return 0;
770 }

```

Figure 460: IPMR - user space calls MFC add: `linux/net/ipv4/ipmr.c`

```
825 static void mrtsock_destruct(struct sock *sk)
826 {
827     rtnl_lock();
828     if (sk == mroute_socket) {
829         ipv4_devconf.mc_forwarding--;
830
831         write_lock_bh(&mrt_lock);
832         mroute_socket=NULL;
833         write_unlock_bh(&mrt_lock);
834
835         mroute_clean_tables(sk);
836     }
837     rtnl_unlock();
838 }
```

Figure 461: IPMR - mrouterd socket dies: `linux/net/ipv4/ipmr.c`

```

840  /*
841  *      Socket options and virtual interface manipulation. The whole
842  *      virtual interface system is a complete heap, but unfortunately
843  *      that's how BSD mouted happens to think. Maybe one day with a proper
844  *      MOSPF/PIM router set up we can clean this up.
845  */
846
847  int ip_mroute_setsockopt(struct sock *sk, int optname, char *optval, int optlen)
848  {
849      int ret;
850      struct vifctl vif;
851      struct mfctl mfc;
852
853      if (optname != MRT_INIT)
854      {
855          if (sk != mroute_socket && !capable(CAP_NET_ADMIN))
856              return -EACCES;
857      }
858
859      switch(optname)
860      {
861          case MRT_INIT:
862              if (sk->type != SOCK_RAW || sk->num != IPPROTO_IGMP)
863                  return -EOPNOTSUPP;
864              if (optlen != sizeof(int))
865                  return -ENOPROTOOPT;
866
867              rtnl_lock();
868              if (mroute_socket) {
869                  rtnl_unlock();
870                  return -EADDRINUSE;
871              }
872
873              ret = ip_ra_control(sk, 1, mrtsock_destruct);
874              if (ret == 0) {
875                  write_lock_bh(&mrt_lock);
876                  mroute_socket = sk;
877                  write_unlock_bh(&mrt_lock);
878
879                  ipv4_devconf.mc_forwarding++;
880              }
881              rtnl_unlock();
882              return ret;
883          case MRT_DONE:
884              if (sk != mroute_socket)
885                  return -EACCES;
886              return ip_ra_control(sk, 0, NULL);

```

Figure 462: IPMR - setsockopt entry handler: `linux/net/ipv4/ipmr.c`

```

887     case MRT_ADD_VIF:
888     case MRT_DEL_VIF:
889         if (optlen != sizeof(vif))
890             return -EINVAL;
891         if (copy_from_user(&vif, optval, sizeof(vif)))
892             return -EFAULT;
893         if (vif.vifc_vifi >= MAXVIFS)
894             return -ENFILE;
895         rtnl_lock();
896         if (optname == MRT_ADD_VIF) {
897             ret = vif_add(&vif, sk == mroute_socket);
898         } else {
899             ret = vif_delete(vif.vifc_vifi);
900         }
901         rtnl_unlock();
902         return ret;
903
904     /*
905     *   Manipulate the forwarding caches. These live
906     *   in a sort of kernel/user symbiosis.
907     */
908     case MRT_ADD_MFC:
909     case MRT_DEL_MFC:
910         if (optlen != sizeof(mfc))
911             return -EINVAL;
912         if (copy_from_user(&mfc, optval, sizeof(mfc)))
913             return -EFAULT;
914         rtnl_lock();
915         if (optname == MRT_DEL_MFC)
916             ret = ipmr_mfc_delete(&mfc);
917         else
918             ret = ipmr_mfc_add(&mfc, sk == mroute_socket);
919         rtnl_unlock();
920         return ret;
921
922     /*
923     *   Control PIM assert.
924     */
925     case MRT_ASSERT:
926     {
927         int v;
928         if (get_user(v, (int *)optval))
929             return -EFAULT;
930         mroute_do_assert = (v)?1:0;
931         return 0;
932     }

```

Figure 463: IPMR - setsockopt entry handler: linux/net/ipv4/ipmr.c

```

932 #ifdef CONFIG_IP_PIMSM
933     case MRT_PIM:
934     {
935         int v;
936         if (get_user(v,(int *)optval))
937             return -EFAULT;
938         v = (v)?1:0;
939         rtnl_lock ();
940         if (v != mroute_do_pim) {
941             mroute_do_pim = v;
942             mroute_do_assert = v;
943 #ifdef CONFIG_IP_PIMSM_V2
944             if (mroute_do_pim)
945                 inet_add_protocol(&pim_protocol);
946             else
947                 inet_del_protocol (&pim_protocol);
948 #endif
949         }
950         rtnl_unlock();
951         return 0;
952     }
953 #endif
954     /*
955     *   Spurious command, or MRT_VERSION which you cannot
956     *   set.
957     */
958     default :
959         return -ENOPROTOPT;
960 }
961 }

```

Figure 464: IPMR - setsockopt entry handler: `linux/net/ipv4/ipmr.c`

```

967 int ip_mroute_getsockopt (struct sock *sk, int optname, char *optval, int *optlen)
968 {
969     int olr;
970     int val;
971
972     if (optname != MRT_VERSION &&
973 #ifdef CONFIG_IP_PIMSM
974         optname != MRT_PIM &&
975 #endif
976         optname != MRT_ASSERT)
977         return -ENOPROTOPT;
978
979     if (get_user(olr, optlen))
980         return -EFAULT;
981
982     olr = min(olr, sizeof(int));
983     if (olr < 0)
984         return -EINVAL;
985
986     if (put_user(olr, optlen))
987         return -EFAULT;
988     if (optname == MRT_VERSION)
989         val = 0x0305;
990 #ifdef CONFIG_IP_PIMSM
991     else if (optname == MRT_PIM)
992         val = mroute_do_pim;
993 #endif
994     else
995         val = mroute_do_assert;
996     if (copy_to_user(optval, &val, olr))

```

Figure 465: IPMR - getsockopt entry handler: `linux/net/ipv4/ipmr.c`


```

1002  *   The IP multicast ioctl support routines.
1003  */
1004
1005  int ipmr_ioctl(struct sock *sk, int cmd, unsigned long arg)
1006  {
1007      struct sioc_sg_req sr;
1008      struct sioc_vif_req vr;
1009      struct vif_device *vif;
1010      struct mfc_cache *c;
1011
1012      switch(cmd)
1013      {
1014          case SIOCGETVIFCNT:
1015              if (copy_from_user(&vr,(void *)arg, sizeof (vr)))
1016                  return -EFAULT;
1017              if (vr.vifi >= maxvif)
1018                  return -EINVAL;
1019              read_lock(&mrt_lock);
1020              vif=&vif_table[vr.vifi];
1021              if (VIF_EXISTS(vr.vifi)) {
1022                  vr.icount=vif->pkt_in;
1023                  vr.ocount=vif->pkt_out;
1024                  vr.abytes=vif->bytes_in;
1025                  vr.obytes=vif->bytes_out;
1026                  read_unlock(&mrt_lock);
1027
1028                  if (copy_to_user((void *)arg,&vr, sizeof (vr)))
1029                      return -EFAULT;
1030                  return 0;
1031              }
1032              read_unlock(&mrt_lock);
1033              return -EADDRNOTAVAIL;
1034          case SIOCGETSGCNT:
1035              if (copy_from_user(&sr,(void *)arg, sizeof (sr)))
1036                  return -EFAULT;
1037
1038              read_lock(&mrt_lock);
1039              c = ipmr_cache_find(sr.src.s_addr, sr.grp.s_addr);
1040              if (c) {
1041                  sr.pktcnt = c->mfc_un.res.pkt;
1042                  sr.bytecnt = c->mfc_un.res.bytes;
1043                  sr.wrong_if = c->mfc_un.res.wrong_if;
1044                  read_unlock(&mrt_lock);
1045
1046                  if (copy_to_user((void *)arg,&sr, sizeof (sr)))
1047                      return -EFAULT;
1048                  return 0;
1049              }
1050              read_unlock(&mrt_lock);
1051              return -EADDRNOTAVAIL;
1052          default :

```

Figure 466: IPMR - : linux/net/ipv4/ipmr.c

```

1076         0
1077     };
1078
1079     /*
1080     *   Encapsulate a packet by attaching a valid IPIP header to it.
1081     *   This avoids tunnel drivers and other mess and gives us the speed so
1082     *   important for multicast video.
1083     */
1084
1085     static void ip_encap(struct sk_buff *skb, u32 saddr, u32 daddr)
1086     {
1087         struct iphdr *iph = (struct iphdr *)skb_push(skb, sizeof(struct iphdr));
1088
1089         iph->version = 4;
1090         iph->tos = skb->nh.iph->tos;
1091         iph->ttl = skb->nh.iph->ttl;
1092         iph->frag_off = 0;
1093         iph->daddr = daddr;
1094         iph->saddr = saddr;
1095         iph->protocol = IPPROTO_IPIP;
1096         iph->ihl = 5;
1097         iph->tot_len = htons(skb->len);
1098         ip_select_ident(iph, skb->dst, NULL);
1099         ip_send_check(iph);
1100
1101         skb->h.ipiph = skb->nh.iph;
1102         skb->nh.iph = iph;
1103     #ifdef CONFIG_NETFILTER
1104         nf_contrack_put(skb->nfct);

```

Figure 467: IPMR - transmit encapsualted: `linux/net/ipv4/ipmr.c`

```
1106 #endif
1107 }
1108
1109 static inline int ipmr_forward_finish(struct sk_buff *skb)
1110 {
1111     struct dst_entry *dst = skb->dst;
1112
1113     if (skb->len <= dst->pmtu)
1114         return dst->output(skb);
```

Figure 468: IPMR - finish forwarding: `linux/net/ipv4/ipmr.c`

```

1116         return ip_fragment(skb, dst->output);
1117     }
1118
1119     /*
1120     *   Processing handlers for ipmr_forward
1121     */
1122
1123     static void ipmr_queue_xmit(struct sk_buff *skb, struct mfc_cache *c,
1124                               int vifi, int last)
1125     {
1126         struct iphdr *iph = skb->nh.iph;
1127         struct vif_device *vif = &vif_table[vifi];
1128         struct net_device *dev;
1129         struct rtable *rt;
1130         int encap = 0;
1131         struct sk_buff *skb2;
1132
1133         if (vif->dev == NULL)
1134             return;
1135
1136     #ifdef CONFIG_IP_PIMSM
1137         if (vif->flags & VIFF_REGISTER) {
1138             vif->pkt_out++;
1139             vif->bytes_out += skb->len;
1140             ((struct net_device_stats *)vif->dev->priv)->tx_bytes += skb->len;
1141             ((struct net_device_stats *)vif->dev->priv)->tx_packets++;
1142             ipmr_cache_report(skb, vifi, IGMPMSG_WHOLEPKT);

```

Figure 469: IPMR - transmission handler: `linux/net/ipv4/ipmr.c`

```

1144     }
1145 #endif
1146
1147     if (vif->flags&VIFF_TUNNEL) {
1148         if (ip_route_output(&rt, vif->remote, vif->local, RT_TOS(iph->tos),
1149                             vif->link))
1149             return;
1150         encap = sizeof(struct iphdr);
1151     } else {
1152         if (ip_route_output(&rt, iph->daddr, 0, RT_TOS(iph->tos), vif->link))
1153             return;
1154     }
1155
1156     dev = rt->u.dst.dev;
1157
1158     if (skb->len+encap > rt->u.dst.pmtu && (ntohs(iph->frag_off) & IP_DF)) {
1159         /* Do not fragment multicasts. Alas, IPv4 does not
1160          allow to send ICMP, so that packets will disappear
1161          to blackhole.
1162          */
1163
1164         IP_INC_STATS_BH(IpFragFails);
1165         ip_rt_put(rt);
1166         return;
1167     }
1168
1169     encap += dev->hard_header_len;
1170
1171     if (skb_headroom(skb) < encap || skb_cloned(skb) || !last)
1172         skb2 = skb_realloc_headroom(skb, (encap + 15)&~15);
1173     else if (atomic_read(&skb->users) != 1)
1174         skb2 = skb_clone(skb, GFP_ATOMIC);
1175     else {
1176         atomic_inc(&skb->users);
1177         skb2 = skb;
1178     }
1179
1180     if (skb2 == NULL) {

```

Figure 470: IPMR - transmission handler: `linux/net/ipv4/ipmr.c`

```

1182         return;
1183     }
1184
1185     vif->pkt_out++;
1186     vif->bytes_out+=skb->len;
1187
1188     dst_release(skb2->dst);
1189     skb2->dst = &rt->u.dst;
1190     iph = skb2->nh.iph;
1191     ip_decrease_ttl(iph);
1192
1193     /* FIXME: forward and output firewalls used to be called here.
1194     * What do we do with netfilter? -- RR */
1195     if (vif->flags & VIFF_TUNNEL) {
1196         ip_encap(skb2, vif->local, vif->remote);
1197         /* FIXME: extra output firewall step used to be here. --RR */
1198         ((struct ip_tunnel *)vif->dev->priv)->stat.tx_packets++;
1199         ((struct ip_tunnel *)vif->dev->priv)->stat.tx_bytes+=skb2->len;
1200     }
1201
1202     IPCB(skb2)->flags |= IPSKB_FORWARDED;
1203
1204     /*
1205     * RFC1584 teaches, that DVMRP/PIM router must deliver packets locally
1206     * not only before forwarding, but after forwarding on all output
1207     * interfaces. It is clear, if mrouter runs a multicasting
1208     * program, it should receive packets not depending to what interface
1209     * program is joined.
1210     * If we will not make it, the program will have to join on all
1211     * interfaces. On the other hand, multihoming host (or router, but
1212     * not mrouter) cannot join to more than one interface - it will
1213     * result in receiving multiple packets.
1214     */

```

Figure 471: IPMR - transmission handler: linux/net/ipv4/ipmr.c

```
1216         ipmr_forward_finish);
1217     }
1218
1219     int ipmr_find_vif(struct net_device *dev)
1220     {
1221         int ct;
1222         for (ct=maxvif-1; ct>=0; ct--) {
1223             if (vif_table[ct].dev == dev)
1224                 break;
```

Figure 472: IPMR - find a vif by device: `linux/net/ipv4/ipmr.c`

```

1226     return ct;
1227 }
1228
1229 /* "local" means that we should preserve one skb (for local delivery) */
1230
1231 int ip_mr_forward(struct sk_buff *skb, struct mfc_cache *cache, int local)
1232 {
1233     int psend = -1;
1234     int vif, ct;
1235
1236     vif = cache->mfc_parent;
1237     cache->mfc_un.res.pkt++;
1238     cache->mfc_un.res.bytes += skb->len;
1239
1240     /*
1241      * Wrong interface: drop packet and (maybe) send PIM assert.
1242      */
1243     if ( vif_table [ vif ].dev != skb->dev) {
1244         int true_vifi;
1245
1246         if (((struct rtable*)skb->dst)->key.iif == 0) {
1247             /* It is our own packet, looped back.
1248              * Very complicated situation...
1249
1250              * The best workaround until routing daemons will be
1251              * fixed is not to redistribute packet, if it was
1252              * send through wrong interface. It means, that
1253              * multicast applications WILL NOT work for
1254              * (S,G), which have default multicast route pointing
1255              * to wrong oif. In any case, it is not a good
1256              * idea to use multicasting applications on router.
1257              */
1258             goto dont_forward;
1259         }
1260
1261         cache->mfc_un.res.wrong_if++;
1262         true_vifi = ipmr_find_vif(skb->dev);
1263
1264         if ( true_vifi >= 0 && mroute_do_assert &&
1265             /* pimsm uses asserts, when switching from RPT to SPT,
1266              * so that we cannot check that packet arrived on an oif.
1267              * It is bad, but otherwise we would need to move pretty
1268              * large chunk of pimd to kernel. Ough... --ANK
1269              */
1270             (mroute_do_pim || cache->mfc_un.res.ttls[true_vifi] < 255) &&
1271             jiffies - cache->mfc_un.res.last_assert > MFC_ASSERT_THRESH) {
1272             cache->mfc_un.res.last_assert = jiffies;
1273             ipmr_cache_report(skb, true_vifi, IGMPMSG_WRONGVIF);

```

Figure 473: IPMR - forwarding: linux/net/ipv4/ipmr.c


```

1275         goto dont_forward;
1276     }
1277
1278     vif_table[vif].pkt_in++;
1279     vif_table[vif].bytes_in += skb->len;
1280
1281     /*
1282     *   Forward the frame
1283     */
1284     for (ct = cache->mfc_un.res.maxvif-1; ct >= cache->mfc_un.res.minvif; ct--)
1285     {
1286         if (skb->nh.iph->ttl > cache->mfc_un.res.ttls[ct]) {
1287             if (psend != -1)
1288                 ipmr_queue_xmit(skb, cache, psend, 0);
1289             psend=ct;
1290         }
1291     }
1292     if (psend != -1)
1293         ipmr_queue_xmit(skb, cache, psend, !local);
1294 dont_forward:
1295     if (!local)

```

Figure 474: IPMR - forwarding: `linux/net/ipv4/ipmr.c`

```

1298 }
1299
1300
1301 /*
1302 *   Multicast packets for forwarding arrive here
1303 */
1304
1305 int ip_mr_input(struct sk_buff *skb)
1306 {
1307     struct mfc_cache *cache;
1308     int local = ((struct rtable*)skb->dst)->rt_flags&RTCF_LOCAL;
1309
1310     /* Packet is looped back after forward, it should not be
1311        forwarded second time, but still can be delivered locally.
1312     */
1313     if (IPCB(skb)->flags&IPSKB_FORWARDED)
1314         goto dont_forward;
1315
1316     if (!local) {
1317         if (IPCB(skb)->opt.router_alert) {
1318             if ( ip_call_ra_chain (skb))
1319                 return 0;
1320         } else if (skb->nh.iph->protocol == IPPROTO_IGMP){
1321             /* IGMPv1 (and broken IGMPv2 implementations sort of
1322                Cisco IOS <= 11.2(8)) do not put router alert
1323                option to IGMP packets destined to routable
1324                groups. It is very bad, because it means
1325                that we can forward NO IGMP messages.
1326             */
1327             read_lock(&mrt_lock);
1328             if (mroute_socket) {
1329                 raw_rcv(mroute_socket, skb);
1330                 read_unlock(&mrt_lock);
1331                 return 0;
1332             }

```

Figure 475: IPMR - handle multicast packets to be forwarded:
linux/net/ipv4/ipmr.c

```

1334     }
1335 }
1336
1337 read_lock(&mrt_lock);
1338 cache = ipmr_cache_find(skb->nh.iph->saddr, skb->nh.iph->daddr);
1339
1340 /*
1341  *   No usable cache entry
1342  */
1343 if (cache==NULL) {
1344     int vif;
1345
1346     if (local) {
1347         struct sk_buff *skb2 = skb_clone(skb, GFP_ATOMIC);
1348         ip_local_deliver(skb);
1349         if (skb2 == NULL) {
1350             read_unlock(&mrt_lock);
1351             return -ENOBUFS;
1352         }
1353         skb = skb2;
1354     }
1355
1356     vif = ipmr_find_vif(skb->dev);
1357     if (vif >= 0) {
1358         int err = ipmr_cache_unresolved(vif, skb);
1359         read_unlock(&mrt_lock);
1360
1361         return err;
1362     }
1363     read_unlock(&mrt_lock);
1364     kfree_skb(skb);
1365     return -ENODEV;
1366 }
1367
1368 ip_mr_forward(skb, cache, local);
1369
1370 read_unlock(&mrt_lock);
1371
1372 if (local)
1373     return ip_local_deliver(skb);
1374
1375 return 0;
1376
1377 dont_forward:
1378     if (local)
1379         return ip_local_deliver(skb);

```

Figure 476: IPMR - handle multicast packets to be forwarded:
linux/net/ipv4/ipmr.c

```

1381     return 0;
1382 }
1383
1384 #ifdef CONFIG_IP_PIMSM_V1
1385 /*
1386  * Handle IGMP messages of PIMv1
1387  */
1388
1389 int pim_rcv_v1(struct sk_buff * skb)
1390 {
1391     struct igmp_hdr *pim = (struct igmp_hdr*)skb->h.raw;
1392     struct ip_hdr *encap;
1393     struct net_device *reg_dev = NULL;
1394
1395     if (skb_is_nonlinear(skb)) {
1396         if (skb_linearize(skb, GFP_ATOMIC) != 0) {
1397             kfree_skb(skb);
1398             return -ENOMEM;
1399         }
1400         pim = (struct igmp_hdr*)skb->h.raw;
1401     }
1402
1403     if (!mroute_do_pim ||
1404         skb->len < sizeof(*pim) + sizeof(*encap) ||
1405         pim->group != PIM_V1_VERSION || pim->code != PIM_V1_REGISTER) {
1406         kfree_skb(skb);
1407         return -EINVAL;
1408     }
1409
1410     encap = (struct ip_hdr*)(skb->h.raw + sizeof(struct igmp_hdr));
1411     /*

```

Figure 477: IPMR - PIM v1 input handler: linux/net/ipv4/ipmr.c

```

1413     a. packet is really destined to a multicast group
1414     b. packet is not a NULL-REGISTER
1415     c. packet is not truncated
1416     */
1417     if (!MULTICAST(encap->daddr) ||
1418         ntohs(encap->tot_len) == 0 ||
1419         ntohs(encap->tot_len) + sizeof(*pim) > skb->len) {
1420         kfree_skb(skb);
1421         return -EINVAL;
1422     }
1423
1424     read_lock(&mrt_lock);
1425     if (reg_vif_num >= 0)
1426         reg_dev = vif_table[reg_vif_num].dev;
1427     if (reg_dev)
1428         dev_hold(reg_dev);
1429     read_unlock(&mrt_lock);
1430
1431     if (reg_dev == NULL) {
1432         kfree_skb(skb);
1433         return -EINVAL;
1434     }
1435
1436     skb->mac.raw = skb->nh.raw;
1437     skb_pull(skb, (u8*)encap - skb->data);
1438     skb->nh.iph = (struct iphdr *)skb->data;
1439     skb->dev = reg_dev;
1440     memset(&(IPCB(skb)->opt), 0, sizeof(struct ip_options));
1441     skb->protocol = _constant_htons(ETH_P_IP);
1442     skb->ip_summed = 0;
1443     skb->pkt_type = PACKET_HOST;
1444     dst_release(skb->dst);

```

Figure 478: IPMR - PIM v1 input handler: `linux/net/ipv4/ipmr.c`

```

1447     ((struct net_device_stats *)reg_dev->priv)->rx_packets++;
1448 #ifdef CONFIG_NETFILTER
1449     nf_conntrack_put(skb->nfct);
1450     skb->nfct = NULL;
1451 #endif
1452     netif_rx(skb);
1453     dev_put(reg_dev);
1454     return 0;
1455 }
1456 #endif
1457
1458 #ifdef CONFIG_IP_PIMSM_V2
1459 int pim_rcv(struct sk_buff *skb)
1460 {
1461     struct pimreghdr *pim = (struct pimreghdr*)skb->h.raw;
1462     struct iphdr *encap;
1463     struct net_device *reg_dev = NULL;
1464
1465     if (skb_is_nonlinear(skb)) {
1466         if (skb_linearize(skb, GFP_ATOMIC) != 0) {
1467             kfree_skb(skb);
1468             return -ENOMEM;
1469         }
1470     }
1471     pim = (struct pimreghdr*)skb->h.raw;

```

Figure 479: IPMR - PIM v2 input handler: `linux/net/ipv4/ipmr.c`

```

1472
1473     if (skb->len < sizeof(*pim) + sizeof(*encap) ||
1474         pim->type != ((PIM_VERSION<<4)|(PIM_REGISTER)) ||
1475         (pim->flags&PIM_NULL_REGISTER) ||
1476         (ip_compute_csum((void *)pim, sizeof(*pim)) != 0 &&
1477         ip_compute_csum((void *)pim, skb->len))) {
1478         kfree_skb(skb);
1479         return -EINVAL;
1480     }
1481
1482     /* check if the inner packet is destined to mcast group */
1483     encap = (struct iphdr*)(skb->h.raw + sizeof(struct pimreghdr));
1484     if (!MULTICAST(encap->daddr) ||
1485         ntohs(encap->tot_len) == 0 ||
1486         ntohs(encap->tot_len) + sizeof(*pim) > skb->len) {
1487         kfree_skb(skb);
1488         return -EINVAL;
1489     }
1490
1491     read_lock(&mrt_lock);
1492     if (reg_vif_num >= 0)
1493         reg_dev = vif_table[reg_vif_num].dev;
1494     if (reg_dev)
1495         dev_hold(reg_dev);
1496     read_unlock(&mrt_lock);
1497
1498     if (reg_dev == NULL) {
1499         kfree_skb(skb);
1500         return -EINVAL;
1501     }
1502
1503     skb->mac.raw = skb->nh.raw;

```

Figure 480: IPMR - PIM v2 input handler: `linux/net/ipv4/ipmr.c`

```

1508     skb->protocol = _constant_htons(ETH_P_IP);
1509     skb->ip_summed = 0;
1510     skb->pkt_type = PACKET_HOST;
1511     dst_release(skb->dst);
1512     ((struct net_device_stats *)reg_dev->priv)->rx_bytes += skb->len;
1513     ((struct net_device_stats *)reg_dev->priv)->rx_packets++;
1514     skb->dst = NULL;
1515     #ifdef CONFIG_NETFILTER
1516     nf_conntrack_put(skb->nfct);
1517     skb->nfct = NULL;
1518     #endif
1519     netif_rx(skb);
1520     dev_put(reg_dev);
1521     return 0;
1522 }
1523 #endif
1524
1525 #ifdef CONFIG_RTNETLINK
1526
1527 static int
1528 ipmr_fill_mroute(struct sk_buff *skb, struct mfc_cache *c, struct rtmsg *rtm)
1529 {
1530     int ct;
1531     struct rt nexthop *nhp;
1532     struct net_device *dev = vif_table[c->mfc_parent].dev;
1533     u8 *b = skb->tail;
1534     struct rtattr *mp_head;
1535
1536     if (dev)
1537         RTA_PUT(skb, RTA_IIF, 4, &dev->ifindex);
1538
1539     mp_head = (struct rtattr *)skb_put(skb, RTA_LENGTH(0));
1540
1541     for (ct = c->mfc_un.res.minvif; ct < c->mfc_un.res.maxvif; ct++) {

```

Figure 481: IPMR - RT Netlink fill route: `linux/net/ipv4/ipmr.c`


```

1543         if (skb_tailroom(skb) < RTA_ALIGN(RTA_ALIGN(sizeof(*nhp))
1544             + 4))
1545             goto rtattr_failure ;
1546         nhp = (struct rtnexthop*)skb_put(skb, RTA_ALIGN(sizeof(*nhp)))
1547             ;
1548         nhp->rtnh_flags = 0;
1549         nhp->rtnh_hops = c->mfc_un.res.ttls[ct];
1550         nhp->rtnh_ifindex = vif_table[ct].dev->ifindex;
1551         nhp->rtnh_len = sizeof(*nhp);
1552     }
1553     mp_head->rta_type = RTA_MULTIPATH;
1554     mp_head->rta_len = skb->tail - (u8*)mp_head;
1555     rtm->rtm_type = RTN_MULTICAST;
1556     return 1;
1557 rtattr_failure :
1558     skb_trim(skb, b - skb->data);
1559     return -EMSGSIZE;
1560 }
1561
1562 int ipmr_get_route(struct sk_buff *skb, struct rtmmsg *rtm, int nowait)
1563 {
1564     int err;
1565     struct mfc_cache *cache;
1566     struct rtable *rt = (struct rtable*)skb->dst;
1567
1568     read_lock(&mrt_lock);
1569     cache = ipmr_cache_find(rt->rt_src, rt->rt_dst);
1570
1571     if (cache==NULL) {
1572         struct net_device *dev;
1573         int vif;
1574
1575         if (nowait) {
1576             read_unlock(&mrt_lock);
1577             return -EAGAIN;
1578         }
1579
1580         dev = skb->dev;
1581         if (dev == NULL || (vif = ipmr_find_vif(dev)) < 0) {

```

Figure 482: IPMR - RT Netlink get route: linux/net/ipv4/ipmr.c

```

1589         skb->nh.iph->version = 0;
1590         err = ipmr_cache_unresolved(vif, skb);
1591         read_unlock(&mrt_lock);
1592         return err;
1593     }
1594
1595     if (!nowait && (rtm->rtm_flags&RTM_F_NOTIFY))
1596         cache->mfc_flags |= MFC_NOTIFY;
1597     err = ipmr_fill_mroute(skb, cache, rtm);
1598     read_unlock(&mrt_lock);
1599     return err;
1600 }
1601 #endif
1602
1603 #ifdef CONFIG_PROC_FS
1604 /*
1605  *   The /proc interfaces to multicast routing /proc/ip_mr_cache /proc/
1606  *   ip_mr_vif
1607  */
1608 static int ipmr_vif_info(char *buffer, char **start, off_t offset, int length)
1609 {
1610     struct vif_device *vif;
1611     int len=0;
1612     off_t pos=0;
1613     off_t begin=0;
1614     int size;
1615     int ct;
1616
1617     len += sprintf(buffer,
1618         "Interface %s BytesIn %s PktsIn %s BytesOut %s PktsOut %s Flags Local %s\n",
1619         Remote\n");
1620
1621     pos=len;
1622
1623     read_lock(&mrt_lock);
1624     for (ct=0;ct<maxvif;ct++)
1625     {
1626         char *name = "none";
1627         vif=&vif_table[ct];
1628         if (!VIF_EXISTS(ct))
1629             continue;
1630         if (vif->dev)
1631             name = vif->dev->name;
1632         size = sprintf(buffer+len, "%2d %10s %8ld %7ld %8ld %7ld %05X\n",
1633             ct, name, vif->bytes_in, vif->pkt_in, vif->bytes_out, vif->
1634             pkt_out,
1635             vif->flags, vif->local, vif->remote);
1636         len+=size;

```

Figure 483: IPMR - PROC FS vif reporting: linux/net/ipv4/ipmr.c

```

1635         if (pos<offset)
1636         {
1637             len=0;
1638             begin=pos;
1639         }
1640         if (pos>offset+length)
1641             break;
1642     }
1643     read_unlock(&mrt_lock);
1644
1645     *start=buffer+(offset-begin);
1646     len-=(offset-begin);
1647     if (len>length)
1648         len=length;
1649     if (len<0)
1650         len = 0;
1651     return len;
1652 }
1653
1654 static int ipmr_mfc_info(char *buffer, char **start, off_t offset, int length)
1655 {
1656     struct mfc_cache *mfc;
1657     int len=0;
1658     off_t pos=0;
1659     off_t begin=0;
1660     int size;
1661     int ct;
1662
1663     len += sprintf(buffer,
1664                   "Group____Origin____Iif____Pkts____Bytes____Wrong_Oifs\n");
1665     pos=len;
1666
1667     read_lock(&mrt_lock);
1668     for (ct=0;ct<MFC_LINES;ct++)
1669     {
1670         for(mfc=mfc_cache_array[ct]; mfc; mfc=mfc->next)
1671         {
1672             int n;
1673
1674             /*
1675              *   Interface forwarding map
1676              */
1677             size = sprintf(buffer+len, "%08lX_%08lX_%-3d_%08ld_%08ld_%8
1678                                ld",
1679                            (unsigned long)mfc->mfc_mcastgrp,
1680                            (unsigned long)mfc->mfc_origin,
1681                            mfc->mfc_parent,
1682                            mfc->mfc_un.res.pkt,

```

Figure 484: IPMR - PROC FS MFC reporting: linux/net/ipv4/ipmr.c

```

1683         mfc->mfc_un.res.wrong_if);
1684     for(n=mfc->mfc_un.res.minvif;n<mfc->mfc_un.res.maxvif;n++)
1685     {
1686         if (VIF_EXISTS(n) && mfc->mfc_un.res.ttls[n] < 255)
1687             size += sprintf(buffer+len+size, "%2d:%-3d", n
1688                             , mfc->mfc_un.res.ttls[n]);
1689     }
1690     size += sprintf(buffer+len+size, "\n");
1691     len+=size;
1692     pos+=size;
1693     if (pos<offset)
1694     {
1695         len=0;
1696         begin=pos;
1697     }
1698     if (pos>offset+length)
1699         goto done;
1700 }
1701
1702 spin_lock_bh(&mfc_unres_lock);
1703 for(mfc=mfc_unres_queue; mfc; mfc=mfc->next) {
1704     size = sprintf(buffer+len, "%08lX,%08lX,%-3d,%8ld,%8ld,%8ld\n",
1705                   (unsigned long)mfc->mfc_mcast_grp,
1706                   (unsigned long)mfc->mfc_origin,
1707                   -1,
1708                   (long)mfc->mfc_un.unres.unresolved.qlen,
1709                   0L, 0L);
1710     len+=size;
1711     pos+=size;
1712     if (pos<offset)
1713     {

```

Figure 485: IPMR - PROC FS MFC reporting: `linux/net/ipv4/ipmr.c`

```

1717         if (pos > offset + length)
1718             break;
1719     }
1720     spin_unlock_bh(&mfc_unres_lock);
1721
1722 done:
1723     read_unlock(&mrt_lock);
1724     *start = buffer + (offset - begin);
1725     len -= (offset - begin);
1726     if (len > length)
1727         len = length;
1728     if (len < 0) {
1729         len = 0;
1730     }
1731     return len;
1732 }
1733
1734 #endif
1735
1736 #ifdef CONFIG_IP_PIMSM_V2
1737 struct inet_protocol pim_protocol =
1738 {
1739     pim_rcv,          /* PIM handler */
1740     NULL,            /* PIM error control */
1741     NULL,            /* next */
1742     IPPROTO_PIM,    /* protocol ID */
1743     0,              /* copy */
1744     NULL,           /* data */
1745     "PIM"          /* name */
1746 };
1747 #endif

```

Figure 486: IPMR - initialise multicast routing: linux/net/ipv4/ipmr.c

```

46 int
47 ip_do_nat(struct sk_buff *skb)
48 {
49     struct rtable *rt = (struct rtable*)skb->dst;
50     struct iphdr *iph = skb->nh.iph;
51     u32 odaddr = iph->daddr;
52     u32 osaddr = iph->saddr;
53     u16  check;
54
55     IPCB(skb)->flags |= IPSKB_TRANSLATED;
56
57     /* Rewrite IP header */
58     iph->daddr = rt->rt_dst_map;
59     iph->saddr = rt->rt_src_map;
60     iph->check = 0;
61     iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);

```

Figure 487: Internet Protocol - Simple NAT: `linux/net/ipv4/ip_nat_dumb.c`

```

63      /* If it is the first fragment, rewrite protocol headers */
64
65      if (!(iph->frag_off & htons(IP_OFFSET))) {
66          u16      *cksum;
67
68          switch(iph->protocol) {
69              case IPPROTO_TCP:
70                  cksum = (u16*)&((struct tcphdr*)((char*)iph) + (iph->ihl<<2))
71                      ->check;
72                  if ((u8*)(cksum+1) > skb->tail)
73                      goto truncated;
74                  check = *cksum;
75                  if (skb->ip_summed != CHECKSUM_HW)
76                      check = ~check;
77                  check = csum_tcpudp_magic(iph->saddr, iph->daddr, 0, 0, check
78                      );
79                  if (skb->ip_summed == CHECKSUM_HW)
80                      check = ~check;
81                  *cksum = check;
82                  break;
83              case IPPROTO_UDP:
84                  cksum = (u16*)&((struct udphdr*)((char*)iph) + (iph->ihl<<2))
85                      ->check;
86                  if ((u8*)(cksum+1) > skb->tail)
87                      goto truncated;

```

Figure 488: Internet Protocol - Simple NAT: linux/net/ipv4/ip_nat_dumb.c

```

86         if ((check = *cksum) != 0) {
87             check = csum_tcpudp_magic(iph->saddr, iph->daddr
88                 , 0, 0, ~check);
89             check = csum_tcpudp_magic(~osaddr, ~odaddr, 0, 0, ~check
90                 );
91             *cksum = check ? : 0xFFFF;
92         }
93     case IPPROTO_ICMP:
94     {
95         struct icmp_hdr *icmp = (struct icmp_hdr*)((char*)iph + (iph->
96             ihl<<2));
97         struct ip_hdr *cip =
98         u32 idaddr, isaddr;
99         int updated;
100
101         if ((icmp->type != ICMP_DEST_UNREACH) &&
102             (icmp->type != ICMP_TIME_EXCEEDED) &&
103             (icmp->type != ICMP_PARAMETERPROB))
104             break;
105
106         cip = (struct ip_hdr *) (icmp + 1);
107
108         if ((u8*)(cip+1) > skb->tail)
109             goto truncated;
110
111         isaddr = cip->saddr;
112         idaddr = cip->daddr;

```

Figure 489: Internet Protocol - Simple NAT: linux/net/ipv4/ip_nat_dumb.c


```

111         updated = 0;
112
113         if (rt->rt_flags&RTCF_DNAT && ciph->saddr == odaddr) {
114             ciph->saddr = iph->daddr;
115             updated = 1;
116         }
117         if (rt->rt_flags&RTCF_SNAT) {
118             if (ciph->daddr != osaddr) {
119                 struct fib_result res;
120                 struct rt_key key;
121                 unsigned flags = 0;
122
123                 key.src = ciph->daddr;
124                 key.dst = ciph->saddr;
125                 key.iif = skb->dev->ifindex;
126                 key.oif = 0;
127 #ifdef CONFIG_IP_ROUTE_TOS
128                 key.tos = RT_TOS(ciph->tos);
129 #endif
130 #ifdef CONFIG_IP_ROUTE_FWMARK
131                 key.fwmark = 0;
132 #endif
133                 /* Use fib_lookup() until we get our own
134                  * hash table of NATed hosts -- Rani
135                  */
136                 if (fib_lookup(&key, &res) == 0) {
137                     if (res.r) {
138                         ciph->daddr = fib_rules_policy(
139                             ciph->daddr, &res, &flags);
140                         if (ciph->daddr != idaddr)
141                             updated = 1;
142                     }
143                     fib_res_put (&res);
144                 } else {
145                     ciph->daddr = iph->saddr;
146                     updated = 1;
147                 }
148             }
149             if (updated) {
150                 cksm = &icmph->checksum;
151                 /* Using tcpudp primitive. Why not? */
152                 check = csum_tcpudp_magic(ciph->saddr, ciph->daddr
153                     , 0, 0, ~(*cksm));
154                 *cksm = csum_tcpudp_magic(~isaddr, ~idaddr, 0, 0, ~
155                     check);
156             }
157             break;
158         }
159         default:
160             break;
161     }
162 }

```

Figure 490: Internet Protocol - Simple NAT: linux/net/ipv4/ip_nat_dumb.c

```

29  /* tunnel.c: an IP tunnel driver
30
31     The purpose of this driver is to provide an IP tunnel through
32     which you can tunnel network traffic transparently across subnets.
33
34     This was written by looking at Nick Holloway's dummy driver
35     Thanks for the great code!
36
37         -Sam Lantinga (slouken@cs.ucdavis.edu) 02/01/95
38
39     Minor tweaks:
40         Cleaned up the code a little and added some pre-1.3.0 tweaks.
41         dev->hard_header/hard_header_len changed to use no headers.
42         Comments/bracketing tweaked.
43         Made the tunnels use dev->name not tunnel: when error reporting.
44         Added tx_dropped stat
45
46         -Alan Cox      (Alan.Cox@linux.org) 21 March 95
47
48     Reworked:
49         Changed to tunnel to destination gateway in addition to the
50         tunnel's pointopoint address
51         Almost completely rewritten
52         Note: There is currently no firewall or ICMP handling done.
53
54         -Sam Lantinga (slouken@cs.ucdavis.edu) 02/13/96
55
56  */

```

Figure 491: IP in IP Tunnel - Opening Remarks: linux/net/ipv4/ipip.c

```

57
58 /* Things I wish I had known when writing the tunnel driver:
59
60     When the tunnel_xmit() function is called, the skb contains the
61     packet to be sent (plus a great deal of extra info), and dev
62     contains the tunnel device that _we_ are.
63
64     When we are passed a packet, we are expected to fill in the
65     source address with our source IP address.
66
67     What is the proper way to allocate, copy and free a buffer?
68     After you allocate it, it is a "0 length" chunk of memory
69     starting at zero. If you want to add headers to the buffer
70     later, you'll have to call "skb_reserve(skb, amount)" with
71     the amount of memory you want reserved. Then, you call
72     "skb_put(skb, amount)" with the amount of space you want in
73     the buffer. skb_put() returns a pointer to the top (#0) of
74     that buffer. skb->len is set to the amount of space you have
75     "allocated" with skb_put(). You can then write up to skb->len
76     bytes to that buffer. If you need more, you can call skb_put()
77     again with the additional amount of space you need. You can
78     find out how much more space you can allocate by calling
79     "skb_tailroom(skb)".
80     Now, to add header space, call "skb_push(skb, header_len)".
81     This creates space at the beginning of the buffer and returns
82     a pointer to this new space. If later you need to strip a
83     header from a buffer, call "skb_pull(skb, header_len)".
84     skb_headroom() will return how much space is left at the top
85     of the buffer (before the main data). Remember, this headroom
86     space must be reserved before the skb_put() function is called.
87     */
88
89 /*
90     This version of net/ipv4/ipip.c is cloned of net/ipv4/ip_gre.c
91
92     For comments look at net/ipv4/ip_gre.c --ANK
93     */

```

Figure 492: Internet Protocol - room for improvement:
linux/net/ipv4/ipip.c

```

119 #define HASH_SIZE 16
120 #define HASH(addr) ((addr^(addr>>4))&0xF)
121
122 static int ipip_fb_tunnel_init (struct net_device *dev);
123 static int ipip_tunnel_init (struct net_device *dev);
124
125 static struct net_device ipip_fb_tunnel_dev = {
126     name: "tunl0",
127     init: ipip_fb_tunnel_init,
128 };
129
130 static struct ip_tunnel ipip_fb_tunnel = {
131     dev: &ipip_fb_tunnel_dev,
132     parms: { name: "tunl0", }
133 };
134
135 static struct ip_tunnel *tunnels_r_l [HASH_SIZE];
136 static struct ip_tunnel *tunnels_r [HASH_SIZE];
137 static struct ip_tunnel *tunnels_l [HASH_SIZE];
138 static struct ip_tunnel *tunnels_wc [1];
139 static struct ip_tunnel **tunnels[4] = { tunnels_wc, tunnels_l, tunnels_r, tunnels_r_l };

```

Figure 493: IP in IP - Data : linux/net/ipv4/ipip.c

```

141 static rwlock_t ipip_lock = RW_LOCK_UNLOCKED;
142
143 static struct ip_tunnel * ipip_tunnel_lookup(u32 remote, u32 local)
144 {
145     unsigned h0 = HASH(remote);
146     unsigned h1 = HASH(local);
147     struct ip_tunnel *t;
148
149     for (t = tunnels_r_l[h0^h1]; t; t = t->next) {
150         if (local == t->parms.iph.saddr &&
151             remote == t->parms.iph.daddr && (t->dev->flags&IFF_UP))
152             return t;
153     }
154     for (t = tunnels_r[h0]; t; t = t->next) {
155         if (remote == t->parms.iph.daddr && (t->dev->flags&IFF_UP))
156             return t;
157     }
158     for (t = tunnels_l[h1]; t; t = t->next) {
159         if (local == t->parms.iph.saddr && (t->dev->flags&IFF_UP))
160             return t;
161     }
162     if ((t = tunnels_wc[0]) != NULL && (t->dev->flags&IFF_UP))
163         return t;

```

Figure 494: IP in IP - Lookup Tunnel based on local and remote addresses:
linux/net/ipv4/ipip.c

```

165 }
166
167 static struct ip_tunnel **ipip_bucket(struct ip_tunnel *t)
168 {
169     u32 remote = t->parms.iph.daddr;
170     u32 local = t->parms.iph.saddr;
171     unsigned h = 0;
172     int prio = 0;
173
174     if (remote) {
175         prio |= 2;
176         h ^= HASH(remote);
177     }
178     if (local) {
179         prio |= 1;
180         h ^= HASH(local);
181     }

```

Figure 495: IP in IP - Tunnel hash bucket: `linux/net/ipv4/ipip.c`

```

184
185
186 static void ipip_tunnel_unlink(struct ip_tunnel *t)
187 {
188     struct ip_tunnel **tp;
189
190     for (tp = ipip_bucket(t); *tp; tp = &(*tp)->next) {
191         if (t == *tp) {
192             write_lock_bh(&ipip_lock);
193             *tp = t->next;
194             write_unlock_bh(&ipip_lock);
195             break;
196         }
197     }
198 }
199
200 static void ipip_tunnel_link(struct ip_tunnel *t)
201 {
202     struct ip_tunnel **tp = ipip_bucket(t);
203
204     t->next = *tp;
205     write_lock_bh(&ipip_lock);
206     *tp = t;

```

Figure 496: IP in IP - unlink/link tunnel hash entry: `linux/net/ipv4/ipip.c`

```

208 }
209
210 struct ip_tunnel * ipip_tunnel_locate (struct ip_tunnel_parm *parms, int create)
211 {
212     u32 remote = parms->iph.daddr;
213     u32 local = parms->iph.saddr;
214     struct ip_tunnel *t, **tp, *nt;
215     struct net_device *dev;
216     unsigned h = 0;
217     int prio = 0;
218
219     if (remote) {
220         prio |= 2;
221         h ^= HASH(remote);
222     }
223     if (local) {
224         prio |= 1;
225         h ^= HASH(local);
226     }
227     for (tp = &tunnels[prio][h]; (t = *tp) != NULL; tp = &t->next) {
228         if (local == t->parms.iph.saddr && remote == t->parms.iph.daddr)
229             return t;
230     }
231     if (!create)
232         return NULL;
233
234     MOD_INC_USE_COUNT;
235     dev = kmalloc(sizeof(*dev) + sizeof(*t), GFP_KERNEL);
236     if (dev == NULL) {
237         MOD_DEC_USE_COUNT;

```

Figure 497: IP in IP - locate a tunnel: `linux/net/ipv4/ipip.c`


```

239     }
240     memset(dev, 0, sizeof(*dev) + sizeof(*t));
241     dev->priv = (void*)(dev+1);
242     nt = (struct ip_tunnel*)dev->priv;
243     nt->dev = dev;
244     dev->init = ipip_tunnel_init;
245     dev->features |= NETIF_F_DYNALLOC;
246     memcpy(&nt->parms, parms, sizeof(*parms));
247     strcpy(dev->name, nt->parms.name);
248     if (dev->name[0] == 0) {
249         int i;
250         for (i=1; i<100; i++) {
251             sprintf(dev->name, "tunl%d", i);
252             if (_dev_get_by_name(dev->name) == NULL)
253                 break;
254         }
255         if (i==100)
256             goto failed;
257         memcpy(parms->name, dev->name, IFNAMSIZ);
258     }
259     if (register_netdevice (dev) < 0)
260         goto failed;
261
262     dev_hold(dev);
263     ipip_tunnel_link(nt);
264     /* Do not decrement MOD_USE_COUNT here. */
265     return nt;
266
267 failed:
268     kfree(dev);
269     MOD_DEC_USE_COUNT;

```

Figure 498: IP in IP - locate a tunnel: `linux/net/ipv4/ipip.c`

```

271 }
272
273 static void ipip_tunnel_destructor (struct net_device *dev)
274 {
275     if (dev != &ipip_fb_tunnel_dev) {
276         MOD_DEC_USE_COUNT;
277     }
278 }
279
280 static void ipip_tunnel_uninit (struct net_device *dev)
281 {
282     if (dev == &ipip_fb_tunnel_dev) {
283         write_lock_bh(&ipip_lock);
284         tunnels_wc[0] = NULL;
285         write_unlock_bh(&ipip_lock);
286     } else
287         ipip_tunnel_unlink ((struct ip_tunnel *)dev->priv);
288     dev_put(dev);
289 }

```

Figure 499: IP in IP - uninitialise and destroy a tunnel:
linux/net/ipv4/ipip.c

```

291 void ipip_err(struct sk_buff *skb, u32 info)
292 {
293     #ifndef L_WISH_WORLD_WERE_PERFECT
294
295     /* It is not :( All the routers (except for Linux) return only
296        8 bytes of packet payload. It means, that precise relaying of
297        ICMP in the real Internet is absolutely infeasible.
298     */
299     struct iphdr *iph = (struct iphdr*)skb->data;
300     int type = skb->h.icmph->type;
301     int code = skb->h.icmph->code;
302     struct ip_tunnel *t;
303
304     switch (type) {
305     default:
306     case ICMP_PARAMETERPROB:
307         return;
308
309     case ICMP_DEST_UNREACH:
310         switch (code) {

```

Figure 500: IP in IP - ICMP Error handling: linux/net/ipv4/ipip.c

```

312         case ICMP_PORT_UNREACH:
313             /* Impossible event. */
314             return;
315         case ICMP_FRAG_NEEDED:
316             /* Soft state for pmtu is maintained by IP core. */
317             return;
318         default:
319             /* All others are translated to HOST_UNREACH.
320              * rfc2003 contains "deep thoughts" about NET_UNREACH,
321              * I believe they are just ether pollution. --ANK
322              */
323             break;
324     }
325     break;
326     case ICMP_TIME_EXCEEDED:
327         if (code != ICMP_EXC_TTL)
328             return;
329         break;
330 }
331
332 read_lock(&ipip_lock);
333 t = ipip_tunnel_lookup(iph->daddr, iph->saddr);
334 if (t == NULL || t->parms.iph.daddr == 0)
335     goto out;
336 if (t->parms.iph.ttl == 0 && type == ICMP_TIME_EXCEEDED)
337     goto out;
338
339 if ( jiffies - t->err_time < IPTUNNEL_ERR_TIMEO)
340     t->err_count++;
341 else
342     t->err_count = 1;
343 t->err_time = jiffies;
344 out:
345 read_unlock(&ipip_lock);
346 return;
347 #else
348 struct iphdr *iph = (struct iphdr*)dp;
349 int hlen = iph->ihl<<2;
350 struct iphdr *eiph;
351 int type = skb->h.icmph->type;
352 int code = skb->h.icmph->code;
353 int rel_type = 0;
354 int rel_code = 0;
355 int rel_info = 0;
356 struct sk_buff *skb2;
357 struct rtable *rt;
358
359 if (len < hlen + sizeof(struct iphdr))
360     return;

```

Figure 501: IP in IP - ICMP Error handling: linux/net/ipv4/ipip.c

```

362
363     switch (type) {
364     default:
365         return;
366     case ICMP_PARAMETERPROB:
367         if (skb->h.icmph->un.gateway < hlen)
368             return;
369
370         /* So... This guy found something strange INSIDE encapsulated
371            packet. Well, he is fool, but what can we do ?
372            */
373         rel_type = ICMP_PARAMETERPROB;
374         rel_info = skb->h.icmph->un.gateway - hlen;
375         break;
376
377     case ICMP_DEST_UNREACH:
378         switch (code) {
379         case ICMP_SR_FAILED:
380         case ICMP_PORT_UNREACH:
381             /* Impossible event. */
382             return;
383         case ICMP_FRAG_NEEDED:
384             /* And it is the only really necessary thing :- ) */
385             rel_info = ntohs(skb->h.icmph->un.frag.mtu);
386             if (rel_info < hlen+68)
387                 return;
388             rel_info -= hlen;
389             /* BSD 4.2 MORE DOES NOT EXIST IN NATURE. */
390             if (rel_info > ntohs(eiph->tot_len))
391                 return;
392             break;
393         default:
394             /* All others are translated to HOST_UNREACH.
395                rfc2003 contains "deep thoughts" about NET_UNREACH,
396                I believe, it is just ether pollution. --ANK
397                */
398             rel_type = ICMP_DEST_UNREACH;
399             rel_code = ICMP_HOST_UNREACH;
400             break;
401         }
402         break;
403     case ICMP_TIME_EXCEEDED:
404         if (code != ICMP_EXC_TTL)
405             return;

```

Figure 502: IP in IP - ICMP Error handling: linux/net/ipv4/ipip.c

```
406         break;
407     }
408
409     /* Prepare fake skb to feed it to icmp_send */
410     skb2 = skb_clone(skb, GFP_ATOMIC);
```

Figure 503: IP in IP - ICMP Error handling: `linux/net/ipv4/ipip.c`

```

412     return;
413     dst_release(skb2->dst);
414     skb2->dst = NULL;
415     skb_pull(skb2, skb->data - (u8*)eiph);
416     skb2->nh.raw = skb2->data;
417
418     /* Try to guess incoming interface */
419     if (ip_route_output(&rt, eiph->saddr, 0, RT_TOS(eiph->tos), 0)) {
420         kfree_skb(skb2);
421         return;
422     }
423     skb2->dev = rt->u.dst.dev;
424
425     /* route "incoming" packet */
426     if (rt->rt_flags&RTCF_LOCAL) {
427         ip_rt_put(rt);
428         rt = NULL;
429         if (ip_route_output(&rt, eiph->daddr, eiph->saddr, eiph->tos, 0) ||
430             rt->u.dst.dev->type != ARPHRD_IPGRE) {
431             ip_rt_put(rt);
432             kfree_skb(skb2);
433             return;
434         }
435     } else {
436         ip_rt_put(rt);
437         if (ip_route_input(skb2, eiph->daddr, eiph->saddr, eiph->tos, skb2->
438             dev) ||
439             skb2->dst->dev->type != ARPHRD_IPGRE) {
440             kfree_skb(skb2);
441             return;
442         }
443     }
444
445     /* change mtu on this route */
446     if (type == ICMP_DEST_UNREACH && code == ICMP_FRAG_NEEDED) {

```

Figure 504: IP in IP - ICMP Error handling: linux/net/ipv4/ipip.c

```

447         kfree_skb(skb2);
448         return;
449     }
450     skb2->dst->pmtu = rel_info;
451     rel_info = htonl(rel_info);
452 } else if (type == ICMP_TIME_EXCEEDED) {
453     struct ip_tunnel *t = (struct ip_tunnel*)skb2->dev->priv;
454     if (t->parms.iph.ttl) {
455         rel_type = ICMP_DEST_UNREACH;
456         rel_code = ICMP_HOST_UNREACH;
457     }
458 }
459
460 icmp_send(skb2, rel_type, rel_code, rel_info);
461 kfree_skb(skb2);
462 return;
463 #endif
464 }
465
466 static inline void ipip_ecn_decapsulate(struct iphdr *iph, struct sk_buff *skb)
467 {

```

Figure 505: IP in IP - ICMP Error handling: `linux/net/ipv4/ipip.c`


```
469     INET_ECN_is_not_ce(skb->nh.iph->tos))
470     IP_ECN_set_ce(iph);
471 }
472
473 int ipip_rcv(struct sk_buff *skb)
474 {
```

Figure 506: IP in IP - ECN handling: `linux/net/ipv4/ipip.c`

```

476     struct ip_tunnel *tunnel;
477
478     if (!pskb_may_pull(skb, sizeof(struct iphdr)))
479         goto out;
480
481     iph = skb->nh.iph;
482     skb->mac.raw = skb->nh.raw;
483     skb->nh.raw = skb->data;
484     memset(&(IPCB(skb)->opt), 0, sizeof(struct ip_options));
485     skb->protocol = _constant_htons(ETH_P_IP);
486     skb->pkt_type = PACKET_HOST;
487
488     read_lock(&ipip_lock);
489     if ((tunnel = ipip_tunnel_lookup(iph->saddr, iph->daddr)) != NULL) {
490         tunnel->stat.rx_packets++;
491         tunnel->stat.rx_bytes += skb->len;
492         skb->dev = tunnel->dev;
493         dst_release(skb->dst);
494         skb->dst = NULL;
495 #ifdef CONFIG_NETFILTER
496         nf_conntrack_put(skb->nfct);
497         skb->nfct = NULL;
498 #ifdef CONFIG_NETFILTER_DEBUG
499         skb->nf_debug = 0;
500 #endif
501 #endif
502         ipip_ecn_decapsulate(iph, skb);
503         netif_rx(skb);
504         read_unlock(&ipip_lock);
505         return 0;
506     }
507     read_unlock(&ipip_lock);
508
509     icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PROT_UNREACH, 0);
510 out:
511     kfree_skb(skb);
512     return 0;
513 }

```

Figure 507: IP in IP - Receive Code: linux/net/ipv4/ipip.c

```
515 /* Need this wrapper because NF_HOOK takes the function address */
516 static inline int do_ip_send(struct sk_buff *skb)
517 {
518     return ip_send(skb);
519 }
```

Figure 508: IP in IP - Tx Wrapper for NF Hook: `linux/net/ipv4/ipip.c`

```

521  /*
522  *   This function assumes it is being called from dev_queue_xmit()
523  *   and that skb is filled properly by that function.
524  */
525
526  static int ipip_tunnel_xmit(struct sk_buff *skb, struct net_device *dev)
527  {
528      struct ip_tunnel *tunnel = (struct ip_tunnel*)dev->priv;
529      struct net_device_stats *stats = &tunnel->stat;
530      struct iphdr *tiph = &tunnel->parms.iph;
531      u8   tos = tunnel->parms.iph.tos;
532      u16  df = tiph->frag_off;
533      struct rtable *rt;           /* Route to the other host */
534      struct net_device *tdev;    /* Device to other host */
535      struct iphdr *old_iph = skb->nh.iph;
536      struct iphdr *iph;         /* Our new IP header */
537      int   max_headroom;        /* The extra header space needed */
538      u32   dst = tiph->daddr;
539      int   mtu;
540
541      if (tunnel->recursion++) {
542          tunnel->stat.collisions++;
543          goto tx_error;
544      }
545
546      if (skb->protocol != _constant_htons(ETH_P_IP))
547          goto tx_error;
548
549      if (tos&1)
550          tos = old_iph->tos;
551
552      if (!dst) {
553          /* NBMA tunnel */
554          if ((rt = (struct rtable*)skb->dst) == NULL) {
555              tunnel->stat.tx_fifo_errors++;
556              goto tx_error;
557          }
558          if ((dst = rt->rt_gateway) == 0)
559              goto tx_error_icmp;
560      }

```

Figure 509: IP in IP - Transmit Code: linux/net/ipv4/ipip.c

```

562     if (ip_route_output(&rt, dst, tiph->saddr, RT_TOS(tos), tunnel->parms.link)) {
563         tunnel->stat.tx_carrier_errors++;
564         goto tx_error_icmp;
565     }
566     tdev = rt->u.dst.dev;
567
568     if (tdev == dev) {
569         ip_rt_put(rt);
570         tunnel->stat.collisions++;
571         goto tx_error;
572     }
573
574     mtu = rt->u.dst.pmtu - sizeof(struct iphdr);
575     if (mtu < 68) {
576         tunnel->stat.collisions++;
577         ip_rt_put(rt);
578         goto tx_error;
579     }
580     if (skb->dst && mtu < skb->dst->pmtu)
581         skb->dst->pmtu = mtu;
582
583     df |= (old_iph->frag_off & _constant_htons(IP_DF));
584
585     if ((old_iph->frag_off & _constant_htons(IP_DF)) && mtu < ntohs(old_iph->
586         tot_len)) {
587         icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED, htonl(
588             mtu));
589         ip_rt_put(rt);
590         goto tx_error;
591     }
592
593     if (tunnel->err_count > 0) {
594         if (jiffies - tunnel->err_time < IPTUNNEL_ERR_TIMEO) {
595             tunnel->err_count--;
596             dst_link_failure(skb);
597         } else
598             tunnel->err_count = 0;
599     }
600
601     skb->h.raw = skb->nh.raw;

```

Figure 510: IP in IP - Transmit Code: linux/net/ipv4/ipip.c

```

601     /*
602     * Okay, now see if we can stuff it in the buffer as-is.
603     */
604     max_headroom = (((tdev->hard_header_len+15)&~15)+sizeof(struct iphdr));
605
606     if (skb_headroom(skb) < max_headroom || skb_cloned(skb) || skb_shared(skb)) {
607         struct sk_buff *new_skb = skb_realloc_headroom(skb, max_headroom);
608         if (!new_skb) {
609             ip_rt_put(rt);
610             stats->tx_dropped++;
611             dev_kfree_skb(skb);
612             tunnel->recursion--;
613             return 0;
614         }
615         if (skb->sk)
616             skb_set_owner_w(new_skb, skb->sk);
617         dev_kfree_skb(skb);
618         skb = new_skb;
619     }
620
621     skb->nh.raw = skb_push(skb, sizeof(struct iphdr));
622     memset(&(IPCB(skb)->opt), 0, sizeof(IPCB(skb)->opt));
623     dst_release(skb->dst);
624     skb->dst = &rt->u.dst;

```

Figure 511: IP in IP - Transmit Code: `linux/net/ipv4/ipip.c`

```

626     /*
627     *       Push down and install the IPIP header.
628     */
629
630     iph                =     skb->nh.iph;
631     iph->version        =     4;
632     iph->ihl            =     sizeof(struct iphdr)>>2;
633     iph->frag_off      =     df;
634     iph->protocol      =     IPPROTO_IPIP;
635     iph->tos            =    _INET_ECN_encapsulate(tos, old_iph->tos);
636     iph->daddr         =     rt->rt_dst;
637     iph->saddr         =     rt->rt_src;
638
639     if ((iph->ttl = tiph->ttl) == 0)
640         iph->ttl     =     old_iph->ttl;
641
642 #ifdef CONFIG_NETFILTER
643     nf_conntrack_put(skb->nfct);
644     skb->nfct = NULL;
645 #ifdef CONFIG_NETFILTER_DEBUG
646     skb->nf_debug = 0;
647 #endif
648 #endif
649
650     IPTUNNEL_XMIT();
651     tunnel->recursion--;
652     return 0;
653
654 tx_error_icmp:
655     dst_link_failure(skb);
656 tx_error:
657     stats->tx_errors++;
658     dev_kfree_skb(skb);
659     tunnel->recursion--;
660     return 0;
661 }

```

Figure 512: IP in IP - Transmit Code: linux/net/ipv4/ipip.c

```

663 static int
664 ipip_tunnel_ioctl (struct net_device *dev, struct ifreq *ifr, int cmd)
665 {
666     int err = 0;
667     struct ip_tunnel_parm p;
668     struct ip_tunnel *t;
669
670     MOD_INC_USE_COUNT;
671
672     switch (cmd) {
673     case SIOCGTUNNEL:
674         t = NULL;
675         if (dev == &ipip_fb_tunnel_dev) {
676             if (copy_from_user(&p, ifr->ifru.ifru_data, sizeof(p))) {
677                 err = -EFAULT;
678                 break;
679             }
680             t = ipip_tunnel_locate(&p, 0);
681         }
682         if (t == NULL)
683             t = (struct ip_tunnel*)dev->priv;
684         memcpy(&p, &t->parms, sizeof(p));
685         if (copy_to_user(ifr->ifru.ifru_data, &p, sizeof(p)))
686             err = -EFAULT;
687         break;

```

Figure 513: IP in IP - Ioctl entry: `linux/net/ipv4/ipip.c`


```

689     case SIOCADDTUNNEL:
690     case SIOCCHGTUNNEL:
691         err = -EPERM;
692         if (!capable(CAP_NET_ADMIN))
693             goto done;
694
695         err = -EFAULT;
696         if (copy_from_user(&p, ifr->ifru.ifru_data, sizeof(p)))
697             goto done;
698
699         err = -EINVAL;
700         if (p.iph.version != 4 || p.iph.protocol != IPPROTO_IPIP ||
701             p.iph.ihl != 5 || (p.iph.frag_off & _constant_htons(~IP_DF)))
702             goto done;
703         if (p.iph.ttl)
704             p.iph.frag_off |= _constant_htons(IP_DF);
705
706         t = ipip_tunnel_locate(&p, cmd == SIOCADDTUNNEL);
707
708         if (dev != &ipip_fb_tunnel_dev && cmd == SIOCCHGTUNNEL &&
709             t != &ipip_fb_tunnel) {
710             if (t != NULL) {
711                 if (t->dev != dev) {
712                     err = -EEXIST;
713                     break;
714                 }
715             } else {
716                 if (((dev->flags&IFF_POINTOPOINT) && !p.iph.daddr
717                     ) ||
718                     (!(dev->flags&IFF_POINTOPOINT) && p.iph.daddr
719                     )) {
720                     err = -EINVAL;
721                     break;
722                 }
723                 t = (struct ip_tunnel*)dev->priv;
724                 ipip_tunnel_unlink(t);
725                 t->parms.iph.saddr = p.iph.saddr;
726                 t->parms.iph.daddr = p.iph.daddr;
727                 memcpy(dev->dev_addr, &p.iph.saddr, 4);
728                 memcpy(dev->broadcast, &p.iph.daddr, 4);
729                 ipip_tunnel_link(t);
730                 netdev_state_change(dev);
731             }
732         }
733     }

```

Figure 514: IP in IP - Ioctl entry: linux/net/ipv4/ipip.c

```

731         if (t) {
732             err = 0;
733             if (cmd == SIOCCHGTUNNEL) {
734                 t->parms.iph.ttl = p.iph.ttl;
735                 t->parms.iph.tos = p.iph.tos;
736                 t->parms.iph.frag_off = p.iph.frag_off;
737             }
738             if (copy_to_user(ifr->ifru.ifru_data, &t->parms, sizeof(p)))
739                 err = -EFAULT;
740         } else
741             err = (cmd == SIOCADDTUNNEL ? -ENOBUFS : -ENOENT);
742         break;
743
744     case SIOCDELTUNNEL:
745         err = -EPERM;
746         if (!capable(CAP_NET_ADMIN))
747             goto done;
748
749         if (dev == &ipip_fb_tunnel_dev) {
750             err = -EFAULT;
751             if (copy_from_user(&p, ifr->ifru.ifru_data, sizeof(p)))
752                 goto done;
753             err = -ENOENT;
754             if ((t = ipip_tunnel_locate(&p, 0)) == NULL)
755                 goto done;
756             err = -EPERM;
757             if (t == &ipip_fb_tunnel)
758                 goto done;
759         }
760         err = unregister_net_device(dev);
761         break;
762
763     default:
764         err = -EINVAL;
765     }
766
767 done:
768     MOD_DEC_USE_COUNT;
769     return err;
770 }

```

Figure 515: IP in IP - Ioctl entry: linux/net/ipv4/ipip.c

```
773 static struct net_device_stats * ipip_tunnel_get_stats (struct net_device *dev)
774 {
775     return &(((struct ip_tunnel*)dev->priv)->stat);
776 }
777
778 static int ipip_tunnel_change_mtu(struct net_device *dev, int new_mtu)
779 {
780     if (new_mtu < 68 || new_mtu > 0xFFF8 - sizeof(struct iphdr))
781         return -EINVAL;
782     dev->mtu = new_mtu;
783     return 0;
784 }
```

Figure 516: IP in IP - stats: linux/net/ipv4/ipip.c

```

786 static void ipip_tunnel_init_gen (struct net_device *dev)
787 {
788     struct ip_tunnel *t = (struct ip_tunnel*)dev->priv;
789
790     dev->uninit          = ipip_tunnel_uninit ;
791     dev->destructor      = ipip_tunnel_destructor ;
792     dev->hard_start_xmit = ipip_tunnel_xmit ;
793     dev->get_stats       = ipip_tunnel_get_stats ;
794     dev->do_ioctl        = ipip_tunnel_ioctl ;
795     dev->change_mtu      = ipip_tunnel_change_mtu ;
796
797     dev->type             = ARPHRD_TUNNEL ;
798     dev->hard_header_len = LL_MAX_HEADER + sizeof(struct iphdr) ;
799     dev->mtu              = 1500 - sizeof(struct iphdr) ;
800     dev->flags            = IFF_NOARP ;
801     dev->iflink           = 0 ;
802     dev->addr_len        = 4 ;
803     memcpy(dev->dev_addr, &t->parms.iph.saddr, 4) ;
804     memcpy(dev->broadcast, &t->parms.iph.daddr, 4) ;
805 }
806
807 static int ipip_tunnel_init (struct net_device *dev)

```

Figure 517: IP in IP - general initialisation: `linux/net/ipv4/ipip.c`

```

809     struct net_device *tdev = NULL;
810     struct ip_tunnel *tunnel;
811     struct iphdr *iph;
812
813     tunnel = (struct ip_tunnel*)dev->priv;
814     iph = &tunnel->parms.iph;
815
816     ipip_tunnel_init_gen (dev);
817
818     if (iph->daddr) {
819         struct rtable *rt;
820         if (!ip_route_output(&rt, iph->daddr, iph->saddr, RT_TOS(iph->tos),
821                             tunnel->parms.link)) {
822             tdev = rt->u.dst.dev;
823             ip_rt_put (rt);
824         }
825         dev->flags |= IFF_POINTOPOINT;
826     }
827
828     if (!tdev && tunnel->parms.link)
829         tdev = _dev_get_by_index(tunnel->parms.link);
830
831     if (tdev) {
832         dev->hard_header_len = tdev->hard_header_len + sizeof(struct iphdr);
833         dev->mtu = tdev->mtu - sizeof(struct iphdr);
834     }
835     dev->iflink = tunnel->parms.link;
836
837     return 0;
838 }
839 #ifdef MODULE

```

Figure 518: IP in IP - initialise a tunnel: `linux/net/ipv4/ipip.c`

```
841 {
842     MOD_INC_USE_COUNT;
843     return 0;
844 }
845
846 static int ipip_fb_tunnel_close (struct net_device *dev)
847 {
848     MOD_DEC_USE_COUNT;
849     return 0;
850 }
851 #endif
852
853 int _init ipip_fb_tunnel_init (struct net_device *dev)
```

Figure 519: IP in IP - module setup/teardown: `linux/net/ipv4/ipip.c`

```

855     struct iphdr *iph;
856
857     ipip_tunnel_init_gen (dev);
858 #ifdef MODULE
859     dev->open          = ipip_fb_tunnel_open;
860     dev->stop          = ipip_fb_tunnel_close;
861 #endif
862
863     iph = &ipip_fb_tunnel.parms.iph;
864     iph->version       = 4;
865     iph->protocol      = IPPROTO_IPIP;
866     iph->ihl           = 5;
867
868     dev_hold(dev);
869     tunnels_wc[0]     = &ipip_fb_tunnel;
870     return 0;
871 }
872
873 static struct inet_protocol ipip_protocol = {
874     handler:          ipip_rcv ,
875     err_handler:      ipip_err ,
876     protocol:         IPPROTO_IPIP,
877     name:             "IPIP"
878 };
879
880 static const char banner[] __initdata =
881     KERN_INFO "IPv4 over IPv4 tunneling driver\n";
882
883 int __init ipip_init (void)

```

Figure 520: IP in IP - module init: linux/net/ipv4/ipip.c

```

48 Problems & solutions
49 -----
50
51 1. The most important issue is detecting local dead loops.
52 They would cause complete host lockup in transmit, which
53 would be "resolved" by stack overflow or, if queueing is enabled,
54 with infinite looping in net_bh.
55
56 We cannot track such dead loops during route installation,
57 it is infeasible task. The most general solutions would be
58 to keep skb->encapsulation counter (sort of local ttl),
59 and silently drop packet when it expires. It is the best
60 solution, but it supposes maintaing new variable in ALL
61 skb, even if no tunneling is used.
62
63 Current solution: t->recursion lock breaks dead loops. It looks
64 like dev->tbusy flag, but I preferred new variable, because
65 the semantics is different. One day, when hard_start_xmit
66 will be multithreaded we will have to use skb->encapsulation.

```

Figure 521: GRE Tunnels - Opening Salvo: `linux/net/ipv4/ip_gre.c`


```

70 2. Networking dead loops would not kill routers, but would really
71 kill network. IP hop limit plays role of "t->recursion" in this case,
72 if we copy it from packet being encapsulated to upper header.
73 It is very good solution, but it introduces two problems:
74
75 - Routing protocols, using packets with ttl=1 (OSPF, RIP2),
76 do not work over tunnels.
77 - traceroute does not work. I planned to relay ICMP from tunnel,
78 so that this problem would be solved and traceroute output
79 would even more informative. This idea appeared to be wrong:
80 only Linux complies to rfc1812 now (yes, guys, Linux is the only
81 true router now :-)), all routers (at least, in neighbourhood of mine)
82 return only 8 bytes of payload. It is the end.
83
84 Hence, if we want that OSPF worked or traceroute said something reasonable,
85 we should search for another solution.
86
87 One of them is to parse packet trying to detect inner encapsulation
88 made by our node. It is difficult or even impossible, especially,
89 taking into account fragmentation. TO be short, it is not solution at all.
90
91 Current solution: The solution was UNEXPECTEDLY SIMPLE.
92 We force DF flag on tunnels with preconfigured hop limit,
93 that is ALL. :-) Well, it does not remove the problem completely,
94 but exponential growth of network traffic is changed to linear
95 (branches, that exceed pmtu are pruned) and tunnel mtu
96 fastly degrades to value <68, where looping stops.
97 Yes, it is not good if there exists a router in the loop,
98 which does not force DF, even when encapsulating packets have DF set.
99 But it is not our problem! Nobody could accuse us, we made
100 all that we could make. Even if it is your gated who injected
101 fatal route to network, even if it were you who configured
102 fatal static route: you are innocent. :-)
103
104
105
106 3. Really, ipv4/ipip.c, ipv4/ip_gre.c and ipv6/sit.c contain
107 practically identical code. It would be good to glue them
108 together, but it is not very evident, how to make them modular.
109 sit is integral part of IPv6, ipip and gre are naturally modular.
110 We could extract common parts (hash table, ioctl etc)
111 to a separate module (ip_tunnel.c).

```

Figure 522: GRE Tunnels - Opening Salvo: `linux/net/ipv4/ip_gre.c`

```

116 static int ipgre_tunnel_init (struct net_device *dev);
117
118 /* Fallback tunnel: no source, no destination, no key, no options */
119
120 static int ipgre_fb_tunnel_init (struct net_device *dev);
121
122 static struct net_device ipgre_fb_tunnel_dev = {
123     "gre0", 0x0, 0x0, 0x0, 0x0, 0, 0, 0, 0, 0, NULL, ipgre_fb_tunnel_init,
124 };
125
126 static struct ip_tunnel ipgre_fb_tunnel = {
127     NULL, &ipgre_fb_tunnel_dev, {0}, 0, 0, 0, 0, 0, 0, { "gre0", }
128 };
129
130 /* Tunnel hash table */
131
132 /*
133     4 hash tables:
134
135     3: (remote,local)
136     2: (remote,*)
137     1: (*,local)
138     0: (*,*)
139
140     We require exact key match i.e. if a key is present in packet
141     it will match only tunnel with the same key; if it is not present,
142     it will match only keyless tunnel.
143
144     All keyless packets, if not matched configured keyless tunnels
145     will match fallback tunnel.
146 */
147
148 #define HASH_SIZE 16
149 #define HASH(addr) ((addr^(addr>>4))&0xF)
150
151 static struct ip_tunnel *tunnels[4][HASH_SIZE];
152
153 #define tunnels_rl (tunnels[3])
154 #define tunnels_r (tunnels[2])
155 #define tunnels_l (tunnels[1])
156 #define tunnels_wc (tunnels[0])
157
158 static rwlock_t ipgre_lock = RW_LOCK_UNLOCKED;

```

Figure 523: GRE Tunnel Data: linux/net/ipv4/ip_gre.c

```

160 /* Given src, dst and key, find appropriate for input tunnel. */
161
162 static struct ip_tunnel * ipgre_tunnel_lookup(u32 remote, u32 local, u32 key)
163 {
164     unsigned h0 = HASH(remote);
165     unsigned h1 = HASH(key);
166     struct ip_tunnel *t;
167
168     for (t = tunnels_r_l[h0^h1]; t; t = t->next) {
169         if (local == t->parms.iph.saddr && remote == t->parms.iph.daddr) {
170             if (t->parms.i_key == key && (t->dev->flags&IFF_UP))
171                 return t;
172         }
173     }
174     for (t = tunnels_r[h0^h1]; t; t = t->next) {
175         if (remote == t->parms.iph.daddr) {
176             if (t->parms.i_key == key && (t->dev->flags&IFF_UP))
177                 return t;
178         }
179     }
180     for (t = tunnels_l[h1]; t; t = t->next) {
181         if (local == t->parms.iph.saddr ||
182             (local == t->parms.iph.daddr && MULTICAST(local))) {
183             if (t->parms.i_key == key && (t->dev->flags&IFF_UP))
184                 return t;
185         }
186     }
187     for (t = tunnels_wc[h1]; t; t = t->next) {
188         if (t->parms.i_key == key && (t->dev->flags&IFF_UP))
189             return t;
190     }
191     if (ipgre_fb_tunnel_dev.flags&IFF_UP)
192         return &ipgre_fb_tunnel;
193     return NULL;
194 }

```

Figure 524: GRE Tunnels - Hash Function: linux/net/ipv4/ip_gre.c

```

196 static struct ip_tunnel **ipgre_bucket(struct ip_tunnel *t)
197 {
198     u32 remote = t->parms.iph.daddr;
199     u32 local = t->parms.iph.saddr;
200     u32 key = t->parms.i_key;
201     unsigned h = HASH(key);
202     int prio = 0;
203
204     if (local)
205         prio |= 1;
206     if (remote && !MULTICAST(remote)) {
207         prio |= 2;
208         h ^= HASH(remote);
209     }
210
211     return &tunnels[prio][h];
212 }

```

Figure 525: GRE Tunnels - Hash Bucket: `linux/net/ipv4/ip_gre.c`

```

214 static void ipgre_tunnel_link(struct ip_tunnel *t)
215 {
216     struct ip_tunnel **tp = ipgre_bucket(t);
217
218     t->next = *tp;
219     write_lock_bh(&ipgre_lock);
220     *tp = t;
221     write_unlock_bh(&ipgre_lock);
222 }
223
224 static void ipgre_tunnel_unlink(struct ip_tunnel *t)
225 {
226     struct ip_tunnel **tp;
227
228     for (tp = ipgre_bucket(t); *tp; tp = &(*tp)->next) {
229         if (t == *tp) {
230             write_lock_bh(&ipgre_lock);
231             *tp = t->next;
232             write_unlock_bh(&ipgre_lock);
233             break;
234         }
235     }
236 }

```

Figure 526: GRE Tunnels - link/unlink hash entry: `linux/net/ipv4/ipgre.c`

```

238 static struct ip_tunnel * ipgre_tunnel_locate (struct ip_tunnel_parm *parms, int create)
239 {
240     u32 remote = parms->iph.daddr;
241     u32 local = parms->iph.saddr;
242     u32 key = parms->i_key;
243     struct ip_tunnel *t, **tp, *nt;
244     struct net_device *dev;
245     unsigned h = HASH(key);
246     int prio = 0;
247
248     if (local)
249         prio |= 1;
250     if (remote && !MULTICAST(remote)) {
251         prio |= 2;
252         h ^= HASH(remote);
253     }
254     for (tp = &tunnels[prio][h]; (t = *tp) != NULL; tp = &t->next) {
255         if (local == t->parms.iph.saddr && remote == t->parms.iph.daddr) {
256             if (key == t->parms.i_key)
257                 return t;
258         }
259     }
260     if (!create)
261         return NULL;
262
263     MOD_INC_USE_COUNT;

```

Figure 527: GRE Tunnels - locate hash entry: `linux/net/ipv4/ip_gre.c`

```

265     if (dev == NULL) {
266         MOD_DEC_USE_COUNT;
267         return NULL;
268     }
269     memset(dev, 0, sizeof(*dev) + sizeof(*t));
270     dev->priv = (void*)(dev+1);
271     nt = (struct ip_tunnel*)dev->priv;
272     nt->dev = dev;
273     dev->init = ipgre_tunnel_init;
274     dev->features |= NETIFF_DYNALLOC;
275     memcpy(&nt->parms, parms, sizeof(*parms));
276     strcpy(dev->name, nt->parms.name);
277     if (dev->name[0] == 0) {
278         int i;
279         for (i=1; i<100; i++) {
280             sprintf(dev->name, "gre%d", i);
281             if (_dev_get_by_name(dev->name) == NULL)
282                 break;
283         }
284         if (i==100)
285             goto failed;
286         memcpy(parms->name, dev->name, IFNAMSIZ);
287     }
288     if (register_netdevice (dev) < 0)
289         goto failed;
290
291     dev_hold(dev);
292     ipgre_tunnel_link (nt);
293     /* Do not decrement MOD_USE_COUNT here. */
294     return nt;
295
296 failed:
297     kfree(dev);
298     MOD_DEC_USE_COUNT;
299     return NULL;
300 }
301
302 static void ipgre_tunnel_destructor (struct net_device *dev)
303 {
304     if (dev != &ipgre_fb_tunnel_dev) {
305         MOD_DEC_USE_COUNT;
306     }
307 }
308
309 static void ipgre_tunnel_uninit (struct net_device *dev)
310 {
311     ipgre_tunnel_unlink((struct ip_tunnel*)dev->priv);
312     dev_put(dev);
313 }

```

Figure 528: GRE Tunnels - uninitialise and destory entry:
linux/net/ipv4/ip_gre.c

```

562 int ipgre_rcv(struct sk_buff *skb)
563 {
564     struct iphdr *iph;
565     u8     *h;
566     u16    flags;
567     u16    csum = 0;
568     u32    key = 0;
569     u32    seqno = 0;
570     struct ip_tunnel *tunnel;
571     int    offset = 4;
572
573     if (!pskb_may_pull(skb, 16))
574         goto drop_nolock;
575
576     iph = skb->nh.iph;
577     h = skb->data;
578     flags = *(u16*)h;
579
580     if (flags & (GRE_CSUM|GRE_KEY|GRE_ROUTING|GRE_SEQ|GRE_VERSION))
581     {
582         /* - Version must be 0.
583            - We do not support routing headers.
584            */
585         if (flags & (GRE_VERSION|GRE_ROUTING))
586             goto drop_nolock;
587
588         if (flags & GRE_CSUM) {
589             if (skb->ip_summed == CHECKSUM_HW) {
590                 csum = (u16)csum_fold(skb->csum);
591                 if (csum)
592                     skb->ip_summed = CHECKSUM_NONE;
593             }
594             if (skb->ip_summed == CHECKSUM_NONE) {
595                 skb->csum = skb_checksum(skb, 0, skb->len, 0);

```

Figure 529: GRE Tunnels- Receive Case: linux/net/ipv4/ip_gre.c


```

595         skb->ip_summed = CHECKSUM_HW;
596         csum = (u16)csum_fold(skb->csum);
597     }
598     offset += 4;
599 }
600 if ( flags & GRE_KEY ) {
601     key = *(u32*)(h + offset);
602     offset += 4;
603 }
604 if ( flags & GRE_SEQ ) {
605     seqno = ntohl(*(u32*)(h + offset));
606     offset += 4;
607 }
608 }
609
610 read_lock(&ipgre_lock);
611 if ((tunnel = ipgre_tunnel_lookup(iph->saddr, iph->daddr, key)) != NULL) {
612     skb->mac.raw = skb->nh.raw;
613     skb->nh.raw = _pskb_pull(skb, offset);
614     memset(&(IPCB(skb)->opt), 0, sizeof(struct ip_options));
615     if (skb->ip_summed == CHECKSUM_HW)
616         skb->csum = csum_sub(skb->csum,
617                             csum_partial(skb->mac.raw, skb->nh.raw -
618                                           skb->mac.raw, 0));

```

Figure 530: GRE Tunnels- Receive Case: linux/net/ipv4/ipgre.c

```

618     skb->protocol = *(u16*)(h + 2);
619     skb->pkt_type = PACKET_HOST;
620 #ifdef CONFIG_NET_IPGRE_BROADCAST
621     if (MULTICAST(iph->daddr)) {
622         /* Looped back packet, drop it! */
623         if (((struct rtable*)skb->dst)->key.iif == 0)
624             goto drop;
625         tunnel->stat.multicast++;
626         skb->pkt_type = PACKET_BROADCAST;
627     }
628 #endif
629
630     if ((( flags&GRE_CSUM) && csum) ||
631         (!( flags&GRE_CSUM) && tunnel->parms.i_flags&GRE_CSUM)) {
632         tunnel->stat.rx_crc_errors++;
633         tunnel->stat.rx_errors++;
634         goto drop;
635     }
636     if (tunnel->parms.i_flags&GRE_SEQ) {
637         if (!( flags&GRE_SEQ) ||
638             (tunnel->i_seqno && (s32)(seqno - tunnel->i_seqno) < 0)) {
639             tunnel->stat.rx_fifo_errors++;
640             tunnel->stat.rx_errors++;
641             goto drop;
642         }
643         tunnel->i_seqno = seqno + 1;
644     }
645     tunnel->stat.rx_packets++;
646     tunnel->stat.rx_bytes += skb->len;
647     skb->dev = tunnel->dev;
648     dst_release(skb->dst);
649     skb->dst = NULL;
650 #ifdef CONFIG_NETFILTER
651     nf_contrack_put(skb->nfct);

```

Figure 531: GRE Tunnels- Receive Case: linux/net/ipv4/ip_gre.c

```
653 #ifdef CONFIG_NETFILTER_DEBUG
654     skb->nf_debug = 0;
655 #endif
656 #endif
657     ipgre_ecn_decapsulate(iph, skb);
```

Figure 532: GRE Tunnels - Tx Wrapper: `linux/net/ipv4/ip_gre.c`

```

659         read_unlock(&ipgre_lock);
660         return(0);
661     }
662     icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PROT_UNREACH, 0);
663
664 drop:
665     read_unlock(&ipgre_lock);
666 drop_nolock:
667     kfree_skb(skb);
668     return(0);
669 }
670
671 /* Need this wrapper because NF_HOOK takes the function address */
672 static inline int do_ip_send(struct sk_buff *skb)
673 {
674     return ip_send(skb);
675 }
676
677 static int ipgre_tunnel_xmit(struct sk_buff *skb, struct net_device *dev)
678 {
679     struct ip_tunnel *tunnel = (struct ip_tunnel*)dev->priv;
680     struct net_device_stats *stats = &tunnel->stat;
681     struct iphdr *old_iph = skb->nh.iph;
682     struct iphdr *tiph;
683     u8    tos;
684     u16   df;
685     struct rtable *rt;           /* Route to the other host */
686     struct net_device *tdev;     /* Device to other host */
687     struct iphdr *iph;          /* Our new IP header */
688     int    max_headroom;        /* The extra header space needed */
689     int    gre_hlen;
690     u32    dst;
691     int    mtu;
692
693     if (tunnel->recursion++) {
694         tunnel->stat.collisions++;

```

Figure 533: GRE Tunnels - Transmit Case: linux/net/ipv4/ipgre.c

```

696     }
697
698     if (dev->hard_header) {
699         gre_hlen = 0;
700         tiph = (struct iphdr*)skb->data;
701     } else {
702         gre_hlen = tunnel->hlen;
703         tiph = &tunnel->parms.iph;
704     }
705
706     if ((dst = tiph->daddr) == 0) {
707         /* NBMA tunnel */
708
709         if (skb->dst == NULL) {
710             tunnel->stat.tx_fifo_errors++;
711             goto tx_error;
712         }
713
714         if (skb->protocol == __constant_htons(ETH_P_IP)) {
715             rt = (struct rtable*)skb->dst;
716             if ((dst = rt->rt_gateway) == 0)
717                 goto tx_error_icmp;
718         }
719 #ifdef CONFIG_IPV6
720         else if (skb->protocol == __constant_htons(ETH_P_IPV6)) {
721             struct in6_addr *addr6;
722             int addr_type;
723             struct neighbour *neigh = skb->dst->neighbour;
724
725             if (neigh == NULL)
726                 goto tx_error;
727
728             addr6 = (struct in6_addr*)&neigh->primary_key;
729             addr_type = ipv6_addr_type(addr6);
730
731             if (addr_type == IPV6_ADDR_ANY) {
732                 addr6 = &skb->nh.ipv6h->daddr;
733                 addr_type = ipv6_addr_type(addr6);
734             }
735
736             if ((addr_type & IPV6_ADDR_COMPATv4) == 0)
737                 goto tx_error_icmp;
738
739             dst = addr6->s6_addr32[3];
740         }
741 #endif
742         else
743             goto tx_error;
744     }

```

Figure 534: GRE Tunnels - Transmit Case: linux/net/ipv4/ip-gre.c

```

747     if (tos&1) {
748         if (skb->protocol == __constant_htons(ETH_P_IP))
749             tos = old_iph->tos;
750         tos &= ~1;
751     }
752
753     if (ip_route_output(&rt, dst, tiph->saddr, RT_TOS(tos), tunnel->parms.link)) {
754         tunnel->stat.tx_carrier_errors++;
755         goto tx_error;
756     }
757     tdev = rt->u.dst.dev;
758
759     if (tdev == dev) {
760         ip_rt_put(rt);
761         tunnel->stat.collisions++;
762         goto tx_error;
763     }
764
765     df = tiph->frag_off;
766     mtu = rt->u.dst.pmtu - tunnel->hlen;
767
768     if (skb->protocol == __constant_htons(ETH_P_IP)) {
769         if (skb->dst && mtu < skb->dst->pmtu && mtu >= 68)
770             skb->dst->pmtu = mtu;
771
772         df |= (old_iph->frag_off&__constant_htons(IP_DF));
773
774         if ((old_iph->frag_off&__constant_htons(IP_DF)) &&
775             mtu < ntohs(old_iph->tot_len)) {
776             icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED
777                 , htons(mtu));
778             ip_rt_put(rt);
779             goto tx_error;
780         }
781     }
782 #ifdef CONFIG_IPV6

```

Figure 535: GRE Tunnels - Transmit Case: linux/net/ipv4/ip_gre.c

```

783         struct rt6_info *rt6 = (struct rt6_info*)skb->dst;
784
785         if (rt6 && mtu < rt6->u.dst.pmtu && mtu >= IPV6_MIN_MTU) {
786             if ((tunnel->parms.iph.daddr && !MULTICAST(tunnel->parms
787                 .iph.daddr)) ||
788                 rt6->rt6i.dst.plen == 128) {
789                 rt6->rt6i.flags |= RTF_MODIFIED;
790                 skb->dst->pmtu = mtu;
791             }
792         }
793
794         if (mtu >= IPV6_MIN_MTU && mtu < skb->len - tunnel->hlen +
795             gre_hlen) {
796             icmpv6_send(skb, ICMPV6_PKT_TOOBIG, 0, mtu, dev);
797             ip_rt_put(rt);
798             goto tx_error;
799         }
800     }
801 #endif
802
803     if (tunnel->err_count > 0) {
804         if ( jiffies - tunnel->err_time < IPTUNNEL_ERR_TIMEO) {
805             tunnel->err_count--;
806
807             dst_link_failure(skb);
808         } else
809             tunnel->err_count = 0;
810     }
811
812     skb->h.raw = skb->nh.raw;
813
814     max_headroom = ((tdev->hard_header_len+15)&~15)+ gre_hlen;
815
816     if (skb_headroom(skb) < max_headroom || skb_cloned(skb) || skb_shared(skb)) {

```

Figure 536: GRE Tunnels - Transmit Case: linux/net/ipv4/ip_gre.c

```

816         if (!new_skb) {
817             ip_rt_put(rt);
818             stats->tx_dropped++;
819             dev_kfree_skb(skb);
820             tunnel->recursion--;
821             return 0;
822         }
823         if (skb->sk)
824             skb_set_owner_w(new_skb, skb->sk);
825         dev_kfree_skb(skb);
826         skb = new_skb;
827     }
828
829     skb->nh.raw = skb_push(skb, gre_hlen);
830     memset(&(IPCB(skb)->opt), 0, sizeof(IPCB(skb)->opt));
831     dst_release(skb->dst);
832     skb->dst = &rt->u.dst;
833
834     /*
835      *   Push down and install the IPIP header.
836      */
837
838     iph          =      skb->nh.iph;
839     iph->version =      4;
840     iph->ihl     =      sizeof(struct iphdr) >> 2;
841     iph->frag_off =      df;

```

Figure 537: GRE Tunnels - Transmit Case: `linux/net/ipv4/ip_gre.c`


```

843     iph->tos           =     ipgre_ecn_encapsulate(tos, old_iph, skb);
844     iph->daddr        =     rt->rt_dst;
845     iph->saddr        =     rt->rt_src;
846
847     if ((iph->ttl = tiph->ttl) == 0) {
848         if (skb->protocol == __constant_htons(ETH_P_IP))
849             iph->ttl = old_iph->ttl;
850 #ifdef CONFIG_IPV6
851         else if (skb->protocol == __constant_htons(ETH_P_IPV6))
852             iph->ttl = ((struct ipv6hdr*)old_iph)->hop_limit;
853 #endif
854         else
855             iph->ttl = sysctl_ip_default_ttl;
856     }
857
858     ((u16*)(iph+1))[0] = tunnel->parms.o_flags;
859     ((u16*)(iph+1))[1] = skb->protocol;
860
861     if (tunnel->parms.o_flags & (GRE_KEY|GRE_CSUM|GRE_SEQ)) {
862         u32 *ptr = (u32*)((u8*)iph) + tunnel->hlen - 4;
863
864         if (tunnel->parms.o_flags & GRE_SEQ) {
865             ++tunnel->o_seqno;
866             *ptr = htonl(tunnel->o_seqno);
867             ptr--;
868         }
869         if (tunnel->parms.o_flags & GRE_KEY) {
870             *ptr = tunnel->parms.o_key;
871             ptr--;
872         }
873         if (tunnel->parms.o_flags & GRE_CSUM) {
874             *ptr = 0;
875             *((u16*)ptr) = ip_compute_csum((void*)(iph+1), skb->len -
876                 sizeof(struct iphdr));
877         }
878     }
879 #ifdef CONFIG_NETFILTER
880     nf_conntrack_put(skb->nfct);
881     skb->nfct = NULL;

```

Figure 538: GRE Tunnels - Transmit Case: `linux/net/ipv4/ip_gre.c`

```

316 void ipgre_err(struct sk_buff *skb, u32 info)
317 {
318 #ifndef L_WISH_WORLD_WERE_PERFECT
319
320 /* It is not :( All the routers (except for Linux) return only
321    8 bytes of packet payload. It means, that precise relaying of
322    ICMP in the real Internet is absolutely infeasible.
323
324    Moreover, Cisco "wise men" put GRE key to the third word
325    in GRE header. It makes impossible maintaining even soft state for keyed
326    GRE tunnels with enabled checksum. Tell them "thank you".
327
328    Well, I wonder, rfc1812 was written by Cisco employee,
329    what the hell these idiots break standards established
330    by themselves???
331 */
332
333     struct iphdr *iph = (struct iphdr*)skb->data;
334     u16 *p = (u16*)(skb->data+(iph->ihl<<2));
335     int grehlen = (iph->ihl<<2) + 4;
336     int type = skb->h.icmph->type;
337     int code = skb->h.icmph->code;
338     struct ip_tunnel *t;
339     u16 flags;
340
341     flags = p[0];
342     if (flags & (GRE_CSUM|GRE_KEY|GRE_SEQ|GRE_ROUTING|GRE_VERSION))
343     {
344         if (flags & (GRE_VERSION|GRE_ROUTING))
345             return;
346         if (flags & GRE_KEY) {
347             grehlen += 4;
348             if (flags & GRE_CSUM)
349                 grehlen += 4;
350         }
351     }
352     /* If only 8 bytes returned, keyed message will be dropped here */
353     if (skb_headlen(skb) < grehlen)
354         return;

```

Figure 539: GRE Tunnels - Error/ICMP cases: linux/net/ipv4/ip_gre.c

```

356     switch (type) {
357     default:
358     case ICMP_PARAMETERPROB:
359         return;
360
361     case ICMP_DEST_UNREACH:
362         switch (code) {
363         case ICMP_SR_FAILED:
364         case ICMP_PORT_UNREACH:
365             /* Impossible event. */
366             return;
367         case ICMP_FRAG_NEEDED:
368             /* Soft state for pmtu is maintained by IP core. */
369             return;
370         default:
371             /* All others are translated to HOST_UNREACH.
372              * rfc2003 contains "deep thoughts" about NET_UNREACH,
373              * I believe they are just ether pollution. --ANK
374              */
375             break;
376         }
377         break;
378     case ICMP_TIME_EXCEEDED:
379         if (code != ICMP_EXC_TTL)
380             return;
381         break;
382     }

```

Figure 540: GRE Tunnels - Error/ICMP cases: linux/net/ipv4/ip_gre.c

```

384     read_lock(&ipgre_lock);
385     t = ipgre_tunnel_lookup(iph->daddr, iph->saddr, (flags&GRE_KEY) ? *(((u32*)
386     p) + (grehlen>>2) - 1) : 0);
387     if (t == NULL || t->parms.iph.daddr == 0 || MULTICAST(t->parms.iph.
388     daddr))
389         goto out;
390     if (t->parms.iph.ttl == 0 && type == ICMP_TIME_EXCEEDED)
391         goto out;
392     if ( jiffies - t->err_time < IPTUNNEL_ERR_TIMEO)
393         t->err_count++;
394     else
395         t->err_count = 1;
396     t->err_time = jiffies;
397 out:
398     read_unlock(&ipgre_lock);
399     return;
400 #else
401     struct iphdr *iph = (struct iphdr*)dp;
402     struct iphdr *eiph;
403     u16 *p = (u16*)(dp+(iph->ihl<<2));
404     int type = skb->h.icmph->type;
405     int code = skb->h.icmph->code;
406     int rel_type = 0;
407     int rel_code = 0;
408     int rel_info = 0;
409     u16 flags;
410     int grehlen = (iph->ihl<<2) + 4;
411     struct sk_buff *skb2;
412     struct rtable *rt;

```

Figure 541: GRE Tunnels - Error/ICMP cases: linux/net/ipv4/ip_gre.c

```

414     if (p[1] != __constant_htons(ETH_P_IP))
415         return;
416
417     flags = p[0];
418     if (flags &(GRE_CSUM|GRE_KEY|GRE_SEQ|GRE_ROUTING|GRE_VERSION))
419     {
420         if (flags &(GRE_VERSION|GRE_ROUTING))
421             return;
422         if (flags &GRE_CSUM)
423             grehlen += 4;
424         if (flags &GRE_KEY)
425             grehlen += 4;
426         if (flags &GRE_SEQ)
427             grehlen += 4;
428     }
429     if (len < grehlen + sizeof(struct iphdr))
430         return;
431     eiph = (struct iphdr*)(dp + grehlen);

```

Figure 542: GRE Tunnels - Error/ICMP cases: `linux/net/ipv4/ip_gre.c`

```

432     switch (type) {
433     default:
434         return;
435     case ICMP_PARAMETERPROB:
436         if (skb->h.icmph->un.gateway < (iph->ihl<<2))
437             return;
438
439         /* So... This guy found something strange INSIDE encapsulated
440            packet. Well, he is fool, but what can we do ?
441            */
442         rel_type = ICMP_PARAMETERPROB;
443         rel_info = skb->h.icmph->un.gateway - grehlen;
444         break;
445
446     case ICMP_DEST_UNREACH:
447         switch (code) {
448         case ICMP_SR_FAILED:
449         case ICMP_PORT_UNREACH:
450             /* Impossible event. */
451             return;
452         case ICMP_FRAG_NEEDED:
453             /* And it is the only really necessary thing :- ) */
454             rel_info = ntohs(skb->h.icmph->un.frag.mtu);
455             if ( rel_info < grehlen+68)
456                 return;
457             rel_info -= grehlen;
458             /* BSD 4.2 MORE DOES NOT EXIST IN NATURE. */
459             if ( rel_info > ntohs(eiph->tot_len))
460                 return;
461             break;
462         default:
463             /* All others are translated to HOST_UNREACH.
464                rfc2003 contains "deep thoughts" about NET_UNREACH,
465                I believe, it is just ether pollution. --ANK
466                */
467             rel_type = ICMP_DEST_UNREACH;
468             rel_code = ICMP_HOST_UNREACH;
469             break;
470         }
471         break;
472     case ICMP_TIME_EXCEEDED:
473         if (code != ICMP_EXC_TTL)
474             return;
475         break;
476 }

```

Figure 543: GRE Tunnels - Error/ICMP cases: linux/net/ipv4/ip_gre.c

```

478     /* Prepare fake skb to feed it to icmp_send */
479     skb2 = skb_clone(skb, GFP_ATOMIC);
480     if (skb2 == NULL)
481         return;
482     dst_release(skb2->dst);
483     skb2->dst = NULL;
484     skb_pull(skb2, skb->data - (u8*)eiph);
485     skb2->nh.raw = skb2->data;
486
487     /* Try to guess incoming interface */
488     if (ip_route_output(&rt, eiph->saddr, 0, RT_TOS(eiph->tos), 0)) {
489         kfree_skb(skb2);
490         return;
491     }
492     skb2->dev = rt->u.dst.dev;

```

Figure 544: GRE Tunnels - Error/ICMP cases: `linux/net/ipv4/ip_gre.c`

```

494     /* route "incoming" packet */
495     if (rt->rt_flags&RTCF_LOCAL) {
496         ip_rt_put(rt);
497         rt = NULL;
498         if (ip_route_output(&rt, eiph->daddr, eiph->saddr, eiph->tos, 0) ||
499             rt->u.dst.dev->type != ARPHRD_IPGRE) {
500             ip_rt_put(rt);
501             kfree_skb(skb2);
502             return;
503         }
504     } else {
505         ip_rt_put(rt);
506         if (ip_route_input(skb2, eiph->daddr, eiph->saddr, eiph->tos, skb2->
507             dev) ||
508             skb2->dst->dev->type != ARPHRD_IPGRE) {
509             kfree_skb(skb2);
510             return;
511         }
512     }
513     /* change mtu on this route */
514     if (type == ICMP_DEST_UNREACH && code == ICMP_FRAG_NEEDED) {
515         if (rel_info > skb2->dst->pmtu) {
516             kfree_skb(skb2);
517             return;
518         }
519         skb2->dst->pmtu = rel_info;
520         rel_info = htonl(rel_info);
521     } else if (type == ICMP_TIME_EXCEEDED) {
522         struct ip_tunnel *t = (struct ip_tunnel*)skb2->dev->priv;
523         if (t->parms.iph.ttl) {
524             rel_type = ICMP_DEST_UNREACH;
525             rel_code = ICMP_HOST_UNREACH;
526         }
527     }
528
529     icmp_send(skb2, rel_type, rel_code, rel_info);
530     kfree_skb(skb2);
531 #endif
532 }

```

Figure 545: GRE Tunnels - Error/ICMP cases: linux/net/ipv4/ip_gre.c


```

534 static inline void ipgre_ecn_decapsulate(struct iphdr *iph, struct sk_buff *skb)
535 {
536     if (INET_ECN_is_ce(iph->tos)) {
537         if (skb->protocol == __constant_htons(ETH_P_IP)) {
538             if (INET_ECN_is_not_ce(skb->nh.iph->tos))
539                 IP_ECN_set_ce(skb->nh.iph);
540         } else if (skb->protocol == __constant_htons(ETH_P_IPV6)) {
541             if (INET_ECN_is_not_ce(ip6_get_dsfield(skb->nh.ipv6h)))
542                 IP6_ECN_set_ce(skb->nh.ipv6h);
543         }
544     }
545 }
546
547 static inline u8
548 ipgre_ecn_encapsulate(u8 tos, struct iphdr *old_iph, struct sk_buff *skb)
549 {
550     #ifdef CONFIG_INET_ECN
551         u8 inner = 0;
552         if (skb->protocol == __constant_htons(ETH_P_IP))
553             inner = old_iph->tos;
554         else if (skb->protocol == __constant_htons(ETH_P_IPV6))
555             inner = ip6_get_dsfield((struct ipv6hdr*)old_iph);
556         return INET_ECN_encapsulate(tos, inner);
557     #else
558         return tos;
559     #endif
560 }

```

Figure 546: GRE Tunnels - ECN Handling: `linux/net/ipv4/ipgre.c`

```

883     skb->nf_debug = 0;
884 #endif
885 #endif
886
887     IPTUNNEL_XMIT();
888     tunnel->recursion--;
889     return 0;
890
891 tx_error_icmp:
892     dst_link_failure(skb);
893
894 tx_error:
895     stats->tx_errors++;
896     dev_kfree_skb(skb);
897     tunnel->recursion--;
898     return 0;
899 }
900
901 static int
902 ipgre_tunnel_ioctl (struct net_device *dev, struct ifreq *ifr, int cmd)
903 {
904     int err = 0;
905     struct ip_tunnel_parm p;
906     struct ip_tunnel *t;

```

Figure 547: GRE Tunnels - ioctl entry handler: `linux/net/ipv4/ip_gre.c`

```

909
910     switch (cmd) {
911     case SIOCGETTUNNEL:
912         t = NULL;
913         if (dev == &ipgre_fb_tunnel_dev) {
914             if (copy_from_user(&p, ifr->ifru.ifru_data, sizeof(p))) {
915                 err = -EFAULT;
916                 break;
917             }
918             t = ipgre_tunnel_locate(&p, 0);
919         }
920         if (t == NULL)
921             t = (struct ip_tunnel*)dev->priv;
922         memcpy(&p, &t->parms, sizeof(p));
923         if (copy_to_user(ifr->ifru.ifru_data, &p, sizeof(p)))
924             err = -EFAULT;
925         break;
926
927     case SIOCADDTUNNEL:
928     case SIOCCHGTUNNEL:
929         err = -EPERM;
930         if (!capable(CAP_NET_ADMIN))
931             goto done;

```

Figure 548: GRE Tunnels - ioctl entry handler: `linux/net/ipv4/ip_gre.c`

```

934     if (copy_from_user(&p, ifr->ifru.ifru_data, sizeof(p)))
935         goto done;
936
937     err = -EINVAL;
938     if (p.iph.version != 4 || p.iph.protocol != IPPROTO_GRE ||
939         p.iph.ihl != 5 || (p.iph.frag_off & _constant_htons(~IP_DF)) ||
940         ((p.i_flags | p.o_flags) & (GRE_VERSION | GRE_ROUTING)))
941         goto done;
942     if (p.iph.ttl)
943         p.iph.frag_off |= _constant_htons(IP_DF);
944
945     if (!(p.i_flags & GRE_KEY))
946         p.i_key = 0;
947     if (!(p.o_flags & GRE_KEY))
948         p.o_key = 0;
949
950     t = ipgre_tunnel_locate(&p, cmd == SIOCADDTUNNEL);
951
952     if (dev != &ipgre_fb_tunnel_dev && cmd == SIOCCHGTUNNEL &&
953         t != &ipgre_fb_tunnel) {
954         if (t != NULL) {
955             if (t->dev != dev) {
956                 err = -EEXIST;
957                 break;
958             }
959         } else {
960             unsigned nflags=0;
961
962             t = (struct ip_tunnel*)dev->priv;
963
964             if (MULTICAST(p.iph.daddr))
965                 nflags = IFF_BROADCAST;

```

Figure 549: GRE Tunnels - ioctl entry handler: `linux/net/ipv4/ip_gre.c`

```

967         nflags = IFF_POINTOPOINT;
968
969         if ((dev->flags^nflags)&(IFF_POINTOPOINT|
970             IFF_BROADCAST)) {
971             err = -EINVAL;
972             break;
973         }
974         ipgre_tunnel_unlink(t);
975         t->parms.iph.saddr = p.iph.saddr;
976         t->parms.iph.daddr = p.iph.daddr;
977         t->parms.i_key = p.i_key;
978         t->parms.o_key = p.o_key;
979         memcpy(dev->dev_addr, &p.iph.saddr, 4);
980         memcpy(dev->broadcast, &p.iph.daddr, 4);
981         ipgre_tunnel_link(t);
982         netdev_state_change(dev);
983     }
984
985     if (t) {
986         err = 0;
987         if (cmd == SIOCCHGTUNNEL) {
988             t->parms.iph.ttl = p.iph.ttl;
989             t->parms.iph.tos = p.iph.tos;
990             t->parms.iph.frag_off = p.iph.frag_off;
991         }
992         if (copy_to_user(&ifr->ifru.ifru_data, &t->parms, sizeof(p)))
993             err = -EFAULT;
994     } else
995         err = (cmd == SIOCADDTUNNEL ? -ENOBUFS : -ENOENT);
996     break;
997
998     case SIOCDELTUNNEL:
999         err = -EPERM;
1000         if (!capable(CAP_NET_ADMIN))
1001             goto done;
1002
1003         if (dev == &ipgre_fb_tunnel_dev) {
1004             err = -EFAULT;
1005             if (copy_from_user(&p, ifr->ifru.ifru_data, sizeof(p)))
1006                 goto done;

```

Figure 550: GRE Tunnels - ioctl entry handler: linux/net/ipv4/ip_gre.c

```

1022     MOD_DEC_USE_COUNT;
1023     return err;
1024 }
1025
1026 static struct net_device_stats *ipgre_tunnel_get_stats(struct net_device *dev)
1027 {
1028     return &(((struct ip_tunnel*)dev->priv)->stat);
1029 }
1030
1031 static int ipgre_tunnel_change_mtu(struct net_device *dev, int new_mtu)
1032 {
1033     struct ip_tunnel *tunnel = (struct ip_tunnel*)dev->priv;
1034     if (new_mtu < 68 || new_mtu > 0xFFF8 - tunnel->hlen)
1035         return -EINVAL;
1036     dev->mtu = new_mtu;
1037     return 0;
1038 }
1039
1040 #ifdef CONFIG_NET_IPGRE_BROADCAST
1041 /* Nice toy. Unfortunately, useless in real life :-)
1042    It allows to construct virtual multiprotocol broadcast "LAN"
1043    over the Internet, provided multicast routing is tuned.
1044
1045
1046    I have no idea was this bicycle invented before me,
1047    so that I had to set ARPHRD_IPGRE to a random value.
1048    I have an impression, that Cisco could make something similar,
1049    but this feature is apparently missing in IOS<=11.2(8).

```

Figure 551: GRE Tunnels - Broadcast Support!: linux/net/ipv4/ip_gre.c

```

1052 with broadcast 224.66.66.66. If you have access to mbone, play with me :-)
1053
1054 ping -t 255 224.66.66.66
1055
1056 If nobody answers, mbone does not work.
1057
1058 ip tunnel add Universe mode gre remote 224.66.66.66 local <Your_real_addr> ttl 255
1059 ip addr add 10.66.66.<somewhat>/24 dev Universe
1060 ifconfig Universe up
1061 ifconfig Universe add fe80::<Your_real_addr>/10
1062 ifconfig Universe add fec0:6666:6666::<Your_real_addr>/96
1063 ftp 10.66.66.66
1064 ...
1065 ftp fec0 :6666:6666::193.233.7.65
1066 ...
1067
1068 */
1069
1070 static int ipgre_header(struct sk_buff *skb, struct net_device *dev, unsigned short type,
1071                       void *daddr, void *saddr, unsigned len)
1072 {
1073     struct ip_tunnel *t = (struct ip_tunnel*)dev->priv;
1074     struct iphdr *iph = (struct iphdr *)skb_push(skb, t->hlen);
1075     u16 *p = (u16*)(iph+1);
1076
1077     memcpy(iph, &t->parms.iph, sizeof(struct iphdr));
1078     p[0] = t->parms.o_flags;

```

Figure 552: GRE Tunnels - Broadcast Support!: `linux/net/ipv4/ip-gre.c`

```

1080
1081     /*
1082     *      Set the source hardware address.
1083     */
1084
1085     if (saddr)
1086         memcpy(&iph->saddr, saddr, 4);
1087
1088     if (daddr) {
1089         memcpy(&iph->daddr, daddr, 4);
1090         return t->hlen;
1091     }
1092     if (iph->daddr && !MULTICAST(iph->daddr))
1093         return t->hlen;
1094
1095     return -t->hlen;
1096 }
1097
1098 static int ipgre_open(struct net_device *dev)
1099 {
1100     struct ip_tunnel *t = (struct ip_tunnel*)dev->priv;
1101
1102     MOD_INC_USE_COUNT;
1103     if (MULTICAST(t->parms.iph.daddr)) {
1104         struct rtable *rt;
1105         if (ip_route_output(&rt, t->parms.iph.daddr,
1106                             t->parms.iph.saddr, RT_TOS(t->parms.iph.tos),
1107                             t->parms.link)) {
1108             MOD_DEC_USE_COUNT;
1109             return -EADDRNOTAVAIL;
1110         }
1111         dev = rt->u.dst.dev;
1112         ip_rt_put(rt);
1113         if (_in_dev_get(dev) == NULL) {
1114             MOD_DEC_USE_COUNT;
1115             return -EADDRNOTAVAIL;
1116         }
1117         t->mink = dev->ifindex;
1118         ip_mc_inc_group(_in_dev_get(dev), t->parms.iph.daddr);
1119     }

```

Figure 553: GRE Tunnels - Broadcast Support!: `linux/net/ipv4/ipgre.c`


```
1008         if ((t = ipgre_tunnel_locate(&p, 0)) == NULL)
1009             goto done;
1010         err = -EPERM;
1011         if (t == &ipgre_fb_tunnel)
1012             goto done;
1013     }
1014     err = unregister_netdevice(dev);
1015     break;
1016
1017 default :
1018     err = -EINVAL;
1019 }
```

Figure 554: GRE Tunnels - stats: linux/net/ipv4/ip_gre.c

```

1121 }
1122
1123 static int ipgre_close (struct net_device *dev)
1124 {
1125     struct ip_tunnel *t = (struct ip_tunnel*)dev->priv;
1126     if (MULTICAST(t->parms.iph.daddr) && t->mink) {
1127         struct in_device *in_dev = inetdev_by_index(t->mink);
1128         if (in_dev) {
1129             ip_mc_dec_group(in_dev, t->parms.iph.daddr);
1130             in_dev_put(in_dev);
1131         }
1132     }
1133     MOD_DEC_USE_COUNT;
1134     return 0;
1135 }
1136
1137 #endif
1138
1139 static void ipgre_tunnel_init_gen (struct net_device *dev)
1140 {
1141     struct ip_tunnel *t = (struct ip_tunnel*)dev->priv;

```

Figure 555: GRE Tunnels - general initialisation: `linux/net/ipv4/ip_gre.c`

```

1144     dev->destructor      = ipgre_tunnel_destructor;
1145     dev->hard_start_xmit = ipgre_tunnel_xmit;
1146     dev->get_stats       = ipgre_tunnel_get_stats;
1147     dev->do_ioctl        = ipgre_tunnel_ioctl;
1148     dev->change_mtu      = ipgre_tunnel_change_mtu;
1149
1150     dev->type             = ARPHRD_IPGRE;
1151     dev->hard_header_len = LL_MAX_HEADER + sizeof(struct iphdr) + 4;
1152     dev->mtu             = 1500 - sizeof(struct iphdr) - 4;
1153     dev->flags           = IFF_NOARP;
1154     dev->iflink          = 0;
1155     dev->addr_len        = 4;
1156     memcpy(dev->dev_addr, &t->parms.iph.saddr, 4);
1157     memcpy(dev->broadcast, &t->parms.iph.daddr, 4);
1158 }
1159
1160 static int ipgre_tunnel_init (struct net_device *dev)
1161 {
1162     struct net_device *tdev = NULL;
1163     struct ip_tunnel *tunnel;
1164     struct iphdr *iph;
1165     int hlen = LL_MAX_HEADER;
1166     int mtu = 1500;
1167     int addend = sizeof(struct iphdr) + 4;

```

Figure 556: GRE Tunnels - tunnel initialisation: `linux/net/ipv4/ip-gre.c`

```

1169     tunnel = (struct ip_tunnel*)dev->priv;
1170     iph = &tunnel->parms.iph;
1171
1172     ipgre_tunnel_init_gen (dev);
1173
1174     /* Guess output device to choose reasonable mtu and hard_header_len */
1175
1176     if (iph->daddr) {
1177         struct rtable *rt;
1178         if (!ip_route_output(&rt, iph->daddr, iph->saddr, RT_TOS(iph->tos),
1179                             tunnel->parms.link)) {
1179             tdev = rt->u.dst.dev;
1180             ip_rt_put (rt);
1181         }
1182
1183         dev->flags |= IFF_POINTOPOINT;
1184
1185 #ifdef CONFIG_NET_IPGRE_BROADCAST
1186         if (MULTICAST(iph->daddr)) {
1187             if (!iph->saddr)
1188                 return -EINVAL;
1189             dev->flags = IFF_BROADCAST;
1190             dev->hard_header = ipgre_header;
1191             dev->open = ipgre_open;
1192             dev->stop = ipgre_close;
1193         }
1194 #endif
1195     }
1196
1197     if (!tdev && tunnel->parms.link)
1198         tdev = _dev_get_by_index(tunnel->parms.link);
1199
1200     if (tdev) {
1201         hlen = tdev->hard_header_len;
1202         mtu = tdev->mtu;
1203     }

```

Figure 557: GRE Tunnels - tunnel initialisation: `linux/net/ipv4/ip_gre.c`

```

1205
1206     /* Precalculate GRE options length */
1207     if (tunnel->parms.o_flags&(GRE_CSUM|GRE_KEY|GRE_SEQ)) {
1208         if (tunnel->parms.o_flags&GRE_CSUM)
1209             addend += 4;
1210         if (tunnel->parms.o_flags&GRE_KEY)
1211             addend += 4;
1212         if (tunnel->parms.o_flags&GRE_SEQ)
1213             addend += 4;
1214     }
1215     dev->hard_header_len = hlen + addend;
1216     dev->mtu = mtu - addend;
1217     tunnel->hlen = addend;
1218     return 0;
1219 }
1220
1221 #ifdef MODULE
1222 static int ipgre_fb_tunnel_open(struct net_device *dev)
1223 {
1224     MOD_INC_USE_COUNT;
1225     return 0;
1226 }
1227
1228 static int ipgre_fb_tunnel_close(struct net_device *dev)
1229 {
1230     MOD_DEC_USE_COUNT;
1231     return 0;
1232 }
1233 #endif
1234
1235 int _init ipgre_fb_tunnel_init(struct net_device *dev)
1236 {
1237     struct ip_tunnel *tunnel = (struct ip_tunnel*)dev->priv;
1238     struct iphdr *iph;

```

Figure 558: GRE Tunnels - module initialisation: `linux/net/ipv4/ip_gre.c`

```

1242     dev->open          = ipgre_fb_tunnel_open;
1243     dev->stop         = ipgre_fb_tunnel_close;
1244 #endif
1245
1246     iph = &ipgre_fb_tunnel.parms.iph;
1247     iph->version      = 4;
1248     iph->protocol     = IPPROTO_GRE;
1249     iph->ihl          = 5;
1250     tunnel->hlen     = sizeof(struct iphdr) + 4;
1251
1252     dev_hold(dev);
1253     tunnels_wc[0]    = &ipgre_fb_tunnel;
1254     return 0;
1255 }
1256
1257
1258 static struct inet_protocol ipgre_protocol = {
1259     ipgre_rcv ,          /* GRE handler      */
1260     ipgre_err ,        /* TUNNEL error control */
1261     0,                  /* next            */
1262     IPPROTO_GRE,       /* protocol ID     */
1263     0,                  /* copy            */
1264     NULL,              /* data            */
1265     "GRE"              /* name            */
1266 };
1267
1268
1269 /*
1270  *   And now the modules code and kernel interface.
1271  */
1272
1273 #ifdef MODULE
1274 int init_module(void)
1275 #else
1276 int _init ipgre_init (void)
1277 #endif
1278 {
1279     printk(KERN_INFO "GRE_over_IPv4_tunneling_driver\n");
1280
1281     ipgre_fb_tunnel_dev.priv = (void*)&ipgre_fb_tunnel;
1282     register_netdev (&ipgre_fb_tunnel_dev);
1283     inet_add_protocol(&ipgre_protocol);
1284     return 0;
1285 }
1286
1287 #ifdef MODULE

```

Figure 559: GRE Tunnels - Protocol setup: linux/net/ipv4/ip_gre.c

```

76  /*
77  *   IPv6 fixed header
78  *
79  *   BEWARE, it is incorrect. The first 4 bits of flow_lbl
80  *   are glued to priority now, forming "class".
81  */
82
83  struct ipv6hdr {
84  #if defined(_LITTLE_ENDIAN_BITFIELD)
85      __u8          priority :4,
86                  version :4;
87  #elif defined(_BIG_ENDIAN_BITFIELD)
88      __u8          version :4,
89                  priority :4;
90  #else
91  #error "Please fix <asm/byteorder.h>"
92  #endif
93      __u8          flow_lbl [3];
94
95      __u16         payload_len;
96      __u8         nexthdr;
97      __u8         hop_limit;
98
99      struct in6_addr  saddr;
100     struct in6_addr  daddr;
101 };

```

Figure 560: IPv6 packet format: linux/include/linux/ipv6.h

```

23 #define SIN6_LEN_RFC2133    24
24
25 /*
26  *   NextHeader field of IPv6 header
27  */
28
29 #define NEXTHDR_HOP         0      /* Hop-by-hop option header. */
30 #define NEXTHDR_TCP         6      /* TCP segment. */
31 #define NEXTHDR_UDP         17     /* UDP message. */
32 #define NEXTHDR_IPV6        41     /* IPv6 in IPv6 */
33 #define NEXTHDR_ROUTING     43     /* Routing header. */
34 #define NEXTHDR_FRAGMENT    44     /* Fragmentation/reassembly header. */
35 #define NEXTHDR_ESP         50     /* Encapsulating security payload. */
36 #define NEXTHDR_AUTH        51     /* Authentication header. */
37 #define NEXTHDR_ICMP        58     /* ICMP for IPv6. */
38 #define NEXTHDR_NONE        59     /* No next header */
39 #define NEXTHDR_DEST        60     /* Destination options header. */
40
41 #define NEXTHDR_MAX         255

```

Figure 561: Next Header Types: linux/include/net/ipv6.h


```

45 #define IPV6_DEFAULT_HOPLIMIT 64
46 #define IPV6_DEFAULT_MCASTHOPS 1
47
48 /*
49 *   Addr type
50 *
51 *   type   -   unicast | multicast | anycast
52 *   scope  -   local   | site     | global
53 *   v4     -   compat
54 *   v4mapped
55 *   any
56 *   loopback
57 */
58
59 #define IPV6_ADDR_ANY      0x0000U
60
61 #define IPV6_ADDR_UNICAST  0x0001U
62 #define IPV6_ADDR_MULTICAST 0x0002U
63 #define IPV6_ADDR_ANYCAST 0x0004U
64
65 #define IPV6_ADDR_LOOPBACK 0x0010U
66 #define IPV6_ADDR_LINKLOCAL 0x0020U
67 #define IPV6_ADDR_SITELOCAL 0x0040U
68
69 #define IPV6_ADDR_COMPATv4 0x0080U
70
71 #define IPV6_ADDR_SCOPE_MASK 0x00f0U
72
73 #define IPV6_ADDR_MAPPED 0x1000U
74 #define IPV6_ADDR_RESERVED 0x2000U /* reserved address space */

```

Figure 562: IPv6 address family information: `linux/include/net/ipv6.h`

```
80 struct frag_hdr {  
81     unsigned char  nexthdr;  
82     unsigned char  reserved;  
83     unsigned short frag_off ;  
84     _u32           identification ;  
85 };
```

Figure 563: IPv6 fragmentation header: `linux/include/net/ipv6.h`

```

23 struct inet6_ifaddr
24 {
25     struct in6_addr    addr;
26     __u32              prefix_len ;
27
28     __u32              valid_lft ;
29     __u32              prefered_lft ;
30     unsigned long      tstamp;
31     atomic_t           refcnt ;
32     spinlock_t         lock;
33
34     __u8               probes;
35     __u8               flags ;
36
37     __u16              scope;
38
39     struct timer_list  timer;
40
41     struct inet6_dev   *idev;
42
43     struct inet6_ifaddr *lst_next;    /* next addr in addr_lst */
44     struct inet6_ifaddr *if_next;    /* next addr in inet6_dev */
45
46     int                dead;
47 };

```

Figure 564: IPv6 network interface structures:
linux/include/net/ipv6/if_inet6.h

```

49 struct ipv6_mc_socklist
50 {
51     struct in6_addr    addr;
52     int                ifindex ;
53     struct ipv6_mc_socklist *next;
54 };
55
56 #define MAF_TIMER_RUNNING 0x01
57 #define MAF_LAST_REPORTER 0x02
58 #define MAF_LOADED      0x04
59
60 struct ifmcaddr6
61 {
62     struct in6_addr    mca_addr;
63     struct inet6_dev   *idev;
64     struct ifmcaddr6   *next;
65     struct timer_list  mca_timer;
66     unsigned           mca_flags;
67     int                mca_users;
68     atomic_t           mca_refcnt;
69     spinlock_t         mca_lock;
70 };

```

Figure 565: IPv6 multicast interface structures:
[linux/include/net/if_inet6.h](#)

```

72 #define IFA_HOST      IPV6_ADDR_LOOPBACK
73 #define IFA_LINK     IPV6_ADDR_LINKLOCAL
74 #define IFA_SITE     IPV6_ADDR_SITELOCAL
75 #define IFA_GLOBAL   0x0000U
76
77 struct ipv6_devconf
78 {
79     int          forwarding;
80     int          hop_limit;
81     int          mtu6;
82     int          accept_ra;
83     int          accept_redirects ;
84     int          autoconf;
85     int          dad_transmits;
86     int          rtr_solicits ;
87     int          rtr_solicit_interval ;
88     int          rtr_solicit_delay ;
89
90     void         *sysctl;
91 };

```

Figure 566: IPv6 device configuration struct: `linux/include/net/ipv6_devconf.h`

```
93 struct inet6_dev
94 {
95     struct net_device      *dev;
96
97     struct inet6_ifaddr    *addr_list ;
98     struct ifmcaddr6       *mc_list ;
99     rwlock_t               lock;
100    atomic_t                refcnt ;
101    _u32                    if_flags ;
102    int                     dead;
103
104    struct neigh_parms      *nd_parms;
105    struct inet6_dev        *next;
106    struct ipv6_devconf     cnf;
107 };
```

Figure 567: IPv6 device data: `linux/include/net/if_inet6.h`

```

111 static inline void ipv6_eth_mc_map(struct in6_addr *addr, char *buf)
112 {
113     /*
114     *   +-----+-----+-----+-----+-----+
115     *   |  33  |  33  | DST13 | DST14 | DST15 | DST16 |
116     *   +-----+-----+-----+-----+-----+
117     */
118
119     buf[0]= 0x33;
120     buf[1]= 0x33;
121
122     memcpy(buf + 2, &addr->s6_addr32[3], sizeof(_u32));
123 }

```

Figure 568: IPv6 example of ethernet to ipv6 map:
[linux/include/net/if_inet6.h](#)

```

26  /*
27  *   IPv6 address structure
28  */
29
30  struct in6_addr
31  {
32      union
33      {
34          __u8          u6_addr8[16];
35          __u16         u6_addr16[8];
36          __u32         u6_addr32[4];
37      } in6_u;
38  #define s6_addr          in6_u.u6_addr8
39  #define s6_addr16       in6_u.u6_addr16
40  #define s6_addr32       in6_u.u6_addr32
41  };
42
43  struct sockaddr_in6 {
44      unsigned short int  sin6_family; /* AF_INET6 */
45      __u16               sin6_port; /* Transport layer port # */
46      __u32               sin6_flowinfo; /* IPv6 flow information */
47      struct in6_addr      sin6_addr; /* IPv6 address */
48      __u32               sin6_scope_id; /* scope id (new in RFC2553) */
49  };

```

Figure 569: IPv6 API - address family: `linux/include/linux/in6.h`


```
51 struct ipv6_mreq {
52     /* IPv6 multicast address of group */
53     struct in6_addr ipv6mr_multiaddr;
54
55     /* local IPv6 address of interface */
56     int             ipv6mr_ifindex;
57 };
```

Figure 570: IPv6 API - multicast programmer interface:
linux/include/linux/in6.h

```

59 struct in6_flowlabel_req
60 {
61     struct in6_addr flr_dst ;
62     __u32   flr_label ;
63     __u8    flr_action ;
64     __u8    flr_share ;
65     __u16   flr_flags ;
66     __u16   flr_expires ;
67     __u16   flr_linger ;
68     __u32   __flr_pad ;
69     /* Options in format of IPV6_PKTOPTIONS */
70 };
71
72 #define IPV6_FL_A_GET 0
73 #define IPV6_FL_A_PUT 1
74 #define IPV6_FL_A_RENEW 2
75
76 #define IPV6_FL_F_CREATE 1
77 #define IPV6_FL_F_EXCL 2
78
79 #define IPV6_FL_S_NONE 0
80 #define IPV6_FL_S_EXCL 1
81 #define IPV6_FL_S_PROCESS 2
82 #define IPV6_FL_S_USER 3
83 #define IPV6_FL_S_ANY 255

```

Figure 571: IPv6 API - flow label programmer interface structure:
linux/include/linux/in6.h

```

126 int ipv6_addr_type(struct in6_addr *addr)
127 {
128     u32 st;
129
130     st = addr->s6_addr32[0];
131
132     /* Consider all addresses with the first three bits different of
133        000 and 111 as unicasts.
134     */
135     if ((st & __constant_htonl(0xE0000000)) != __constant_htonl(0x00000000) &&
136         (st & __constant_htonl(0xE0000000)) != __constant_htonl(0xE0000000))
137         return IPV6_ADDR_UNICAST;
138
139     if ((st & __constant_htonl(0xFF000000)) == __constant_htonl(0xFF000000)) {
140         int type = IPV6_ADDR_MULTICAST;
141
142         switch((st & __constant_htonl(0x00FF0000))) {
143             case __constant_htonl(0x00010000):
144                 type |= IPV6_ADDR_LOOPBACK;
145                 break;
146
147             case __constant_htonl(0x00020000):
148                 type |= IPV6_ADDR_LINKLOCAL;
149                 break;
150
151             case __constant_htonl(0x00050000):
152                 type |= IPV6_ADDR_SITELOCAL;
153                 break;
154         };
155         return type;
156     }

```

Figure 572: IP Address Configurator - Determine Address Type:
linux/net/ipv6/addrconf.c

```

158     if ((st & _constant_htonl(0xFFC00000)) == _constant_htonl(0xFE800000))
159         return (IPV6_ADDR_LINKLOCAL | IPV6_ADDR_UNICAST);
160
161     if ((st & _constant_htonl(0xFFC00000)) == _constant_htonl(0xFEC00000))
162         return (IPV6_ADDR_SITELOCAL | IPV6_ADDR_UNICAST);
163
164     if ((addr->s6_addr32[0] | addr->s6_addr32[1]) == 0) {
165         if (addr->s6_addr32[2] == 0) {
166             if (addr->in6_u.u6_addr32[3] == 0)
167                 return IPV6_ADDR_ANY;
168
169             if (addr->s6_addr32[3] == _constant_htonl(0x00000001))
170                 return (IPV6_ADDR_LOOPBACK |
171                     IPV6_ADDR_UNICAST);
172
173             return (IPV6_ADDR_COMPATv4 | IPV6_ADDR_UNICAST);
174         }
175         if (addr->s6_addr32[2] == _constant_htonl(0x0000ffff))
176             return IPV6_ADDR_MAPPED;
177     }
178
179     return IPV6_ADDR_RESERVED;
180 }

```

Figure 573: IP Address Configurator - Determine Address Type:
[linux/net/ipv6/addrconf.c](#)

```

434     in6_ifa_put (ifp);
435 }
436
437
438 /*
439  * Choose an appropriate source address
440  * should do:
441  * i)    get an address with an appropriate scope
442  * ii)   see if there is a specific route for the destination and use
443  *       an address of the attached interface
444  * iii)  don't use deprecated addresses
445  */
446 int ipv6_get_saddr(struct dst_entry *dst,
447                  struct in6_addr *daddr, struct in6_addr *saddr)
448 {
449     int scope;
450     struct inet6_ifaddr *ifp = NULL;
451     struct inet6_ifaddr *match = NULL;
452     struct net_device *dev = NULL;
453     struct inet6_dev *idev;
454     struct rt6_info *rt;
455     int err;
456
457     rt = (struct rt6_info *) dst;
458     if (rt)
459         dev = rt->rt6i_dev;
460
461     scope = ipv6_addr_scope(daddr);
462     if (rt && (rt->rt6i_flags & RTF_ALLONLINK)) {
463         /*
464          * route for the "all destinations on link" rule

```

Figure 574: Select IPv6 Source Address given Destination to get to:
linux/net/ipv6/addrconf.c

```

468     }
469
470     /*
471     *    known dev
472     *    search dev and walk through dev addresses
473     */
474
475     if (dev) {
476         if (dev->flags & IFF_LOOPBACK)
477             scope = IFA_HOST;
478
479         read_lock(&addrconf_lock);
480         idev = __in6_dev_get(dev);
481         if (idev) {
482             read_lock_bh(&idev->lock);
483             for (ifp=idev->addr_list; ifp; ifp=ifp->if_next) {
484                 if (ifp->scope == scope) {
485                     if (!(ifp->flags & (IFA_F_DEPRECATED|
486                                     IFA_F_TENTATIVE))) {
487                         in6_ifa_hold(ifp);
488                         read_unlock_bh(&idev->lock);
489                         read_unlock(&addrconf_lock);
490                         goto out;
491                     }
492
493                     if (!match && !(ifp->flags &
494                                     IFA_F_TENTATIVE)) {
495                         match = ifp;
496                         in6_ifa_hold(ifp);
497                     }
498                 }
499             }
500         }
501     }

```

Figure 575: Select IPv6 Source Address given Destination to get to:
linux/net/ipv6/addrconf.c

```

499     }
500     read_unlock(&addrconf_lock);
501 }
502
503 if (scope == IFA_LINK)
504     goto out;
505
506 /*
507  * dev == NULL or search failed for specified dev
508  */
509
510 read_lock(&dev_base_lock);
511 read_lock(&addrconf_lock);
512 for (dev = dev_base; dev; dev=dev->next) {
513     idev = _in6_dev_get(dev);
514     if (idev) {
515         read_lock_bh(&idev->lock);
516         for (ifp=idev->addr_list; ifp; ifp=ifp->if_next) {
517             if (ifp->scope == scope) {
518                 if (!(ifp->flags&(IFA_F_DEPRECATED|
519                     IFA_F_TENTATIVE))) {
520                     in6_ifa_hold(ifp);
521                     read_unlock_bh(&idev->lock);
522                     goto out_unlock_base;
523                 }
524                 if (!match && !(ifp->flags&IFA_F_TENTATIVE
525                     )) {
526                     match = ifp;
527                     in6_ifa_hold(ifp);
528                 }

```

Figure 576: Select IPv6 Source Address given Destination to get to:
linux/net/ipv6/addrconf.c

```

529         }
530         read_unlock_bh(&idev->lock);
531     }
532 }
533
534 out_unlock_base:
535     read_unlock(&addrconf_lock);
536     read_unlock(&dev_base_lock);
537
538 out:
539     if (ifp == NULL) {
540         ifp = match;
541         match = NULL;
542     }
543
544     err = -EADDRNOTAVAIL;
545     if (ifp) {
546         ipv6_addr_copy(saddr, &ifp->addr);
547         err = 0;
548         in6_ifa_put(ifp);
549     }
550     if (match)

```

Figure 577: Select IPv6 Source Address given Destination to get to:
[linux/net/ipv6/addrconf.c](#)


```

667 }
668
669
670 #ifdef CONFIG_IPV6_EUI64
671 static int ipv6_generate_eui64(u8 *eui, struct net_device *dev)
672 {
673     switch (dev->type) {
674     case ARPHRD_ETHER:
675     case ARPHRD_FDDI:
676     case ARPHRD_IEEE802_TR:
677         if (dev->addr_len != ETH_ALEN)
678             return -1;
679         memcpy(eui, dev->dev_addr, 3);
680         memcpy(eui + 5, dev->dev_addr+3, 3);
681         eui[3] = 0xFF;
682         eui[4] = 0xFE;
683         eui[0] ^= 2;

```

Figure 578: Deriving EUI64 bits of IPv6 Address:
linux/net/ipv6/addrconf.c

```

707  /*
708  *    Add prefix route.
709  */
710
711  static void
712  addrconf_prefix_route(struct in6_addr *pfx, int plen, struct net_device *dev,
713                      unsigned long expires, unsigned flags)
714  {
715      struct in6_rtmmsg rtmsg;
716
717      memset(&rtmsg, 0, sizeof(rtmsg));
718      memcpy(&rtmsg.rtmmsg_dst, pfx, sizeof(struct in6_addr));
719      rtmsg.rtmmsg_dst_len = plen;
720      rtmsg.rtmmsg_metric = IP6_RT_PRIO_ADDRCONF;
721      rtmsg.rtmmsg_ifindex = dev->ifindex;
722      rtmsg.rtmmsg_info = expires;
723      rtmsg.rtmmsg_flags = RTF_UP|flags;
724      rtmsg.rtmmsg_type = RTMSG_NEWROUTE;
725
726      /* Prevent useless cloning on PtP SIT.
727       * This thing is done here expecting that the whole
728       * class of non-broadcast devices need not cloning.
729       */
730      if (dev->type == ARPHRD_SIT && (dev->flags&IFF_POINTOPOINT))

```

Figure 579: Adding prefix route: `linux/net/ipv6/addrconf.c`

```

791     addrconf_add_lroute(dev);
792     return idev;
793 }
794
795 void addrconf_prefix_rcv(struct net_device *dev, u8 *opt, int len)
796 {
797     struct prefix_info *pinfo;
798     struct rt6_info *rt;
799     _u32 valid_lft;
800     _u32 preferred_lft;
801     int addr_type;
802     unsigned long rt_expires;
803     struct inet6_dev *in6_dev;
804
805     pinfo = (struct prefix_info *) opt;
806
807     if (len < sizeof(struct prefix_info)) {
808         ADBG(("addrconf:_prefix_option_too_short\n"));
809         return;
810     }
811
812     /*
813      *   Validation checks ([ADDRCONF], page 19)
814      */

```

Figure 580: Adding prefix route: linux/net/ipv6/addrconf.c

```
817     if ( addr_type & (IPV6_ADDR_MULTICAST|IPV6_ADDR_LINKLOCAL))
818         return;
819
820     valid_lft = ntohl(pinfo->valid);
821     preferred_lft = ntohl(pinfo->preferred);
822
823     if ( preferred_lft > valid_lft ) {
824         printk(KERN_WARNING "addrconf: prefix option has invalid lifetime\n"
825                );
826         return;
827     }
828
829     in6_dev = in6_dev_get(dev);
```

Figure 581: Adding prefix route: `linux/net/ipv6/addrconf.c`

```

832         printk(KERN_DEBUG "addrconf: device %s not configured\n", dev->
833                name);
834         return;
835     }
836     /*
837     *   Two things going on here:
838     *   1) Add routes for on-link prefixes
839     *   2) Configure prefixes with the auto flag set
840     */
841
842     /* Avoid arithmetic overflow. Really, we could
843        save rt_expires in seconds, likely valid_lft,
844        but it would require division in fib gc, that it
845        not good.
846     */
847     if ( valid_lft >= 0x7FFFFFFF/HZ)
848         rt_expires = 0;
849     else
850         rt_expires = jiffies + valid_lft * HZ;
851
852     rt = rt6_lookup(&pinfo->prefix, NULL, dev->ifindex, 1);
853
854     if (rt && ((rt->rt6i_flags & (RTF_GATEWAY | RTF_DEFAULT)) == 0)) {
855         if (rt->rt6i_flags & RTF_EXPIRES) {
856             if (pinfo->onlink == 0 || valid_lft == 0) {
857                 ip6_del_rt (rt);
858                 rt = NULL;
859             } else {
860                 rt->rt6i_expires = rt_expires;
861             }
862         }
863     } else if (pinfo->onlink && valid_lft) {
864         addrconf_prefix_route(&pinfo->prefix, pinfo->prefix_len,

```

Figure 582: Adding prefix route: linux/net/ipv6/addrconf.c

```

866     }
867     if (rt)
868         dst_release(&rt->u.dst);
869
870     /* Try to figure out our local address for this prefix */
871
872     if (pinfo->autoconf && in6_dev->cnf.autoconf) {
873         struct inet6_ifaddr * ifp;
874         struct in6_addr addr;
875         int plen;
876
877         plen = pinfo->prefix_len >> 3;
878
879 #ifdef CONFIG_IPV6_EUI64
880         if (pinfo->prefix_len == 64) {
881             memcpy(&addr, &pinfo->prefix, 8);
882             if (ipv6_generate_eui64(addr.s6_addr + 8, dev) &&
883                 ipv6_inherit_eui64(addr.s6_addr + 8, in6_dev)) {
884                 in6_dev_put(in6_dev);
885                 return;
886             }
887             goto ok;
888         }
889 #endif
890 #ifndef CONFIG_IPV6_NO_PB
891         if (pinfo->prefix_len == ((sizeof(struct in6_addr) - dev->addr_len)
892             << 3)) {
893             memcpy(&addr, &pinfo->prefix, plen);
894             memcpy(addr.s6_addr + plen, dev->dev_addr,
895                 dev->addr_len);
896             goto ok;
897         }

```

Figure 583: Adding prefix route: `linux/net/ipv6/addrconf.c`

```

898         printk(KERN_DEBUG "IPv6 addrconf: prefix with wrong length\n",
899                pinfo->prefix_len);
900         in6_dev_put(in6_dev);
901         return;
902     ok:
903
904         ifp = ipv6_get_ifaddr(&addr, dev);
905
906         if ( ifp == NULL && valid_lft) {
907             ifp = ipv6_add_addr(in6_dev, &addr, pinfo->prefix_len,
908                               addr_type&IPV6_ADDR_SCOPE_MASK, 0);
909
910             if ( ifp == NULL) {
911                 in6_dev_put(in6_dev);
912                 return;
913             }
914
915             addrconf_dad_start(ifp);
916         }

```

Figure 584: Adding prefix route: `linux/net/ipv6/addrconf.c`

```

919         ipv6_del_addr(ifp);
920         ifp = NULL;
921     }
922
923     if (ifp) {
924         int flags;
925
926         spin_lock(&ifp->lock);
927         ifp->valid_lft = valid_lft;
928         ifp->preferred_lft = preferred_lft;
929         ifp->tstamp = jiffies;
930         flags = ifp->flags;
931         ifp->flags &= ~IFA_F_DEPRECATED;
932         spin_unlock(&ifp->lock);
933
934         if (!(flags & IFA_F_TENTATIVE))
935             ipv6_ifa_notify ((flags & IFA_F_DEPRECATED) ?
936                             0 : RTM_NEWADDR, ifp);
937         in6_ifa_put(ifp);

```

Figure 585: Adding prefix route: `linux/net/ipv6/addrconf.c`


```

1886         return ret;
1887     }
1888 }
1889
1890 static struct addrconf_sysctl_table
1891 {
1892     struct ctl_table_header *sysctl_header;
1893     ctl_table addrconf_vars[11];
1894     ctl_table addrconf_dev[2];
1895     ctl_table addrconf_conf_dir[2];
1896     ctl_table addrconf_proto_dir[2];
1897     ctl_table addrconf_root_dir[2];
1898 } addrconf_sysctl = {
1899     NULL,
1900     {{NET_IPV6_FORWARDING, "forwarding",
1901      &ipv6_devconf.forwarding, sizeof(int), 0644, NULL,
1902      &addrconf_sysctl_forward},
1903
1904     {NET_IPV6_HOP_LIMIT, "hop_limit",
1905      &ipv6_devconf.hop_limit, sizeof(int), 0644, NULL,
1906      &proc_dointvec},
1907
1908     {NET_IPV6_MTU, "mtu",
1909      &ipv6_devconf.mtu6, sizeof(int), 0644, NULL,
1910      &proc_dointvec},

```

Figure 586: IPv6 addrconf sysctl: `linux/net/ipv6/addrconf.c`

```

1912     {NET_IPV6_ACCEPT_RA, "accept_ra",
1913       &ipv6_devconf.accept_ra, sizeof(int), 0644, NULL,
1914       &proc_dointvec},
1915
1916     {NET_IPV6_ACCEPT_REDIRECTS, "accept_redirects",
1917       &ipv6_devconf.accept_redirects, sizeof(int), 0644, NULL,
1918       &proc_dointvec},
1919
1920     {NET_IPV6_AUTOCONF, "autoconf",
1921       &ipv6_devconf.autoconf, sizeof(int), 0644, NULL,
1922       &proc_dointvec},
1923
1924     {NET_IPV6_DAD_TRANSMITS, "dad_transmits",
1925       &ipv6_devconf.dad_transmits, sizeof(int), 0644, NULL,
1926       &proc_dointvec},
1927
1928     {NET_IPV6_RTR_SOLICITS, "router_solicitations",
1929       &ipv6_devconf.rtr_solicits, sizeof(int), 0644, NULL,
1930       &proc_dointvec},
1931
1932     {NET_IPV6_RTR_SOLICIT_INTERVAL, "router_solicitation_interval",
1933       &ipv6_devconf.rtr_solicit_interval, sizeof(int), 0644, NULL,
1934       &proc_dointvec_jiffies},
1935
1936     {NET_IPV6_RTR_SOLICIT_DELAY, "router_solicitation_delay",
1937       &ipv6_devconf.rtr_solicit_delay, sizeof(int), 0644, NULL,
1938       &proc_dointvec_jiffies},
1939
1940     {0}},
1941
1942     {{NET_PROTO_CONF_ALL, "all", NULL, 0, 0555, addrconf_sysctl.addrconf_vars
      }, {0}},

```

Figure 587: IPv6 addrconf sysctl: `linux/net/ipv6/addrconf.c`

```

86 static struct neigh_ops ndisc_generic_ops =
87 {
88     AF_INET6,
89     NULL,
90     ndisc_solicit ,
91     ndisc_error_report ,
92     neigh_resolve_output ,
93     neigh_connected_output ,
94     dev_queue_xmit,
95     dev_queue_xmit
96 };
97
98 static struct neigh_ops ndisc_hh_ops =
99 {
100     AF_INET6,
101     NULL,
102     ndisc_solicit ,
103     ndisc_error_report ,
104     neigh_resolve_output ,
105     neigh_resolve_output ,
106     dev_queue_xmit,
107     dev_queue_xmit
108 };
109
110
111 static struct neigh_ops ndisc_direct_ops =
112 {
113     AF_INET6,
114     NULL,
115     NULL,
116     NULL,
117     dev_queue_xmit,
118     dev_queue_xmit,
119     dev_queue_xmit,
120     dev_queue_xmit
121 };

```

Figure 588: IPv6 Neighbour Discovery Operations: `linux/net/ipv6/ndisc.c`

```

317  /*
318  *   Send a Neighbour Advertisement
319  */
320
321  void ndisc_send_na(struct net_device *dev, struct neighbour *neigh,
322                   struct in6_addr *daddr, struct in6_addr *solicited_addr,
323                   int router, int solicited, int override, int inc_opt)
324  {
325     struct sock *sk = ndisc_socket->sk;
326     struct nd_msg *msg;
327     int len;
328     struct sk_buff *skb;
329     int err;
330
331     len = sizeof(struct icmp6hdr) + sizeof(struct in6_addr);
332
333     if (inc_opt) {
334         if (dev->addr_len)
335             len += NDISC_OPT_SPACE(dev->addr_len);
336         else
337             inc_opt = 0;
338     }
339
340     skb = sock_alloc_send_skb(sk, MAX_HEADER + len + dev->hard_header_len
341                              + 15,
342                              0, &err);
343
344     if (skb == NULL) {
345         ND_PRINTK1("send_na: alloc_skb_failed\n");
346         return;
347     }
348
349     if (ndisc_build_ll_hdr(skb, dev, daddr, neigh, len) == 0) {
350         kfree_skb(skb);
351         return;
352     }
353
354     ip6_nd_hdr(sk, skb, dev, solicited_addr, daddr, IPPROTO_ICMPV6, len);
355
356     msg = (struct nd_msg *) skb_put(skb, len);
357
358     msg->icmph.icmp6_type = NDISC_NEIGHBOUR_ADVERTISEMENT;
359     msg->icmph.icmp6_code = 0;
360     msg->icmph.icmp6_cksum = 0;

```

Figure 589: IPv6 Neighbour Advertisement: `linux/net/ipv6/ndisc.c`

```

361     msg->icmph.icmp6_unused = 0;
362     msg->icmph.icmp6_router = router;
363     msg->icmph.icmp6_solicited = solicited;
364     msg->icmph.icmp6_override = !override;
365
366     /* Set the target address. */
367     ipv6_addr_copy(&msg->target, solicited_addr);
368
369     if (inc_opt)
370         ndisc_fill_option ((void*)&msg->opt, ND_OPT_TARGET_LL_ADDR, dev
371                             ->dev_addr, dev->addr_len);
372
373     /* checksum */
374     msg->icmph.icmp6_cksum = csum_ipv6_magic(solicited_addr, daddr, len,
375                                             IPPROTO_ICMPV6,
376                                             csum_partial((__u8 *) msg,
377                                                         len, 0));
378
379     dev_queue_xmit(skb);
380
381     ICMP6_INC_STATS(Icmp6OutNeighborAdvertisements);
382     ICMP6_INC_STATS(Icmp6OutMsgs);
383 }

```

Figure 590: IPv6 Neighbour Advertisement: `linux/net/ipv6/ndisc.c`

```

569 }
570
571 static void ndisc_router_discovery (struct sk_buff *skb)
572 {
573     struct ra_msg *ra_msg = (struct ra_msg *) skb->h.raw;
574     struct neighbour *neigh;
575     struct inet6_dev *in6_dev;
576     struct rt6_info *rt;
577     int lifetime;
578     int optlen;
579
580     __u8 * opt = (__u8 *) (ra_msg + 1);
581
582     optlen = (skb->tail - skb->h.raw) - sizeof(struct ra_msg);
583
584     if (skb->nh.ipv6h->hop_limit != 255) {
585         printk(KERN_INFO
586             "NDISC: fake_router_advertisement_received\n");
587         return;
588     }
589
590     /*
591      *   set the RA_RECV flag in the interface
592      */
593
594     in6_dev = in6_dev_get(skb->dev);
595     if (in6_dev == NULL) {
596         ND_PRINTK1("RA: can't find in6 device\n");
597         return;
598     }
599     if (in6_dev->cnf.forwarding || !in6_dev->cnf.accept_ra) {
600         in6_dev_put(in6_dev);

```

Figure 591: IPv6 Router Discovery: `linux/net/ipv6/ndisc.c`

```

602     }
603
604     if (in6_dev->if_flags & IF_RS_SENT) {
605         /*
606          *   flag that an RA was received after an RS was sent
607          *   out on this interface.
608          */
609         in6_dev->if_flags |= IF_RA_RCVD;
610     }
611
612     lifetime = ntohs(ra_msg->icmph.icmp6_rt_lifetime);
613
614     rt = rt6_get_dflt_router (&skb->nh.ipv6h->saddr, skb->dev);
615
616     if (rt && lifetime == 0) {
617         ip6_del_rt (rt);
618         rt = NULL;
619     }
620
621     if (rt == NULL && lifetime) {
622         ND_PRINTK2("ndisc_rdisc: adding default router\n");
623
624         rt = rt6_add_dflt_router (&skb->nh.ipv6h->saddr, skb->dev);
625         if (rt == NULL) {
626             ND_PRINTK1("route_add_failed\n");
627             in6_dev_put(in6_dev);
628             return;
629         }
630
631         neigh = rt->rt6i_nexthop;
632         if (neigh == NULL) {
633             ND_PRINTK1("nd: add default router: null neighbour\n");
634             dst_release (&rt->u.dst);
635             in6_dev_put(in6_dev);
636             return;
637         }
638         neigh->flags |= NTF_ROUTER;
639
640         /*
641          *   If we were using an "all destinations on link" route
642          *   delete it
643          */

```

Figure 592: IPv6 Router Discovery: `linux/net/ipv6/ndisc.c`

```

646     }
647
648     if (rt)
649         rt->rt6i_expires = jiffies + (HZ * lifetime);
650
651     if (ra_msg->icmph.icmp6_hop_limit)
652         in6_dev->cnf.hop_limit = ra_msg->icmph.icmp6_hop_limit;
653
654     /*
655      *   Update Reachable Time and Retrans Timer
656      */
657
658     if (in6_dev->nd_parms) {
659         __u32 rtime = ntohl(ra_msg->retrans_timer);
660
661         if (rtime && rtime/1000 < MAX_SCHEDULE_TIMEOUT/HZ) {
662             rtime = (rtime*HZ)/1000;
663             if (rtime < HZ/10)
664                 rtime = HZ/10;
665             in6_dev->nd_parms->retrans_time = rtime;
666         }
667
668         rtime = ntohl(ra_msg->reachable_time);
669         if (rtime && rtime/1000 < MAX_SCHEDULE_TIMEOUT/(3*HZ)) {
670             rtime = (rtime*HZ)/1000;
671
672             if (rtime < HZ/10)
673                 rtime = HZ/10;
674
675             if (rtime != in6_dev->nd_parms->base_reachable_time) {
676                 in6_dev->nd_parms->base_reachable_time = rtime;
677                 in6_dev->nd_parms->gc_staletime = 3 * rtime;
678                 in6_dev->nd_parms->reachable_time =
679                     neigh_rand_reach_time(rtime);
680             }
681         }
682     }

```

Figure 593: IPv6 Router Discovery: `linux/net/ipv6/ndisc.c`


```

681     }
682
683     /*
684     *     Process options.
685     */
686
687     while (optlen > 0) {
688         int len = (opt[1] << 3);
689
690         if (len == 0) {
691             ND_PRINTK0("RA: _opt_has_0_len\n");
692             break;
693         }
694
695         switch(*opt) {
696             case ND_OPT_SOURCE_LL_ADDR:
697
698                 if (rt == NULL)
699                     break;
700
701                 if ((neigh = rt->rt6i_nexthop) != NULL &&
702                     skb->dev->addr_len + 2 >= len)
703                     neigh_update(neigh, opt+2, NUD_STALE, 1, 1);
704                 break;
705
706             case ND_OPT_PREFIX_INFO:

```

Figure 594: IPv6 Router Discovery: `linux/net/ipv6/ndisc.c`

```

708         break;
709
710     case ND_OPT_MTU:
711         {
712             int mtu;
713
714             mtu = htonl*((u32*)(opt+4));
715
716             if (mtu < IPV6_MIN_MTU || mtu > skb->dev->mtu) {
717                 ND_PRINTK0("NDISC:router_
718                     "announcement_with_mtu_=%d\n",
719                     mtu);
720                 break;
721             }
722
723             if (in6_dev->cnf.mtu6 != mtu) {
724                 in6_dev->cnf.mtu6 = mtu;
725
726                 if (rt)
727                     rt->u.dst.pmtu = mtu;
728
729                 rt6_mtu_change(skb->dev, mtu);
730             }
731         }
732         break;
733
734     case ND_OPT_TARGET_LL_ADDR:
735     case ND_OPT_REDIRECT_HDR:
736         ND_PRINTK0("got illegal_option_with_RA");
737         break;
738     default :
739         ND_PRINTK0("unkown_option_in_RA\n");
740     };
741     optlen -= len;
742     opt += len;
743 }
744 if (rt)
745     dst_release(&rt->u.dst);

```

Figure 595: IPv6 Router Discovery: `linux/net/ipv6/ndisc.c`

```

54 {
55     struct ipv6hdr *hdr;
56     u32         pkt_len;
57
58     if (skb->pkt_type == PACKET_OTHERHOST)
59         goto drop;
60
61     IP6_INC_STATS_BH(Ip6InReceives);
62
63     if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL)
64         goto out;
65
66     /* Store incoming device index. When the packet will
67        be queued, we cannot refer to skb->dev anymore.
68        */
69     ((struct inet6_skb_parm *)skb->cb)->iif = dev->ifindex;
70
71     if (skb->len < sizeof(struct ipv6hdr))
72         goto err;
73
74     if (!pskb_may_pull(skb, sizeof(struct ipv6hdr)))
75         goto drop;
76
77     hdr = skb->nh.ipv6h;

```

Figure 596: IPv6 Receive: `linux/net/ipv6/ip6_input.c`

```

79     if (hdr->version != 6)
80         goto err;
81
82     pkt_len = ntohs(hdr->payload_len);
83
84     /* pkt_len may be zero if Jumbo payload option is present */
85     if (pkt_len || hdr->nexthdr != NEXTHDR_HOP) {
86         if (pkt_len + sizeof(struct ipv6hdr) > skb->len)
87             goto truncated;
88         if (pkt_len + sizeof(struct ipv6hdr) < skb->len) {
89             if (_pskb_trim(skb, pkt_len + sizeof(struct ipv6hdr)))
90                 goto drop;
91             hdr = skb->nh.ipv6h;
92             if (skb->ip_summed == CHECKSUM_HW)
93                 skb->ip_summed = CHECKSUM_NONE;
94         }
95     }
96
97     if (hdr->nexthdr == NEXTHDR_HOP) {
98         skb->h.raw = (u8*)(hdr+1);
99         if (ipv6_parse_hopopts(skb, offsetof(struct ipv6hdr, nexthdr)) < 0) {
100             IP6_INC_STATS_BH(Ip6InHdrErrors);
101             return 0;
102         }

```

Figure 597: IPv6 Receive: linux/net/ipv6/ip6_input.c

```
45 static inline int ip6_rcv_finish ( struct sk_buff *skb)
46 {
47     if (skb->dst == NULL)
48         ip6_route_input (skb);
49
50     return skb->dst->input(skb);
51 }
```

Figure 598: IPv6 Receive Finish: `linux/net/ipv6/ip6_input.c`

```

98
99 int ip6_output(struct sk_buff *skb)
100 {
101     struct dst_entry *dst = skb->dst;
102     struct net_device *dev = dst->dev;
103
104     skb->protocol = _constant_htons(ETH_P_IPV6);
105     skb->dev = dev;
106
107     if ( ip6_addr_is_multicast(&skb->nh.ipv6h->daddr) ) {
108         if (!( dev->flags&IFF_LOOPBACK) &&
109             (skb->sk == NULL || skb->sk->net_pinfo.af_inet6.mc_loop) &&
110             ipv6_chk_mcast_addr(dev, &skb->nh.ipv6h->daddr) ) {
111             struct sk_buff *newskb = skb_clone(skb, GFP_ATOMIC);
112
113             /* Do not check for IFF_ALLMULTI; multicast routing
114              is not supported in any case.
115             */
116             if (newskb)
117                 NF_HOOK(PF_INET6, NF_IP6_POST_ROUTING, newskb
118                     , NULL,
119                     newskb->dev,
120                     ip6_dev_loopback_xmit);
121
122             if (skb->nh.ipv6h->hop_limit == 0) {
123                 kfree_skb(skb);
124                 return 0;
125             }
126
127             IP6_INC_STATS(Ip6OutMcastPkts);
128         }
129     }
130     return NF_HOOK(PF_INET6, NF_IP6_POST_ROUTING, skb, NULL, skb->dev,
131                 ip6_output_finish);

```

Figure 599: IPv6 - Output: linux/net/ipv6/ip6_output.c

```

180
181 int ip6_xmit(struct sock *sk, struct sk_buff *skb, struct flowi *fl,
182             struct ipv6_txoptions *opt)
183 {
184     struct ipv6_pinfo *np = sk ? &sk->net_pinfo.af_inet6 : NULL;
185     struct in6_addr *first_hop = fl->nl_u.ip6_u.daddr;
186     struct dst_entry *dst = skb->dst;
187     struct ipv6hdr *hdr;
188     u8 proto = fl->proto;
189     int seg_len = skb->len;
190     int hlimit;
191
192     if (opt) {
193         int head_room;
194
195         /* First: exthdrs may take lots of space (~8K for now)
196            MAX_HEADER is not enough.
197            */
198         head_room = opt->opt_nflen + opt->opt_flen;
199         seg_len += head_room;
200         head_room += sizeof(struct ipv6hdr) + ((dst->dev->hard_header_len
201            + 15)&~15);
202
203         if (skb_headroom(skb) < head_room) {
204             struct sk_buff *skb2 = skb_realloc_headroom(skb, head_room);
205             kfree_skb(skb);
206             skb = skb2;
207             if (skb == NULL)
208                 return -ENOBUFS;
209             if (sk)
210                 skb_set_owner_w(skb, sk);
211         }
212         if (opt->opt_flen)
213             ipv6_push_frag_opts(skb, opt, &proto);
214         if (opt->opt_nflen)
215             ipv6_push_nfrag_opts(skb, opt, &proto, &first_hop);
216     }
217 }

```

Figure 600: IPv6 - Xmit: linux/net/ipv6/ip6_output.c

```

218
219     /*
220     *     Fill in the IPv6 header
221     */
222
223     *(u32*)hdr = _constant_htonl(0x60000000) | fl->fl6_flowlabel;
224     hlimit = -1;
225     if (np)
226         hlimit = np->hop_limit;
227     if (hlimit < 0)
228         hlimit = ((struct rt6_info *)dst)->rt6i_hoplimit;
229
230     hdr->payload_len = htons(seg_len);
231     hdr->nexthdr = proto;
232     hdr->hop_limit = hlimit;
233
234     ipv6_addr_copy(&hdr->saddr, fl->nl_u.ipv6_u.saddr);
235     ipv6_addr_copy(&hdr->daddr, first_hop);
236
237     if (skb->len <= dst->pmtu) {
238         IP6_INC_STATS(Ip6OutRequests);
239         return NF_HOOK(PF_INET6, NF_IP6_LOCAL_OUT, skb, NULL, dst->
240             dev, ip6_maybe_reroute);
241     }
242
243     printk(KERN_DEBUG "IPv6: _sending_pkt_too_big_to_self\n");
244     icmpv6_send(skb, ICMPV6_PKT_TOOBIG, 0, dst->pmtu, skb->dev);
245     kfree_skb(skb);
246     return -EMSGSIZE;

```

Figure 601: IPv6 - Output: linux/net/ipv6/ip6_output.c



Figure 602: IPv6 - Output: `linux/net/ipv6/ip6_output.c`

```

413     last_skb->dst = dst_clone(dst);
414
415     skb_reserve(last_skb, (dst->dev->hard_header_len + 15) & ~15);
416
417     hdr = ip6_bld_l(sk, last_skb, fl, hlimit, frag_len + unfrag_len);
418     prev_hdr = &hdr->nexthdr;
419
420
421     if (opt && opt->opt_nflen)
422         prev_hdr = ipv6_build_nfrag_opts(last_skb, prev_hdr, opt, final_dst, 0);
423
424     prev_hdr = ipv6_build_fraghdr(last_skb, prev_hdr, frag_off);
425     fhdr_dist = prev_hdr - last_skb->data;
426
427     err = getfrag(data, &hdr->saddr, last_skb->tail, data_off, last_len);
428
429     if (!err) {
430         while (nfrags--) {
431             struct sk_buff *skb;
432
433             struct frag_hdr *fhdr2;
434
435             skb = skb_copy(last_skb, sk->allocation);
436
437             if (skb == NULL) {
438                 IP6_INC_STATS(Ip6FragFails);
439                 kfree_skb(last_skb);
440                 return -ENOMEM;
441             }
442
443             frag_off -= frag_len;
444             data_off -= frag_len;
445
446             fhdr2 = (struct frag_hdr *) (skb->data + fhdr_dist);
447
448             /* more flag on */
449             fhdr2->frag_off = htons(frag_off | 1);
450
451             /* Write fragmentable exthdrs to the first chunk */
452             if (nfrags == 0 && opt && opt->opt_nflen) {
453                 ipv6_build_frag_opts(skb, &fhdr2->nexthdr, opt);
454                 frag_len -= opt->opt_nflen;
455                 data_off = 0;
456             }
457
458             err = getfrag(data, &hdr->saddr, skb_put(skb, frag_len),
459                         data_off, frag_len);
460
461             if (err) {
462                 kfree_skb(skb);
463                 break;
464             }
465
466             IP6_INC_STATS(Ip6FragCreates);
467             IP6_INC_STATS(Ip6OutRequests);
468             err = NF_HOOK(PF_INET6, NF_IP6_LOCAL_OUT, skb, NULL,
469                         dst->dev, ip6_maybe_reroute);
470             if (err) {
471                 kfree_skb(last_skb);
472                 return err;
473             }
474         }
475
476     if (err) {
477         IP6_INC_STATS(Ip6FragFails);
478         kfree_skb(last_skb);
479         return -EFAULT;
480     }
481
482     hdr->payload_len = htons(unfrag_len + last_len - sizeof(struct ipv6hdr));
483
484     /*
485     *   update last_skb to reflect the getfrag we did
486     *   on start.

```

```

548     if (dst == NULL)
549         dst = ip6_route_output(sk, fl);
550
551     if (dst->error) {
552         IP6_INC_STATS(Ip6OutNoRoutes);
553         dst_release(dst);
554         return -ENETUNREACH;
555     }
556
557     if (fl->fl6_src == NULL) {
558         err = ipv6_get_saddr(dst, fl->fl6_dst, &saddr);
559
560         if (err) {
561             #if IP6_DEBUG >= 2
562                 printk(KERN_DEBUG "ip6_build_xmit:_"
563                     "no_available_source_address\n");
564             #endif
565             goto out;
566         }
567         fl->fl6_src = &saddr;
568     }
569     pktlength = length;
570
571     if (hlimit < 0) {
572         if (ipv6_addr_is_multicast(fl->fl6_dst))
573             hlimit = np->mcast_hops;
574         else
575             hlimit = np->hop_limit;
576     }
577     if (hlimit < 0)
578         hlimit = ((struct rt6_info *)dst)->rt6i_hoplimit;

```

Figure 604: IPv6 - Output: linux/net/ipv6/ip6_output.c

```

580
581     jumbolen = 0;
582
583     if (!sk->protinfo.af_inet.hdrincl) {
584         pktlength += sizeof(struct ipv6hdr);
585         if (opt)
586             pktlength += opt->opt_flen + opt->opt_nflen;
587
588         if (pktlength > 0xFFFF + sizeof(struct ipv6hdr)) {
589             /* Jumbo datagram.
590              * It is assumed, that in the case of hdrincl
591              * jumbo option is supplied by user.
592              */
593             pktlength += 8;
594             jumbolen = pktlength - sizeof(struct ipv6hdr);
595         }
596     }
597
598     mtu = dst->pmtu;
599     if (np->frag_size < mtu) {
600         if (np->frag_size)
601             mtu = np->frag_size;
602         else if (np->pmtudisc == IPV6_PMTUDISC_DONT)
603             mtu = IPV6_MIN_MTU;
604     }
605
606     /* Critical arithmetic overflow check.
607      * FIXME: may gcc optimize it out? --ANK (980726)
608      */
609     if (pktlength < length) {
610         ipv6_local_error(sk, EMSGSIZE, fl, mtu);
611         err = -EMSGSIZE;
612         goto out;
613     }
614
615     if (flags & MSG_CONFIRM)

```

Figure 605: IPv6 - Output: linux/net/ipv6/ip6_output.c

```

617
618     if (pktlength <= mtu) {
619         struct sk_buff *skb;
620         struct ipv6hdr *hdr;
621         struct net_device *dev = dst->dev;
622
623         err = 0;
624         if ( flags &MSG_PROBE)
625             goto out;
626
627         skb = sock_alloc_send_skb(sk, pktlength + 15 +
628                                 dev->hard_header_len,
629                                 flags & MSG_DONTWAIT, &err);
630
631         if (skb == NULL) {
632             IP6_INC_STATS(Ip6OutDiscards);
633             goto out;
634         }
635
636         skb->dst = dst_clone(dst);
637
638         skb_reserve(skb, (dev->hard_header_len + 15) & ~15);
639
640         hdr = (struct ipv6hdr *) skb->tail;
641         skb->nh.ipv6h = hdr;
642
643         if (!sk->protinfo.af_inet.hdrincl) {
644             ip6_bld_1(sk, skb, fl, hlimit,
645                     jumbolen ? sizeof(struct ipv6hdr) : pktlength);
646
647             if (opt || jumbolen) {
648                 u8 *prev_hdr = &hdr->nexthdr;
649                 prev_hdr = ipv6_build_nfrag_opts(skb, prev_hdr, opt,
650                                                 final_dst, jumbolen);
651                 if (opt && opt->opt_flen)
652                     ipv6_build_frag_opts(skb, prev_hdr, opt);
653             }
654         }

```

Figure 606: IPv6 - Output: linux/net/ipv6/ip6_output.c

```

654     skb_put(skb, length);
655     err = getfrag(data, &hdr->saddr,
656                 ((char *) hdr) + (pktlength - length),
657                 0, length);
658
659     if (!err) {
660         IP6_INC_STATS(Ip6OutRequests);
661         err = NF_HOOK(PF_INET6, NF_IP6_LOCAL_OUT, skb, NULL,
662                     dst->dev, ip6_maybe_reroute);
663     } else {
664         err = -EFAULT;
665         kfree_skb(skb);
666     }
667 } else {
668     if (sk->protinfo.af_inet.hdrincl || jumbolen ||
669         np->pmtudisc == IPV6_PMTUDISC_DO) {
670         ipv6_local_error(sk, EMSGSIZE, fl, mtu);
671         err = -EMSGSIZE;
672         goto out;
673     }
674
675     err = ip6_frag_xmit(sk, getfrag, data, dst, fl, opt, final_dst, hlimit,
676                       flags, length, mtu);
677 }
678
679 /*
680  *   cleanup
681  */
682 out:
683     ip6_dst_store(sk, dst, fl->nl_u.ip6_u.daddr == &np->daddr ? &np->daddr :
684                 NULL);
685     if (err > 0)
686         err = np->recverr ? net_xmit_errno(err) : 0;
687     return err;

```

Figure 607: IPv6 - Output: linux/net/ipv6/ip6_output.c

```

720 int ip6_forward(struct sk_buff *skb)
721 {
722     struct dst_entry *dst = skb->dst;
723     struct ipv6hdr *hdr = skb->nh.ipv6h;
724     struct inet6_skb_parm *opt =(struct inet6_skb_parm*)skb->cb;
725
726     if (ipv6_devconf.forwarding == 0 && opt->srct == 0)
727         goto error;
728
729     skb->ip_summed = CHECKSUM_NONE;
730
731     /*
732     *   We DO NOT make any processing on
733     *   RA packets, pushing them to user level AS IS
734     *   without ane WARRANTY that application will be able
735     *   to interpret them. The reason is that we
736     *   cannot make anything clever here.
737     *
738     *   We are not end-node, so that if packet contains
739     *   AH/ESP, we cannot make anything.
740     *   Defragmentation also would be mistake, RA packets
741     *   cannot be fragmented, because there is no warranty
742     *   that different fragments will go along one path. --ANK
743     */
744     if (opt->ra) {
745         u8 *ptr = skb->nh.raw + opt->ra;
746         if ( ip6_call_ra_chain (skb, (ptr[2]<<8) + ptr[3]))
747             return 0;
748     }
749
750     /*
751     *   check and decrement ttl
752     */
753     if (hdr->hop_limit <= 1) {
754         /* Force OUTPUT device used as source address */
755         skb->dev = dst->dev;
756         icmpv6_send(skb, ICMPV6_TIME_EXCEED, ICMPV6_EXC_HOPLIMIT,
757                    0, skb->dev);

```

Figure 608: IPv6 - Forward: linux/net/ipv6/ip6_output.c

```

760     kfree_skb(skb);
761     return -ETIMEDOUT;
762 }
763
764 /* IPv6 specs say nothing about it, but it is clear that we cannot
765    send redirects to source routed frames.
766 */
767 if (skb->dev == dst->dev && dst->neighbour && opt->srct == 0) {
768     struct in6_addr *target = NULL;
769     struct rt6_info *rt;
770     struct neighbour *n = dst->neighbour;
771
772     /*
773      *   incoming and outgoing devices are the same
774      *   send a redirect.
775      */
776
777     rt = (struct rt6_info *) dst;
778     if ((rt->rt6i_flags & RTF_GATEWAY))
779         target = (struct in6_addr*)&n->primary_key;
780     else
781         target = &hdr->daddr;
782
783     /* Limit redirects both by destination (here)
784        and by source (inside ndisc_send_redirect)
785        */
786     if (xrlim_allow(dst, 1*HZ))
787         ndisc_send_redirect(skb, n, target);
788 } else if (ipv6_addr_type(&hdr->saddr)&(IPV6_ADDR_MULTICAST|
789     IPV6_ADDR_LOOPBACK
790     |IPV6_ADDR_LINKLOCAL)) {

```

Figure 609: IPv6 - Forward: linux/net/ipv6/ip6_output.c


```

791         goto error;
792     }
793
794     if (skb->len > dst->pmtu) {
795         /* Again, force OUTPUT device used as source address */
796         skb->dev = dst->dev;
797         icmpv6_send(skb, ICMPV6_PKT_TOOBIG, 0, dst->pmtu, skb->dev);
798         IP6_INC_STATS_BH(Ip6InTooBigErrors);
799         kfree_skb(skb);
800         return -EMSGSIZE;
801     }
802
803     if (skb_cow(skb, dst->dev->hard_header_len))
804         goto drop;
805
806     hdr = skb->nh.ipv6h;
807
808     /* Mangling hops number delayed to point after skb COW */
809
810     hdr->hop_limit--;
811
812     IP6_INC_STATS_BH(Ip6OutForwDatagrams);
813     return NF_HOOK(PF_INET6, NF_IP6_FORWARD, skb, skb->dev, dst->dev,
814                  ip6_forward_finish);
815
816 error:
817     IP6_INC_STATS_BH(Ip6InAddrErrors);

```

Figure 610: IPv6 - Forward: linux/net/ipv6/ip6_output.c

```

122 int ipv6_setsockopt(struct sock *sk, int level, int optname, char *optval,
123                   int optlen)
124 {
125     struct ipv6_pinfo *np = &sk->net_pinfo.af_inet6;
126     int val, valbool;
127     int retv = -ENOPROTOOPT;
128
129     if (level == SOL_IP && sk->type != SOCK_RAW)
130         return udp_prot.setsockopt(sk, level, optname, optval, optlen);
131
132     if (level != SOL_IPV6)
133         goto out;
134
135     if (optval == NULL)
136         val=0;
137     else if (get_user(val, (int *) optval))
138         return -EFAULT;
139
140     valbool = (val!=0);
141
142     lock_sock(sk);
143
144     switch (optname) {

```

Figure 611: IPv6 - Socket Glue Set Socket Options:
linux/net/ipv6/ipv6_sockglue.c

```

146     case IPV6_ADDRFORM:
147         if (val == PF_INET) {
148             struct ipv6_txoptions *opt;
149             struct sk_buff *pktopt;
150
151             if (sk->protocol != IPPROTO_UDP &&
152                 sk->protocol != IPPROTO_TCP)
153                 break;
154
155             if (sk->state != TCP_ESTABLISHED) {
156                 retv = -ENOTCONN;
157                 break;
158             }
159
160             if (!(ipv6_addr_type(&np->daddr) & IPV6_ADDR_MAPPED)) {
161                 retv = -EADDRNOTAVAIL;
162                 break;
163             }
164
165             fl6_free_socklist (sk);
166             ipv6_sock_mc_close(sk);
167
168             if (sk->protocol == IPPROTO_TCP) {
169                 struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
170
171                 local_bh_disable ();
172                 sock_prot_dec_use(sk->prot);
173                 sock_prot_inc_use(&tcp_prot);
174                 local_bh_enable ();
175                 sk->prot = &tcp_prot;
176                 tp->af_specific = &ipv4_specific;
177                 sk->socket->ops = &inet_stream_ops;
178                 sk->family = PF_INET;
179                 tcp_sync_mss(sk, tp->pmtu_cookie);
180             } else {
181                 local_bh_disable ();
182                 sock_prot_dec_use(sk->prot);
183                 sock_prot_inc_use(&udp_prot);
184                 local_bh_enable ();
185                 sk->prot = &udp_prot;
186                 sk->socket->ops = &inet_dgram_ops;
187                 sk->family = PF_INET;
188             }
189             opt = xchg(&np->opt, NULL);
190             if (opt)
191                 sock_kfree_s (sk, opt, opt->tot_len);
192             pktopt = xchg(&np->pktoptions, NULL);
193             if (pktopt)
194                 kfree_skb(pktopt);
195
196             sk->destruct = inet_sock_destruct;
197 #ifdef INET_REFCNT_DEBUG
198             atomic_dec(&inet6_sock_nr);
199 #endif
200             MOD_DEC_USE_COUNT;
201             retv = 0;
202             break;
203         }
204         goto e_inval;

```

Figure 612: IPv6 - Socket Glue Set Socket Options:
linux/net/ipv6/ipv6_sockglue.c

```

206     case IPV6_PKTINFO:
207         np->rxopt.bits.rxinfo = valbool;
208         retv = 0;
209         break;
210
211     case IPV6_HOPLIMIT:
212         np->rxopt.bits.rxhlim = valbool;
213         retv = 0;
214         break;
215
216     case IPV6_RTHDR:
217         if (val < 0 || val > 2)
218             goto e_inval;
219         np->rxopt.bits.srcrt = val;
220         retv = 0;
221         break;
222
223     case IPV6_HOPOPTS:
224         np->rxopt.bits.hopopts = valbool;
225         retv = 0;
226         break;
227
228     case IPV6_AUTHHDR:
229         np->rxopt.bits.authhdr = valbool;
230         retv = 0;
231         break;
232
233     case IPV6_DSTOPTS:
234         np->rxopt.bits.dstopts = valbool;
235         retv = 0;
236         break;
237
238     case IPV6_FLOWINFO:
239         np->rxopt.bits.rxflow = valbool;
240         retv = 0;
241         break;

```

Figure 613: IPv6 - Socket Glue Set Socket Options:
linux/net/ipv6/ipv6_sockglue.c

```

243     case IPV6_PKTOPTIONS:
244     {
245         struct ipv6_txoptions *opt = NULL;
246         struct msghdr msg;
247         struct flowi fl ;
248         int junk;
249
250         fl . fl6_flowlabel = 0;
251         fl . oif = sk->bound_dev_if;
252
253         if (optlen == 0)
254             goto update;
255
256         /* 1K is probably excessive
257          * 1K is surely not enough, 2K per standard header is 16K.
258          */
259         retv = -EINVAL;
260         if (optlen > 64*1024)
261             break;
262
263         opt = sock_kmalloc(sk, sizeof(*opt) + optlen, GFP_KERNEL);
264         retv = -ENOBUFS;
265         if (opt == NULL)
266             break;
267
268         memset(opt, 0, sizeof(*opt));
269         opt->tot_len = sizeof(*opt) + optlen;
270         retv = -EFAULT;
271         if (copy_from_user(opt+1, optval, optlen))
272             goto done;
273
274         msg.msg_controllen = optlen;
275         msg.msg_control = (void*)(opt+1);
276
277         retv = datagram_send_ctl(&msg, &fl, opt, &junk);
278         if (retv)
279             goto done;
280     update:
281         retv = 0;
282         if (sk->type == SOCK_STREAM) {
283             if (opt) {
284                 struct tcp_opt *tp = &sk->tp_pinfo.af_tcp;
285                 if (!(1 << sk->state) & (TCPF_LISTEN | TCPF_CLOSE))
286                     && sk->daddr != LOOPBACK4_IPV6) {
287                     tp->ext_header_len = opt->opt_len + opt->
288                         opt_nflen;
289                     tcp_sync_mss(sk, tp->pmtu_cookie);
290                 }
291                 opt = xchg(&np->opt, opt);
292                 sk_dst_reset(sk);
293             } else {
294                 write_lock(&sk->dst_lock);
295                 opt = xchg(&np->opt, opt);
296                 write_unlock(&sk->dst_lock);
297                 sk_dst_reset(sk);

```

Figure 614: IPv6 - Socket Glue Set Socket Options:
linux/net/ipv6/ipv6_sockglue.c

```

298         }
299
300 done:
301         if (opt)
302             sock_kfree_s(sk, opt, opt->tot_len);
303         break;
304     }
305     case IPV6_UNICAST_HOPS:
306         if (val > 255 || val < -1)
307             goto e_inval;
308         np->hop_limit = val;
309         retv = 0;
310         break;
311
312     case IPV6_MULTICAST_HOPS:
313         if (sk->type == SOCK_STREAM)
314             goto e_inval;
315         if (val > 255 || val < -1)
316             goto e_inval;
317         np->mcast_hops = val;
318         retv = 0;
319         break;
320
321     case IPV6_MULTICAST_LOOP:
322         np->mc_loop = valbool;
323         retv = 0;
324         break;
325
326     case IPV6_MULTICAST_IF:
327         if (sk->type == SOCK_STREAM)
328             goto e_inval;
329         if (sk->bound_dev_if && sk->bound_dev_if != val)
330             goto e_inval;

```

Figure 615: IPv6 - Socket Glue Set Socket Options:
linux/net/ipv6/ipv6_sockglue.c

```

332         if (_dev_get_by_index(val) == NULL) {
333             retv = -ENODEV;
334             break;
335         }
336         np->mcast_oif = val;
337         retv = 0;
338         break;
339     case IPV6_ADD_MEMBERSHIP:
340     case IPV6_DROP_MEMBERSHIP:
341     {
342         struct ipv6_mreq mreq;
343
344         retv = -EFAULT;
345         if (copy_from_user(&mreq, optval, sizeof(struct ipv6_mreq)))
346             break;

```

Figure 616: IPv6 - Socket Glue Set Socket Options:
linux/net/ipv6/ipv6_sockglue.c

```

347         if (optname == IPV6_ADD_MEMBERSHIP)
348             retv = ipv6_sock_mc_join(sk, mreq.ipv6mr_ifindex, &mreq.
349                 ipv6mr_multiaddr);
350         else
351             retv = ipv6_sock_mc_drop(sk, mreq.ipv6mr_ifindex, &mreq.
352                 ipv6mr_multiaddr);
353     }
354     case IPV6_ROUTER_ALERT:
355         retv = ip6_ra_control(sk, val, NULL);
356         break;
357     case IPV6_MTU_DISCOVER:
358         if (val < 0 || val > 2)
359             goto e_inval;
360         np->pmtudisc = val;
361         retv = 0;
362         break;
363     case IPV6_MTU:
364         if (val && val < IPV6_MIN_MTU)
365             goto e_inval;
366         np->frag_size = val;
367         retv = 0;
368         break;
369     case IPV6_RECVERR:
370         np->recvrr = valbool;
371         if (!val)
372             skb_queue_purge(&sk->error_queue);
373         retv = 0;
374         break;
375     case IPV6_FLOWINFO_SEND:
376         np->sndflow = valbool;
377         retv = 0;
378         break;
379     case IPV6_FLOWLABEL_MGR:
380         retv = ipv6_flowlabel_opt(sk, optval, optlen);
381         break;
382
383     #ifdef CONFIG_NETFILTER
384     default:
385         retv = nf_setsockopt(sk, PF_INET6, optname, optval,
386                             optlen);
387         break;
388     #endif
389 }
390 release_sock(sk);
391

```

Figure 617: IPv6 - Socket Glue Set Socket Options:
linux/net/ipv6/ipv6_sockglue.c


```
274 static struct ip6_flowlabel *
275 fl_create (struct in6_flowlabel_req *freq, char *optval, int optlen, int *err_p)
276 {
277     struct ip6_flowlabel *fl;
278     int olen;
279     int addr_type;
280     int err;
281
282     err = -ENOMEM;
283     fl = kmalloc(sizeof(*fl), GFP_KERNEL);
284     if (fl == NULL)
285         goto done;
286     memset(fl, 0, sizeof(*fl));
```

Figure 618: IPv6 Flow Label Creation: `linux/net/ipv6/ip6_flowlabel.c`

```

288     olen = optlen - CMSG_ALIGN(sizeof(*freq));
289     if (olen > 0) {
290         struct msghdr msg;
291         struct flowi flowi;
292         int junk;
293
294         err = -ENOMEM;
295         fl->opt = kmalloc(sizeof(*fl->opt) + olen, GFP_KERNEL);
296         if (fl->opt == NULL)
297             goto done;
298
299         memset(fl->opt, 0, sizeof(*fl->opt));
300         fl->opt->tot_len = sizeof(*fl->opt) + olen;
301         err = -EFAULT;
302         if (copy_from_user(fl->opt+1, optval+CMSG_ALIGN(sizeof(*freq)), olen)
303             )
304             goto done;
305
306         msg.msg_controllen = olen;
307         msg.msg_control = (void*)(fl->opt+1);
308         flowi.oif = 0;
309
310         err = datagram_send_ctl(&msg, &flowi, fl->opt, &junk);
311         if (err)
312             goto done;
313         err = -EINVAL;
314         if (fl->opt->opt_nflen)
315             goto done;
316         if (fl->opt->opt_nflen == 0) {
317             kfree(fl->opt);
318             fl->opt = NULL;
319         }
320     }

```

Figure 619: IPv6 Flow Label Creation: `linux/net/ipv6/ip6_flowlabel.c`

```

321     fl->expires = jiffies;
322     err = fl6_renew(fl, freq->flr_linger, freq->flr_expires);
323     if (err)
324         goto done;
325     fl->share = freq->flr_share;
326     addr_type = ipv6_addr_type(&freq->flr_dst);
327     if ((addr_type & IPV6_ADDR_MAPPED)
328         || addr_type == IPV6_ADDR_ANY)
329         goto done;
330     ipv6_addr_copy(&fl->dst, &freq->flr_dst);
331     atomic_set(&fl->users, 1);
332     switch (fl->share) {
333     case IPV6_FL_S_EXCL:
334     case IPV6_FL_S_ANY:
335         break;
336     case IPV6_FL_S_PROCESS:
337         fl->owner = current->pid;
338         break;
339     case IPV6_FL_S_USER:
340         fl->owner = current->euid;
341         break;
342     default:
343         err = -EINVAL;
344         goto done;
345     }
346     return fl;
347
348 done:
349     if (fl)
350         fl_free(fl);
351     *err_p = err;
352     return NULL;
353 }

```

Figure 620: IPv6 Flow Label Creation: `linux/net/ipv6/ip6_flowlabel.c`

```

700         if (opt2->hopopt)
701             *((char*)&opt2->hopopt) += dif;
702         if (opt2->dst0opt)
703             *((char*)&opt2->dst0opt) += dif;
704         if (opt2->dst1opt)
705             *((char*)&opt2->dst1opt) += dif;
706         if (opt2->auth)
707             *((char*)&opt2->auth) += dif;
708         if (opt2->srcrt)
709             *((char*)&opt2->srcrt) += dif;
710     }
711     return opt2;
712 }
713
714
715 /*
716  * find out if nexthdr is a well-known extension header or a protocol
717  */
718
719 int ipv6_ext_hdr(u8 nexthdr)
720 {
721     /*
722      * find out if nexthdr is an extension header or a protocol
723      */
724     return ( (nexthdr == NEXTHDR_HOP) ||
725             (nexthdr == NEXTHDR_ROUTING) ||
726             (nexthdr == NEXTHDR_FRAGMENT) ||
727             (nexthdr == NEXTHDR_AUTH) ||
728             (nexthdr == NEXTHDR_NONE) ||
729             (nexthdr == NEXTHDR_DEST) );
730 }
731
732 /*
733  * Skip any extension headers. This is used by the ICMP module.
734  *
735  * Note that strictly speaking this conflicts with RFC1883 4.0:
736  * ...The contents and semantics of each extension header determine whether
737  * or not to proceed to the next header. Therefore, extension headers must
738  * be processed strictly in the order they appear in the packet; a
739  * receiver must not, for example, scan through a packet looking for a
740  * particular kind of extension header and process that header prior to

```

Figure 621: IPv6 Extension Headers: linux/net/ipv6/exthdrs.c

```

742 *
743 * We do exactly this. This is a protocol bug. We can't decide after a
744 * seeing an unknown discard-with-error flavour TLV option if it's a
745 * ICMP error message or not (errors should never be send in reply to
746 * ICMP error messages).
747 *
748 * But I see no other way to do this. This might need to be reexamined
749 * when Linux implements ESP (and maybe AUTH) headers.
750 * --AK
751 *
752 * This function parses (probably truncated) exthdr set "hdr"
753 * of length "len". "nexthdrp" initially points to some place,
754 * where type of the first header can be found.
755 *
756 * It skips all well-known exthdrs, and returns pointer to the start
757 * of unparsable area i.e. the first header with unknown type.
758 * If it is not NULL *nexthdr is updated by type/protocol of this header.
759 *
760 * NOTES: - if packet terminated with NEXTHDR_NONE it returns NULL.
761 *         - it may return pointer pointing beyond end of packet,
762 *         - if the last recognized header is truncated in the middle.
763 *         - if packet is truncated, so that all parsed headers are skipped,
764 *         - it returns NULL.
765 *         - First fragment header is skipped, not-first ones
766 *         - are considered as unparsable.
767 *         - ESP is unparsable for now and considered like
768 *         - normal payload protocol.
769 *         - Note also special handling of AUTH header. Thanks to IPsec wizards.
770 *
771 * --ANK (980726)

```

Figure 622: IPv6 Extension Headers: `linux/net/ipv6/exthdrs.c`

```

54  /*
55     This version of net/ipv6/sit.c is cloned of net/ipv4/ip_gre.c
56
57     For comments look at net/ipv4/ip_gre.c --ANK
58  */
59
60  #define HASH_SIZE 16
61  #define HASH(addr) ((addr^(addr>>4))&0xF)
62
63  static int ipip6_fb_tunnel_init (struct net_device *dev);
64  static int ipip6_tunnel_init (struct net_device *dev);
65
66  static struct net_device ipip6_fb_tunnel_dev = {
67      "sit0", 0x0, 0x0, 0x0, 0x0, 0, 0, 0, 0, 0, NULL, ipip6_fb_tunnel_init,
68  };
69
70  static struct ip_tunnel ipip6_fb_tunnel = {
71      NULL, &ipip6_fb_tunnel_dev, {0}, 0, 0, 0, 0, 0, 0, { "sit0", }
72  };
73
74  static struct ip_tunnel *tunnels_r_l [HASH_SIZE];
75  static struct ip_tunnel *tunnels_r [HASH_SIZE];
76  static struct ip_tunnel *tunnels_l [HASH_SIZE];
77  static struct ip_tunnel *tunnels_wc [1];
78  static struct ip_tunnel **tunnels[4] = { tunnels_wc, tunnels_l, tunnels_r, tunnels_r_l };

```

Figure 623: SIT: linux/net/ipv6/sit.c

```

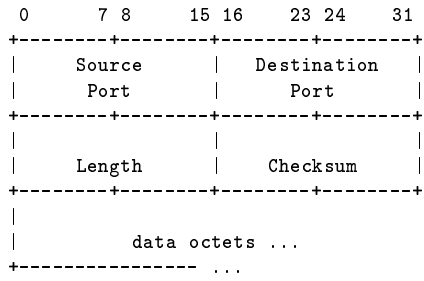
82 static struct ip_tunnel * ipip6_tunnel_lookup(u32 remote, u32 local)
83 {
84     unsigned h0 = HASH(remote);
85     unsigned h1 = HASH(local);
86     struct ip_tunnel *t;
87
88     for (t = tunnels_r_l[h0^h1]; t; t = t->next) {
89         if (local == t->parms.iph.saddr &&
90             remote == t->parms.iph.daddr && (t->dev->flags&IFF_UP))
91             return t;
92     }
93     for (t = tunnels_r[h0]; t; t = t->next) {
94         if (remote == t->parms.iph.daddr && (t->dev->flags&IFF_UP))
95             return t;
96     }
97     for (t = tunnels_l[h1]; t; t = t->next) {
98         if (local == t->parms.iph.saddr && (t->dev->flags&IFF_UP))
99             return t;
100    }
101    if ((t = tunnels_wc[0]) != NULL && (t->dev->flags&IFF_UP))
102        return t;
103    return NULL;
104 }

```

Figure 624: SIT: linux/net/ipv6/sit.c

```
tcpmux      1/tcp          # TCP port service multiplexer
echo        7/tcp
echo        7/udp
discard     9/tcp          sink null
discard     9/udp          sink null
sysstat     11/tcp         users
daytime     13/tcp
daytime     13/udp
netstat     15/tcp
qotd        17/tcp         quote
msp         18/tcp         # message send protocol
msp         18/udp         # message send protocol
chargen    19/tcp         ttytst source
chargen    19/udp         ttytst source
ftp-data    20/tcp
ftp         21/tcp
fsp         21/udp         fspd
ssh         22/tcp         # SSH Remote Login Protocol
ssh         22/udp         # SSH Remote Login Protocol
telnet     23/tcp
```

Figure 625: A sample from the /etc/services file



```

922 struct proto_ops inet_stream_ops = {
923     family:      PF_INET,
924
925     release:     inet_release ,
926     bind:        inet_bind ,
927     connect:     inet_stream_connect ,
928     socketpair:  sock_no_socketpair ,
929     accept:      inet_accept ,
930     getname:     inet_getname,
931     poll:        tcp_poll ,
932     ioctl:       inet_ioctl ,
933     listen:      inet_listen ,
934     shutdown:    inet_shutdown,
935     setsockopt:  inet_setsockopt ,
936     getsockopt:  inet_getsockopt ,
937     sendmsg:     inet_sendmsg,
938     recvmsg:     inet_recvmsg,
939     mmap:        sock_no_mmap,
940     sendpage:    tcp_sendpage
941 };
942
943 struct proto_ops inet_dgram_ops = {

```

Figure 626: linkage structures and functions (i)

```

1008 struct proto udp_prot = {
1009     name:      "UDP",
1010     close:     udp_close,
1011     connect:   udp_connect,
1012     disconnect: udp_disconnect,
1013     ioctl:    udp_ioctl,
1014     setsockopt: ip_setsockopt,
1015     getsockopt: ip_getsockopt,
1016     sendmsg:   udp_sendmsg,
1017     recvmsg:   udp_recvmsg,
1018     backlog_rcv: udp_queue_rcv_skb,
1019     hash:      udp_v4_hash,
1020     unhash:    udp_v4_unhash,
1021     get_port:  udp_v4_get_port,
1022 };

```

```

2165 struct proto tcp_prot = {
2166     name:      "TCP",
2167     close:     tcp_close,
2168     connect:   tcp_v4_connect,
2169     disconnect: tcp_disconnect,
2170     accept:    tcp_accept,
2171     ioctl:    tcp_ioctl,
2172     init:     tcp_v4_init_sock,
2173     destroy:   tcp_v4_destroy_sock,
2174     shutdown: tcp_shutdown,
2175     setsockopt: tcp_setsockopt,
2176     getsockopt: tcp_getsockopt,
2177     sendmsg:   tcp_sendmsg,
2178     recvmsg:   tcp_recvmsg,
2179     backlog_rcv: tcp_v4_do_rcv,
2180     hash:      tcp_v4_hash,
2181     unhash:    tcp_unhash,
2182     get_port:  tcp_v4_get_port,
2183 };

```

Figure 627: linkage structures and functions (ii)

```
1865 struct tcp_func ipv4_specific = {
1866     ip_queue_xmit,
1867     tcp_v4_send_check,
1868     tcp_v4_rebuild_header,
1869     tcp_v4_conn_request,
1870     tcp_v4_syn_recv_sock,
1871     tcp_v4_hash_connecting,
1872     tcp_v4_remember_stamp,
1873     sizeof (struct iphdr),
1874
1875     ip_setsockopt ,
1876     ip_getsockopt ,
1877     v4_addr2sockaddr,
1878     sizeof (struct sockaddr_in)
1879 };
```

```

487 struct sock {
488     /* Socket demultiplex comparisons on incoming packets. */
489     __u32          daddr;      /* Foreign IPv4 addr
490     */
491     __u32          rcv_saddr;  /* Bound local IPv4 addr
492     */
493     __u16          dport;      /* Destination port
494     */
495     unsigned short num;        /* Local port
496     */
497     int            bound_dev_if; /* Bound device index if != 0
498     */
499
500     /* Main hash linkage for various protocol lookup tables. */
501     struct sock    *next;
502     struct sock    **pprev;
503     struct sock    *bind_next;
504     struct sock    **bind_pprev;

```

Figure 628: start of the sock structure linux{2.4.4/include/net/sock.h

```

1 socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 3
2 connect(3, {sin_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("
   xxx.xxx.xxx.xxx")}}, 28) = 0
3 send(3, "xE\1\0\0\1\0\0\0\0\0\0\3www\5lboro\2ac\2uk\0\0\1\0"... , 33, 0)
   = 33
4 gettimeofday({986376794, 91755}, NULL) = 0
5 poll([{fd=3, events=POLLIN, revents=POLLIN}], 1, 5000) = 1
6 recvfrom(3, "xE\205\200\0\1\0\2\0\5\0\4\3www\5lboro\2ac\2uk\0\0\1\0"
   ..., 1024, 0, {sin_family=AF_INET, sin_port=htons(53), sin_addr=
   inet_addr("xxx.xxx.xxx.xxx")}}, [16]) = 255
7 close(3)

```

Figure 629: Strace of a DNS lookup

```

711 int udp_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
712 {
713     struct sockaddr_in *usin = (struct sockaddr_in *) uaddr;
714     struct rtable *rt;
715     int err;
716
717
718     if (addr_len < sizeof(*usin))
719         return -EINVAL;
720
721     if (usin->sin_family != AF_INET)
722         return -EAFNOSUPPORT;
723
724     sk_dst_reset(sk);
725
726     err = ip_route_connect(&rt, usin->sin_addr.s_addr, sk->saddr,
727                          sk->protoinfo.af_inet.tos | sk->localroute, sk->
728                          bound_dev_if);
729     if (err)
730         return err;
731     if ((rt->rt_flags & RTCF_BROADCAST) && !sk->broadcast) {
732         ip_rt_put(rt);
733         return -EACCES;
734     }
735     if (!sk->saddr)
736         sk->saddr = rt->rt_src;      /* Update source address */
737     if (!sk->rcv_saddr)
738         sk->rcv_saddr = rt->rt_src;
739     sk->daddr = rt->rt_dst;
740     sk->dport = usin->sin_port;
741     sk->state = TCP_ESTABLISHED;
742     sk->protoinfo.af_inet.id = jiffies;
743
744     sk_dst_set(sk, &rt->u.dst);
745     return(0);
746 }

```

Figure 630: UDP connect function linux{2.4.4/net/ipv4/udp.c

```

747 int udp_disconnect(struct sock *sk, int flags)
748 {
749     /*
750      *    1003.1g - break association.
751      */
752
753     sk->state = TCP_CLOSE;
754     sk->daddr = 0;
755     sk->dport = 0;
756     sk->bound_dev_if = 0;
757     if (!(sk->userlocks&SOCK_BINDADDR_LOCK)) {
758         sk->rcv_saddr = 0;
759         sk->saddr = 0;
760 #if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
761         memset(&sk->net_pinfo.af_inet6.saddr, 0, 16);
762         memset(&sk->net_pinfo.af_inet6.rcv_saddr, 0, 16);
763 #endif
764     }
765     if (!(sk->userlocks&SOCK_BINDPORT_LOCK)) {
766         sk->prot->unhash(sk);
767         sk->sport = 0;
768     }
769     sk_dst_reset(sk);
770     return 0;
771 }

```

Figure 631: UDP lookup disconnect function `linux{2.4.4/net/ipv4/udp.c}`


```

535     ufh.saddr = rt->rt_src;
536     if (!ipc.addr)
537         ufh.daddr = ipc.addr = rt->rt_dst;
538     ufh.uh.len = htons(ulen);
539     ufh.uh.check = 0;
540     ufh.iov = msg->msg_iov;
541     ufh.wcheck = 0;
542
543     /* RFC1122: OK. Provides the checksumming facility (MUST) as per */
544     /* 4.1.3.4. It's configurable by the application via setsockopt() */
545     /* (MAY) and it defaults to on (MUST). */
546
547     err = ip_build_xmit(sk,
548                       (sk->no_check == UDP_CSUM_NOXMIT ?
549                        udp_getfrag_nosum :
550                        udp_getfrag),
551                       &ufh, ulen, &ipc, rt, msg->msg_flags);
552
553 out:
554     ip_rt_put(rt);
555     if (free)
556         kfree(ipc.opt);
557     if (!err) {
558         UDP_INC_STATS_USER(UdpOutDatagrams);
559         return len;
560     }
561     return err;
562
563 do_confirm:

```

Figure 632: Building and sending a udp packet (from `udp_sendmsg`)

```

211 struct sock *udp_v4_lookup_longway(u32 saddr, u16 sport, u32 daddr, u16 dport, int dif)
212 {
213     struct sock *sk, *result = NULL;
214     unsigned short hnum = ntohs(dport);
215     int badness = -1;
216
217     for(sk = udp_hash[hnum & (UDP_HTABLE_SIZE - 1)]; sk != NULL; sk = sk->
218         next) {
219         if(sk->num == hnum) {
220             int score = 0;
221             if(sk->rcv_saddr) {
222                 if(sk->rcv_saddr != daddr)
223                     continue;
224                 score++;
225             }
226             if(sk->daddr) {
227                 if(sk->daddr != saddr)
228                     continue;
229                 score++;
230             }
231             if(sk->dport) {
232                 if(sk->dport != sport)
233                     continue;
234                 score++;
235             }
236             if(sk->bound_dev_if) {
237                 if(sk->bound_dev_if != dif)
238                     continue;
239                 score++;
240             }
241             if(score == 4) {
242                 result = sk;
243                 break;
244             } else if(score > badness) {
245                 result = sk;
246                 badness = score;
247             }
248         }
249     }
250     return result;
}

```

Figure 633: UDP lookup socket function `linux{2.4.4}/net/ipv4/udp.c`

```

899     if (udp_checksum_init(skb, uh, ulen, saddr, daddr) < 0)
900         goto csum_error;
901
902     if (rt->rt_flags & (RTCF_BROADCAST|RTCF_MULTICAST))
903         return udp_v4_mcast_deliver(skb, uh, saddr, daddr);
904
905     sk = udp_v4_lookup(saddr, uh->source, daddr, uh->dest, skb->dev->ifindex);
906
907     if (sk != NULL) {
908         udp_queue_rcv_skb(sk, skb);
909         sock_put(sk);
910         return 0;
911     }
912
913     /* No socket. Drop packet silently, if checksum is wrong */
914     if (udp_checksum_complete(skb))
915         goto csum_error;
916
917     UDP_INC_STATS_BH(UdpNoPorts);
918     icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0);
919
920     /*
921     * Hmm. We got an UDP packet to a port to which we
922     * don't wanna listen. Ignore it.
923     */
924     kfree_skb(skb);
925     return(0);

```

Figure 634: UDP actions after lookup `linux{2.4.4}/net/ipv4/udp.c`

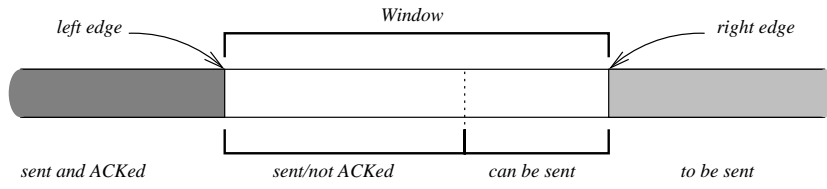


Figure 635: Sliding window operation

```

1  ...
2  connect(3, {sin_family=AF_INET, sin_port=htons(80), sin_addr=inet_addr("x
   .x.x.x")}), 16) = 0
3  ioctl(3, FIONBIO, [0]) = 0
4  write(3, "GET_/test/index.html_HTTP/1.0\r\nH"..., 1140) = 1140
5  gettimeofday({990094476, 833725}, NULL) = 0
6  time(NULL) = 990094476
7  select(1024, [3], NULL, NULL, {0, 100000}) = 1 (in [3], left {0, 90000})
8  read(3, "HTTP/1.1_200_OK\r\nDate:_Thu,_17_M"..., 1535) = 450
9  gettimeofday({990094476, 838885}, NULL) = 0
10 brk(0x8196000) = 0x8196000
11 brk(0x8198000) = 0x8198000
12 brk(0x819b000) = 0x819b000
13 brk(0x81a0000) = 0x81a0000
14 gettimeofday({990094476, 840321}, NULL) = 0
15 time(NULL) = 990094476
16 select(1024, [3], NULL, NULL, {0, 100000}) = 1 (in [3], left {0, 100000})
17 read(3, "", 4096) = 0
18 close(3) = 0
19 fstat64(1, {st_mode=S_IFCHR|0666, st_rdev=makedev(3, 2), ...}) = 0
20 mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
   = 0x402f2000
21 ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
22 write(1, "\n", 1) = 1
23 write(1, "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"..., 47) = 47
24 write(1, "\n", 1) = 1
25 write(1, "XXXHere_is_an_image_[test.png]\n", 31) = 31
26 ...

```

Figure 636: Strace output from a web browse with lynx

```

From client to server
1 GET /test/test.html HTTP/1.0
2 Host: server
3 Accept: text/html, text/plain, application/mac-binhex40, application/
  msword, application/pdf, application/vnd.ms-excel, application/
  postscript, application/x-dvi, application/frame-idraw, audio/mod
  , image/*, video/*, video/mpeg, application/pgp
4 Accept: application/pgp, application/pdf, message/partial, message/external-body
  , application/postscript, x-be2, application/andrew-inset, text/richtext,
  text/enriched, x-sun-attachment, audio-file, postscript-file, default,
  mail-file
5 Accept: sun-deskset-message, application/x-metamail-patch, application/msword,
  text/sgml, video/mpeg, image/jpeg, image/tiff, image/x-rgb, image/png,
  image/x-xbitmap, image/x-xbm, image/gif, application/postscript, */*;q
  =0.01
6 Accept-Encoding: gzip, compress
7 Accept-Language: en
8 User-Agent: Lynx/2.8.4dev.16 libwww-FM/2.14 SSL-MM/1.4.1 OpenSSL/0.9.6

From server to client
1 HTTP/1.1 200 OK
2 Date: Thu, 17 May 2001 10:15:15 GMT
3 Server: Apache/1.3.12 (Unix) (Red Hat/Linux) mod_ssl/2.6.6 OpenSSL/0.9.5a
  DAV/1.0.1 PHP/4.0.1pl2 mod_perl/1.24
4 Last-Modified: Thu, 17 May 2001 09:22:34 GMT
5 ETag: "7c1c3-79-3b03985a"
6 Accept-Ranges: bytes
7 Content-Length: 121
8 Connection: close
9 Content-Type: text/html
10
11 <html>
12 <head>
13 </head>
14 <body>
15
16 <h1> Test HTML page</h1>
17
18 <p>Here is an image
19
20 </img>
21
22 </body>
23 </html>

```

Figure 637: tcpflow output from a web browsing session

```
1 browser.42978 > server.http: S 799139388:799139388(0) win 5840 <mss 1460,
  sackOK,timestamp 69053329 0,nop,wscale 0> (DF)
2 server.http > browser.42978: S 846949739:846949739(0) ack 799139389 win
  32120 <mss 1460,sackOK,timestamp 188746286 69053329,nop,wscale 0> (
  DF)
3 browser.42978 > server.http: . 1:1(0) ack 1 win 5840 <nop,nop,timestamp
  69053329 188746286> (DF)
4 browser.42978 > server.http: P 1:1141(1140) ack 1 win 5840 <nop,nop,
  timestamp 69053329 188746286> (DF)
5 server.http > browser.42978: . 1:1(0) ack 1141 win 32120 <nop,nop,
  timestamp 188746286 69053329> (DF)
6 server.http > browser.42978: P 1:451(450) ack 1141 win 32120 <nop,nop,
  timestamp 188746286 69053329> (DF)
7 browser.42978 > server.http: . 1141:1141(0) ack 451 win 6432 <nop,nop,
  timestamp 69053330 188746286> (DF)
8 server.http > browser.42978: F 451:451(0) ack 1141 win 32120 <nop,nop,
  timestamp 188746287 69053329> (DF)
9 browser.42978 > server.http: F 1141:1141(0) ack 452 win 6432 <nop,nop,
  timestamp 69053330 188746287> (DF)
10 server.http > browser.42978: . 452:452(0) ack 1142 win 32120 <nop,nop,
  timestamp 188746287 69053330> (DF)
```

```

1  ...
2  socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
3  setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
4  bind(3, {sin_family=AF_INET, sin_port=htons(5656), sin_addr=inet_addr("
      0.0.0.0")}}, 16) = 0
5  listen(3, 1) = 0
6  rt_sigaction(SIGALRM, {SIG_IGN}, {SIG_DFL}, 8) = 0
7  alarm(0) = 0
8  accept(3, {sin_family=AF_INET, sin_port=htons(43036), sin_addr=inet_addr(
      "127.0.0.1")}}, [16]) = 4
9  rt_sigaction(SIGALRM, {SIG_IGN}, {SIG_IGN}, 8) = 0
10 alarm(0) = 0
11 close(3) = 0
12 getsockname(4, {sin_family=AF_INET, sin_port=htons(5656), sin_addr=
      inet_addr("127.0.0.1")}}, [16]) = 0
13 select(16, [0 4], NULL, NULL, NULL) = 1 (in [4])
14 read(4, "", 8192) = 0
15 close(4) = 0
16 ...

```

Figure 638: strace of a server process

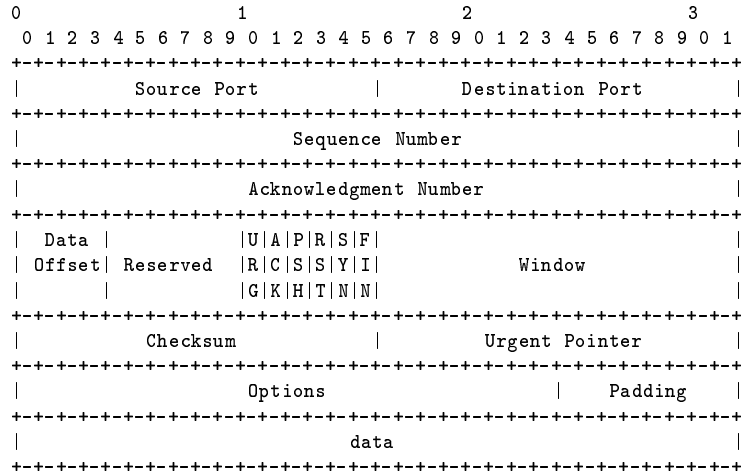


Figure 639: The TCP header

LISTEN - represents waiting for a connection request from any remote TCP and port.

SYN-SENT - represents waiting for a matching connection request after having sent a connection request.

SYN-RECEIVED - represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.

ESTABLISHED - represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.

FIN-WAIT-1 - represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.

FIN-WAIT-2 - represents waiting for a connection termination request from the remote TCP.

CLOSE-WAIT - represents waiting for a connection termination request from the local user.

CLOSING - represents waiting for a connection termination request acknowledgment from the remote TCP.

LAST-ACK - represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (which includes an acknowledgment of its connection termination request).

TIME-WAIT - represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.

CLOSED - represents no connection state at all.

Figure 640: TCP States (from RFC793)

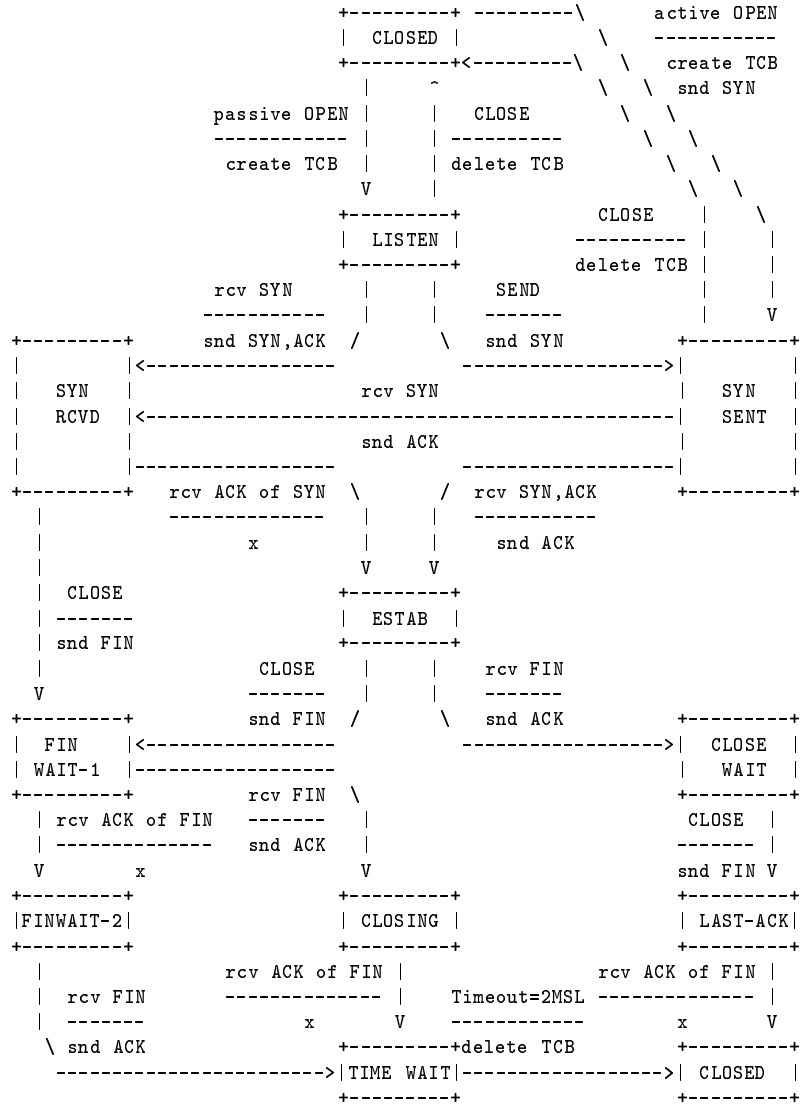


Figure 641: The TCP Connection creation state diagram (from RFC793)

```
1 struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
```

Figure 642: Assignment to ^{tp}

```

248 struct tcp_opt {
249     int      tcp_header_len; /* Bytes of tcp header to send      */
250
251     /*
252     *   Header prediction flags
253     *   0x5?10 << 16 + snd_wnd in net byte order
254     */
255     __u32    pred_flags;
256
257     /*
258     *   RFC793 variables by their proper names. This means you can
259     *   read the code and the spec side by side (and laugh ...)
260     *   See RFC793 and RFC1122. The RFC writes these in capitals.
261     */
262     __u32    rcv_nxt;        /* What we want to receive next      */
263     __u32    snd_nxt;        /* Next sequence we send              */
264
265     __u32    snd_una;        /* First byte we want an ack for      */
266     __u32    snd_sml;        /* Last byte of the most recently transmitted small
267     packet */
268     __u32    rcv_tstamp;     /* timestamp of last received ACK (for keepalives)
269     */
270     __u32    lsndtime;       /* timestamp of last sent data packet (for restart
271     window) */
272
273     /* Delayed ACK control data */
274     struct {
275         __u8    pending;     /* ACK is pending */
276         __u8    quick;       /* Scheduled number of quick acks     */
277         __u8    pingpong;    /* The session is interactive         */
278         __u8    blocked;     /* Delayed ACK was blocked by socket lock*/
279         __u32    ato;         /* Predicted tick of soft clock       */
280         unsigned long timeout; /* Currently scheduled timeout        */
281         __u32    lrcvtime;    /* timestamp of last received data packet*/
282         __u16    last_seg_size; /* Size of last incoming segment      */
283         __u16    rcv_mss;     /* MSS used for delayed ACK decisions */
284     } ack;
285
286     /* Data for direct copy to user */
287     struct {
288         struct sk_buff_head    prequeue;
289         int                    memory;
290         struct task_struct     *task;
291     };
292 }

```

tcp_opt
Figure 643: (i)

```

288         struct iovec      *iov;
289         int                len;
290     } ucopy;
291
292     __u32  snd_wll;      /* Sequence for window update */
293     __u32  snd_wnd;     /* The window we expect to receive */
294     __u32  max_window; /* Maximal window ever seen from peer */
295     __u32  pmtu_cookie; /* Last pmtu seen by socket */
296     __u16  mss_cache;  /* Cached effective mss, not including SACKS */
297     __u16  mss_clamp;  /* Maximal mss, negotiated at connection setup */
298     __u16  ext_header_len; /* Network protocol overhead (IP/IPv6 options) */
299     __u8   ca_state;   /* State of fast-retransmit machine */
300     __u8   retransmits; /* Number of unrecovered RTO timeouts. */
301
302     __u8   reordering; /* Packet reordering metric. */
303     __u8   queue_shrunk; /* Write queue has been shrunk recently. */
304     __u8   defer_accept; /* User waits for some data after accept() */
305
306     /* RTT measurement */
307     __u8   backoff; /* backoff */
308     __u32  srtt; /* smothed round trip time << 3 */
309     __u32  mdev; /* medium deviation */
310     __u32  mdev_max; /* maximal mdev for the last rtt period */
311     __u32  rttvar; /* smoothed mdev_max */
312     __u32  rtt_seq; /* sequence number to update rttvar */
313     __u32  rto; /* retransmit timeout */
314
315     __u32  packets_out; /* Packets which are "in flight" */
316     __u32  left_out; /* Packets which leaved network */
317     __u32  retrans_out; /* Retransmitted packets out */
318
319
320     /*
321     * Slow start and congestion control (see also Nagle, and Karn & Partridge)
322     */
323     __u32  snd_ssthresh; /* Slow start size threshold */
324     __u32  snd_cwnd; /* Sending congestion window */
325     __u16  snd_cwnd_cnt; /* Linear increase counter */
326     __u16  snd_cwnd_clamp; /* Do not allow snd_cwnd to grow above this */
327     __u32  snd_cwnd_used;

```

tcp_opt

Figure 644: (ii)

```

328     __u32    snd_cwnd_stamp;
329
330     /* Two commonly used timers in both sender and receiver paths. */
331     unsigned long    timeout;
332     struct timer_list    retransmit_timer;    /* Resend (no ack) */
333     struct timer_list    delack_timer;    /* Ack delay */
334
335     struct sk_buff_head    out_of_order_queue; /* Out of order segments go here
336         */
337
338     struct tcp_func
339         specific /*
340         struct sk_buff    *send_head;    /* Front of stuff to transmit
341         */
342     struct page    *sndmsg_page; /* Cached page for sendmsg
343     */
344     u32    sndmsg_off;    /* Cached offset for sendmsg
345     */
346
347     __u32    rcv_wnd;    /* Current receiver window */
348     __u32    rcv_wup;    /* rcv_nxt on last window update sent */
349     __u32    write_seq;    /* Tail(+1) of data held in tcp send buffer */
350     __u32    pushed_seq;    /* Last pushed seq, required to talk to windows */
351     __u32    copied_seq;    /* Head of yet unread data */
352
353     /*
354     * Options received (usually on last packet, some only on SYN packets).
355     */
356     char    tstamp_ok;    /* TIMESTAMP seen on SYN packet */
357     char    wscale_ok;    /* Wscale seen on SYN packet */
358     char    sack_ok;    /* SACK seen on SYN packet */
359     char    saw_tstamp;    /* Saw TIMESTAMP on last packet */
360     __u8    snd_wscale;    /* Window scaling received from sender */
361     __u8    rcv_wscale;    /* Window scaling to send to receiver */
362     __u8    nonagle;    /* Disable Nagle algorithm? */
363     __u8    keepalive_probes; /* num of allowed keep alive probes */
364
365     /* PAWS/RTT data */
366     __u32    rcv_tsval;    /* Time stamp value */
367     __u32    rcv_tsecr;    /* Time stamp echo reply */
368     __u32    ts_recent;    /* Time stamp to echo next */
369     long    ts_recent_stamp; /* Time we stored ts_recent (for aging) */
370
371     /* SACKs data */
372     __u16    user_mss;    /* mss requested by user in ioctl */
373     __u8    dsack;    /* D-SACK is scheduled */

```

tcp_opt

Figure 645: (iii)

```

368     __u8     eff_sacks ;      /* Size of SACK array to send with next packet */
369     struct tcp_sack_block duplicate_sack [1]; /* D-SACK block */
370     struct tcp_sack_block selective_acks [4]; /* The SACKS themselves*/
371
372     __u32    window_clamp; /* Maximal window to advertise */
373     __u32    rcv_ssthresh ; /* Current window clamp */
374     __u8     probes_out;    /* unanswered 0 window probes */
375     __u8     num_sacks;     /* Number of SACK blocks */
376     __u16    advmss;        /* Advertised MSS */
377
378     __u8     syn_retries ;  /* num of allowed syn retries */
379     __u8     ecn_flags ;    /* ECN status bits. */
380     __u16    prior_ssthresh ; /* ssthresh saved at recovery start */
381     __u32    lost_out ;    /* Lost packets */
382     __u32    sacked_out;   /* SACK'd packets */
383     __u32    fackets_out;  /* FACK'd packets */
384     __u32    high_seq;     /* snd_nxt at onset of congestion */
385
386     __u32    retrans_stamp; /* Timestamp of the last retransmit,
387                             * also used in SYN-SENT to remember stamp of
388                             * the first SYN. */
389     __u32    undo_marker;  /* tracking retrans started here. */
390     int      undo_retrans; /* number of undoable retransmissions. */
391     __u32    urg_seq;      /* Seq of received urgent pointer */
392     __u16    urg_data;     /* Saved octet of OOB data and control flags */
393     __u8     pending;      /* Scheduled timer event */
394     __u8     urg_mode;     /* In urgent mode */
395     __u32    snd_up;       /* Urgent pointer */
396
397     /* The syn_wait_lock is necessary only to avoid tcp_get_info having
398     * to grab the main lock sock while browsing the listening hash
399     * (otherwise it's deadlock prone).
400     * This lock is acquired in read mode only from tcp_get_info() and
401     * it's acquired in write mode _only_ from code that is actively
402     * changing the syn_wait_queue. All readers that are holding
403     * the master sock lock don't need to grab this lock in read mode
404     * too as the syn_wait_queue writes are always protected from
405     * the main sock lock.
406     */
407     rwlock_t syn_wait_lock;

```

tcp_opt

Figure 646: (iv)


```

408     struct tcp_listen_opt  *listen_opt ;
409
410     /* FIFO of established children */
411     struct open_request    *accept_queue;
412     struct open_request    *accept_queue_tail;
413
414     int                    write_pending; /* A write to socket waits to start
415     . */
416
417     unsigned int           keepalive_time; /* time before keep alive takes
418     place */
419     unsigned int           keepalive_intvl ; /* time interval between keep alive
420     probes */
421     int                    linger2;
422 };

```

tcp_opt
Figure 647: (v)

```

977 /* This is what the send packet queuing engine uses to pass
978 * TCP per-packet control information to the transmission
979 * code. We also store the host-order sequence numbers in
980 * here too. This is 36 bytes on 32-bit architectures,
981 * 40 bytes on 64-bit machines, if this grows please adjust
982 * skbuff.h:skbuff->cb[xxx] size appropriately.
983 */
984 struct tcp_skb_cb {
985     union {
986         struct inet_skb_parm  h4;
987 #if defined(CONFIG_IPV6) || defined (CONFIG_IPV6_MODULE)
988         struct inet6_skb_parm  h6;
989 #endif
990     } header;      /* For incoming frames      */
991     __u32          seq;          /* Starting sequence number */
992     __u32          end_seq;      /* SEQ + FIN + SYN + datalen */
993     __u32          when;        /* used to compute rtt's    */
994     __u8           flags;       /* TCP header flags.        */
995
996     /* NOTE: These must match up to the flags byte in a
997     *        real TCP header.
998     */
999 #define TCPCB_FLAG_FIN    0x01
1000 #define TCPCB_FLAG_SYN   0x02
1001 #define TCPCB_FLAG_RST   0x04
1002 #define TCPCB_FLAG_PSH   0x08
1003 #define TCPCB_FLAG_ACK   0x10
1004 #define TCPCB_FLAG_URG   0x20
1005 #define TCPCB_FLAG_ECE   0x40
1006 #define TCPCB_FLAG_CWR   0x80
1007
1008     __u8          sacked;      /* State flags for SACK/FAK. */
1009 #define TCPCB_SACKED_ACKED 0x01 /* SKB ACK'd by a SACK block */
1010 #define TCPCB_SACKED_RETRANS 0x02 /* SKB retransmitted */
1011 #define TCPCB_LOST        0x04 /* SKB is lost */
1012 #define TCPCB_TAGBITS     0x07 /* All tag bits */
1013
1014 #define TCPCB_EVER_RETRANS 0x80 /* Ever retransmitted frame */
1015 #define TCPCB_RETRANS      (TCPCB_SACKED_RETRANS|
1016     TCPCB_EVER_RETRANS)
1017
1018 #define TCPCB_URG          0x20 /* Urgent pointer advanced here */
1019
1020 #define TCPCB_AT_TAIL      (TCPCB_URG)
1021
1022     __u16         urg_ptr;     /* Valid w/URG flags is set. */
1023     __u32         ack_seq;     /* Sequence number ACK'd */
1024 };

```

tcp_skb_cb
 Figure 648: structure

```

1008 int tcp_sendmsg(struct sock *sk, struct msghdr *msg, int size)
1009 {
1010     struct iovec *iov;
1011     struct tcp_opt *tp;
1012     struct sk_buff *skb;
1013     int iovlen, flags;
1014     int mss_now;
1015     int err, copied;
1016     long timeo;
1017
1018     tp = &(sk->tp_pinfo.af_tcp);
1019
1020     lock_sock(sk);
1021     TCP_CHECK_TIMER(sk);
1022
1023     flags = msg->msg_flags;
1024     timeo = sock_sndtimeo(sk, flags&MSG_DONTWAIT);
1025
1026     /* Wait for a connection to finish. */
1027     if ((1 << sk->state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT))
1028         if ((err = wait_for_tcp_connect(sk, flags, &timeo)) != 0)
1029             goto out_err;
1030
1031     /* This should be in poll */
1032     clear_bit(SOCK_ASYNC_NOSPACE, &sk->socket->flags);
1033
1034     mss_now = tcp_current_mss(sk);
1035
1036     /* Ok commence sending. */
1037     iovlen = msg->msg_iovlen;
1038     iov = msg->msg_iov;
1039     copied = 0;
1040
1041     err = -EPIPE;
1042     if (sk->err || (sk->shutdown&SEND_SHUTDOWN))
1043         goto do_error;

```

tcp_sendmsg

Figure 649: Start of

```

892 /* Compute the current effective MSS, taking SACKs and IP options,
893 * and even PMTU discovery events into account.
894 */
895
896 static __inline__ unsigned int tcp_current_mss(struct sock *sk)
897 {
898     struct tcp_opt *tp = &sk->tp_pinfo.af_tcp;
899     struct dst_entry *dst = _sk_dst_get(sk);
900     int mss_now = tp->mss_cache;
901
902     if (dst && dst->pmtu != tp->pmtu_cookie)
903         mss_now = tcp_sync_mss(sk, dst->pmtu);
904
905     if (tp->eff_sacks)
906         mss_now -= (TCPOLEN_SACK_BASE_ALIGNED +
907                   (tp->eff_sacks * TCPOLEN_SACK_PERBLOCK));
908     return mss_now;
909 }

```

Figure 650: Determine MSS

```

1045     while (--iovlen >= 0) {
1046         int seglen=iov->iov_len;
1047         unsigned char * from=iov->iov_base;
1048
1049         iov++;
1050
1051         while (seglen > 0) {
1052             int copy;
1053
1054             skb = sk->write_queue.prev;
1055
1056             if (tp->send_head == NULL ||
1057                 (copy = mss_now - skb->len) <= 0) {
1058
1059 new_segment:
1060                 /* Allocate new segment. If the interface is SG,
1061                  * allocate skb fitting to single page.
1062                  */
1063                 if (!tcp_memory_free(skb))
1064                     goto wait_for_sndbuf;
1065
1066                 skb = tcp_alloc_skb(sk, select_size(sk, tp), 0, sk->
1067                     allocation);
1068                 if (skb == NULL)
1069                     goto wait_for_memory;
1070
1071                 skb_entail(sk, tp, skb);
1072                 copy = mss_now;
1073             }
1074
1075             /* Try to append data to the end of skb. */
1076             if (copy > seglen)
1077                 copy = seglen;

```

Figure 651: Start of send loop

```

1078     /* Where to copy to? */
1079     if (skb_tailroom(skb) > 0) {
1080         /* We have some space in skb head. Superb! */
1081         if (copy > skb_tailroom(skb))
1082             copy = skb_tailroom(skb);
1083         if ((err = skb_add_data(skb, from, copy)) != 0)
1084             goto do_fault;
1085     } else {
1086         int merge = 0;
1087         int i = skb_shinfo(skb)->nr_frags;
1088         struct page *page = TCP_PAGE(sk);
1089         int off = TCP_OFF(sk);
1090
1091         if (can_coalesce(skb, i, page, off) && off != PAGE_SIZE
1092             ) {
1093             /* We can extend the last page fragment. */
1094             merge = 1;
1095         } else if (i == MAX_SKB_FRAGS ||
1096                 (i == 0 && !(sk->route_caps&NETIF_F_SG))
1097                 ) {
1098             /* Need to add new fragment and cannot
1099              * do this because interface is non-SG,
1100              * or because all the page slots are busy.
1101              */
1102             tcp_mark_push(tp, skb);
1103             goto new_segment;
1104         } else if (page) {
1105             /* If page is cached, align
1106              * offset to L1 cache boundary
1107              */
1108             off = (off + L1_CACHE_BYTES - 1) & ~(
1109                 L1_CACHE_BYTES - 1);
1110             if (off == PAGE_SIZE) {
1111                 put_page(page);
1112                 TCP_PAGE(sk) = page = NULL;
1113             }
1114         }
1115     }
1116
1117     if (!page) {
1118         /* Allocate new cache page. */
1119         if (!(page = tcp_alloc_page(sk)))
1120             goto wait_for_memory;
1121         off = 0;

```

Figure 652: Finding space to copy the data to (i)

```

1118     }
1119
1120     if (copy > PAGE_SIZE-off)
1121         copy = PAGE_SIZE-off;
1122
1123     /* Time to copy data. We are close to the end! */
1124     err = tcp_copy_to_page(sk, from, skb, page, off, copy);
1125     if (err)
1126         goto do_error;
1127
1128     /* Update the skb. */
1129     if (merge) {
1130         skb_shinfo(skb)->frags[i-1].size += copy;
1131     } else {
1132         fill_page_desc(skb, i, page, off, copy);
1133         if (TCP_PAGE(sk)) {
1134             get_page(page);
1135         } else if (off + copy < PAGE_SIZE) {
1136             get_page(page);
1137             TCP_PAGE(sk) = page;
1138         }
1139     }
1140
1141     TCP_OFF(sk) = off+copy;
1142 }

```

Figure 653: Finding space to copy the data to (ii)

```

1144         if (!copied)
1145             TCP_SKB_CB(skb)->flags &= ~TCPCB_FLAG_PSH;
1146
1147         tp->write_seq += copy;
1148         TCP_SKB_CB(skb)->end_seq += copy;
1149
1150         from += copy;
1151         copied += copy;
1152         seglen -= copy;
1153
1154         if (skb->len != mss_now || (flags&MSG_OOB))
1155             continue;
1156
1157         if (forced_push(tp)) {
1158             tcp_mark_push(tp, skb);
1159             _tcp_push_pending_frames(sk, tp, mss_now, 1);
1160         } else if (skb == tp->send_head)
1161             tcp_push_one(sk, mss_now);
1162         continue;
1163
1164     wait_for_sndbuf:
1165         set_bit(SOCK_NOSPACE, &sk->socket->flags);
1166     wait_for_memory:
1167         if (copied)
1168             tcp_push(sk, tp, flags & ~MSG_MORE, mss_now, 1);
1169
1170         if ((err = wait_for_tcp_memory(sk, &timeo)) != 0)
1171             goto do_error;
1172
1173         mss_now = tcp_current_mss(sk);
1174     }
1175 }
1176
1177 out:
1178     if (copied)
1179         tcp_push(sk, tp, flags, mss_now, tp->nonagle);
1180     TCP_CHECK_TIMER(sk);
1181     release_sock(sk);
1182     return copied;

```

tcp_send

Figure 654: Rest of


```

1184 do_fault:
1185     if (skb->len==0) {
1186         if (tp->send_head == skb) {
1187             tp->send_head = skb->prev;
1188             if (tp->send_head == (struct sk_buff*)&sk->write_queue)
1189                 tp->send_head = NULL;
1190         }
1191         __skb_unlink(skb, skb->list);
1192         tcp_free_skb(sk, skb);
1193     }
1194
1195 do_error:
1196     if (copied)
1197         goto out;
1198 out_err:
1199     err = tcp_error(sk, flags, err);
1200     TCP_CHECK_TIMER(sk);
1201     release_sock(sk);
1202     return err;
1203 }

```

tcp_send

Figure 655: Rest of

```

538 /* This routine writes packets to the network. It advances the
539 * send_head. This happens as incoming acks open up the remote
540 * window for us.
541 *
542 * Returns 1, if no segments are in flight and we have queued segments, but
543 * cannot send anything now because of SWS or another problem.
544 */
545 int tcp_write_xmit(struct sock *sk, int nonagle)
546 {
547     struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
548     unsigned int mss_now;
549
550     /* If we are closed, the bytes will have to remain here.
551     * In time closedown will finish, we empty the write queue and all
552     * will be happy.
553     */
554     if (sk->state != TCP_CLOSE) {
555         struct sk_buff *skb;
556         int sent_pkts = 0;
557
558         /* Account for SACKS, we may need to fragment due to this.
559         * It is just like the real MSS changing on us midstream.
560         * We also handle things correctly when the user adds some
561         * IP options mid-stream. Silly to do, but cover it.
562         */
563         mss_now = tcp_current_mss(sk);
564
565         while((skb = tp->send_head) &&
566             tcp_snd_test(tp, skb, mss_now, tcp_skb_is_last(sk, skb) ? nonagle
567                 : 1)) {
568             if (skb->len > mss_now) {
569                 if (tcp_fragment(sk, skb, mss_now))
570                     break;
571             }
572
573             TCP_SKB_CB(skb)->when = tcp_time_stamp;
574             if (tcp_transmit_skb(sk, skb_clone(skb, GFP_ATOMIC)))
575                 break;
576
577             /* Advance the send_head. This one is sent out. */
578             update_send_head(sk, tp, skb);
579             tcp_minshall_update(tp, mss_now, skb);
580             sent_pkts = 1;
581         }
582
583         if (sent_pkts) {
584             tcp_cwnd_validate(sk, tp);
585             return 0;
586         }
587
588         return !tp->packets_out && tp->send_head;
589     }
590     return 0;
591 }

```

Figure 656: Writing socket buffers to the network

```

1181 /* This checks if the data bearing packet SKB (usually tp->send_head)
1182 * should be put on the wire right now.
1183 */
1184 static __inline__ int tcp_snd_test(struct tcp_opt *tp, struct sk_buff *skb,
1185                                 unsigned cur_mss, int nonagle)
1186 {
1187     /* RFC 1122 - section 4.2.3.4
1188     *
1189     * We must queue if
1190     *
1191     * a) The right edge of this frame exceeds the window
1192     * b) There are packets in flight and we have a small segment
1193     *    [SWS avoidance and Nagle algorithm]
1194     *    (part of SWS is done on packetization)
1195     *    Minshall version sounds: there are no _small_
1196     *    segments in flight. (tcp_nagle_check)
1197     * c) We have too many packets 'in flight'
1198     *
1199     * Don't use the nagle rule for urgent data (or
1200     * for the final FIN -DaveM).
1201     *
1202     * Also, Nagle rule does not apply to frames, which
1203     * sit in the middle of queue (they have no chances
1204     * to get new data) and if room at tail of skb is
1205     * not enough to save something seriously (<32 for now).
1206     */
1207
1208     /* Don't be strict about the congestion window for the
1209     * final FIN frame. -DaveM
1210     */
1211     return ((nonagle==1 || tp->urg_mode
1212             || !tcp_nagle_check(tp, skb, cur_mss, nonagle)) &&
1213            (( tcp_packets_in_flight (tp) < tp->snd_cwnd) ||
1214             (TCP_SKB_CB(skb)->flags & TCP_CB_FLAG_FIN)) &&
1215            !after(TCP_SKB_CB(skb)->end_seq, tp->snd_una + tp->snd_wnd));
1216 }

```

Figure 657: Should we send

```

177 /* This routine actually transmits TCP packets queued in by
178 * tcp_do_sendmsg(). This is used by both the initial
179 * transmission and possible later retransmissions.
180 * All SKB's seen here are completely headerless. It is our
181 * job to build the TCP header, and pass the packet down to
182 * IP so it can do the same plus pass the packet off to the
183 * device.
184 *
185 * We are working here with either a clone of the original
186 * SKB, or a fresh unique copy made by the retransmit engine.
187 */
188 int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb)
189 {
190     if (skb != NULL) {
191         struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
192         struct tcp_skb_cb *tcb = TCP_SKB_CB(skb);
193         int tcp_header_size = tp->tcp_header_len;
194         struct tcphdr *th;
195         int sysctl_flags;
196         int err;
197
198 #define SYSCTL_FLAG_TSTAMPS 0x1
199 #define SYSCTL_FLAG_WSCALE 0x2
200 #define SYSCTL_FLAG_SACK 0x4
201
202         sysctl_flags = 0;
203         if (tcb->flags & TCP_CB_FLAG_SYN) {
204             tcp_header_size = sizeof(struct tcphdr) + TCPOLEN_MSS;
205             if (sysctl_tcp_timestamps) {
206                 tcp_header_size += TCPOLEN_TSTAMP_ALIGNED;
207                 sysctl_flags |= SYSCTL_FLAG_TSTAMPS;
208             }
209             if (sysctl_tcp_window_scaling) {
210                 tcp_header_size += TCPOLEN_WSCALE_ALIGNED;
211                 sysctl_flags |= SYSCTL_FLAG_WSCALE;
212             }
213             if (sysctl_tcp_sack) {
214                 sysctl_flags |= SYSCTL_FLAG_SACK;
215                 if (!(sysctl_flags & SYSCTL_FLAG_TSTAMPS))
216                     tcp_header_size +=
217                         TCPOLEN_SACKPERM_ALIGNED;
218             }
219         } else if (tp->eff_sacks) {
220             /* A SACK is 2 pad bytes, a 2 byte header, plus
221              * 2 32-bit sequence numbers for each SACK block.
222              */
223             tcp_header_size += (TCPOLEN_SACK_BASE_ALIGNED +
224                               (tp->eff_sacks *
225                                TCPOLEN_SACK_PERBLOCK));
226         }
227         th = (struct tcphdr *) skb_push(skb, tcp_header_size);
228         skb->h.th = th;

```

Figure 658: Building a TCP header (i)

```

227     skb_set_owner_w(skb, sk);
228
229     /* Build TCP header and checksum it. */
230     th->source      = sk->sport;
231     th->dest        = sk->dport;
232     th->seq         = htonl(tcb->seq);
233     th->ack_seq     = htonl(tp->rcv_nxt);
234     *(((u16 *)th) + 6) = htons(((tcp_header_size >> 2) << 12) | tcb->
        flags);
235     if (tcb->flags & TCPCB_FLAG_SYN) {
236         /* RFC1323: The window in SYN & SYN/ACK segments
237          * is never scaled.
238          */
239         th->window  = htons(tp->rcv_wnd);
240     } else {
241         th->window  = htons(tcp_select_window(sk));
242     }
243     th->check       = 0;
244     th->urg_ptr     = 0;
245
246     if (tp->urg_mode &&
247         between(tp->snd_up, tcb->seq+1, tcb->seq+0xFFFF)) {
248         th->urg_ptr = htons(tp->snd_up-tcb->seq);
249         th->urg     = 1;
250     }
251
252     if (tcb->flags & TCPCB_FLAG_SYN) {
253         tcp_syn_build_options(((u32 *)th + 1),
254                               tcp_advertise_mss(sk),
255                               (sysctl_flags & SYSCTL_FLAG_TSTAMPS
256                                ),
257                               (sysctl_flags & SYSCTL_FLAG_SACK),
258                               (sysctl_flags & SYSCTL_FLAG_WSCALE),
259                               tp->rcv_wscale,
260                               tcb->when,
261                               tp->ts_recent);
262     } else {
263         tcp_build_and_update_options(((u32 *)th + 1),
264                                     tp, tcb->when);
265
266         TCP_ECN_send(sk, tp, skb, tcp_header_size);
267     }
268     tp->af_specific->send_check(sk, th, skb->len, skb);
269
270     if (tcb->flags & TCPCB_FLAG_ACK)
271         tcp_event_ack_sent(sk);
272
273     if (skb->len != tcp_header_size)
274         tcp_event_data_sent(tp, skb);
275
276     TCP_INC_STATS(TcpOutSegs);

```

Figure 659: Building a TCP header (ii)

```

277     err = tp->af_specific->queue_xmit(skb);
278     if (err <= 0)
279         return err;
280
281     tcp_enter_cwr(tp);
282
283     /* NET_XMIT_CN is special. It does not guarantee,
284      * that this packet is lost. It tells that device
285      * is about to start to drop packets or already
286      * drops some packets of the same priority and
287      * invokes us to send less aggressively.
288      */
289     return err == NET_XMIT_CN ? 0 : err;
290 }
291 return -ENOBUFS;
292 #undef SYSCTL_FLAG_TSTAMPS
293 #undef SYSCTL_FLAG_WSCALE
294 #undef SYSCTL_FLAG_SACK
295 }

```

Figure 660: Building a TCP header (ii)

```

1003 /* This routine computes an IPv4 TCP checksum. */
1004 void tcp_v4_send_check(struct sock *sk, struct tcphdr *th, int len,
1005                       struct sk_buff *skb)
1006 {
1007     if (skb->ip_summed == CHECKSUM_HW) {
1008         th->check = ~tcp_v4_check(th, len, sk->saddr, sk->daddr, 0);
1009         skb->csum = offsetof(struct tcphdr, check);
1010     } else {
1011         th->check = tcp_v4_check(th, len, sk->saddr, sk->daddr,
1012                                 csum_partial((char *)th, th->doff<<2, skb->
1013                                               csum));
1014     }
1015 }

```

Figure 661: Checksumming TCP packets

```

1592 int tcp_v4_rcv(struct sk_buff *skb)
1593 {
1594     struct tcphdr *th;
1595     struct sock *sk;
1596     int ret;
1597
1598     if (skb->pkt_type!=PACKET_HOST)
1599         goto discard_it ;
1600
1601     /* Count it even if it's bad */
1602     TCP_INC_STATS_BH(TcpInSegs);
1603
1604     if (!pskb_may_pull(skb, sizeof(struct tcphdr)))
1605         goto discard_it ;
1606
1607     th = skb->h.th;
1608
1609     if (th->doff < sizeof(struct tcphdr)/4)
1610         goto bad_packet;
1611     if (!pskb_may_pull(skb, th->doff*4))
1612         goto discard_it ;
1613
1614     /* An explanation is required here, I think.
1615     * Packet length and doff are validated by header prediction,
1616     * provided case of th->doff==0 is eliminated.
1617     * So, we defer the checks. */
1618     if ((skb->ip_summed != CHECKSUM_UNNECESSARY &&
1619         tcp_v4_checksum_init(skb) < 0))
1620         goto bad_packet;
1621
1622     th = skb->h.th;
1623     TCP_SKB_CB(skb)->seq = ntohl(th->seq);
1624     TCP_SKB_CB(skb)->end_seq = (TCP_SKB_CB(skb)->seq + th->syn + th->
1625         fin +
1626         skb->len - th->doff*4);
1627     TCP_SKB_CB(skb)->ack_seq = ntohl(th->ack_seq);
1628     TCP_SKB_CB(skb)->when = 0;
1629     TCP_SKB_CB(skb)->flags = skb->nh.iph->tos;
1630     TCP_SKB_CB(skb)->sacked = 0;
1631
1632     sk = __tcp_v4_lookup(skb->nh.iph->saddr, th->source,
1633         skb->nh.iph->daddr, ntohs(th->dest), tcp_v4_iif(skb));
1634
1635     if (!sk)
1636         goto no_tcp_socket ;
1637
1638     process:
1639     if (!ipsec_sk_policy(sk,skb))
1640         goto discard_and_release ;
1641
1642     if (sk->state == TCP_TIME_WAIT)
1643         goto do_time_wait;
1644
1645     skb->dev = NULL;
1646     bh_lock_sock(sk);

```

Figure 662: Receiving a packet by the TCP layer (i)


```

1647     ret = 0;
1648     if (!sk->lock.users) {
1649         if (!tcp_prequeue(sk, skb))
1650             ret = tcp_v4_do_rcv(sk, skb);
1651     } else
1652         sk_add_backlog(sk, skb);
1653     bh_unlock_sock(sk);
1654
1655     sock_put(sk);
1656
1657     return ret;
1658
1659 no_tcp_socket:
1660     if (skb->len < (th->doff<<2) || tcp_checksum_complete(skb)) {
1661 bad_packet:
1662         TCP_INC_STATS_BH(TcpInErrs);
1663     } else {
1664         tcp_v4_send_reset(skb);
1665     }
1666
1667 discard_it:
1668     /* Discard frame. */
1669     kfree_skb(skb);
1670     return 0;
1671
1672 discard_and_release:
1673     sock_put(sk);
1674     goto discard_it;
1675
1676 do_time_wait:
1677     if (skb->len < (th->doff<<2) || tcp_checksum_complete(skb)) {
1678         TCP_INC_STATS_BH(TcpInErrs);
1679         goto discard_and_release;
1680     }
1681     switch(tcp_timewait_state_process((struct tcp_tw_bucket *)sk,
1682                                     skb, th, skb->len)) {
1683     case TCP_TW_SYN:
1684     {
1685         struct sock *sk2;
1686
1687         sk2 = tcp_v4_lookup_listener(skb->nh.iph->daddr, ntohs(th->dest),
1688                                   tcp_v4_iif(skb));
1689         if (sk2 != NULL) {
1690             tcp_tw_deschedule((struct tcp_tw_bucket *)sk);
1691             tcp_timewait_kill((struct tcp_tw_bucket *)sk);
1692             tcp_tw_put((struct tcp_tw_bucket *)sk);
1693             sk = sk2;
1694             goto process;
1695         }
1696         /* Fall through to ACK */
1697     }
1698     case TCP_TW_ACK:
1699         tcp_v4_timewait_ack(sk, skb);
1700         break;
1701     case TCP_TW_RST:
1702         goto no_tcp_socket;
1703     case TCP_TW_SUCCESS:;
1704     }
1705     goto discard_it;

```

Figure 663: Receiving a packet by the TCP layer (ii)

```

1309 /* Packet is added to VJ-style prequeue for processing in process
1310 * context, if a reader task is waiting. Apparently, this exciting
1311 * idea (VJ's mail "Re: query about TCP header on tcp-ip" of 07 Sep 93)
1312 * failed somewhere. Latency? Burstiness? Well, at least now we will
1313 * see, why it failed. 8)8) --ANK
1314 *
1315 * NOTE: is this not too big to inline?
1316 */
1317 static __inline__ int tcp_prequeue(struct sock *sk, struct sk_buff *skb)
1318 {
1319     struct tcp_opt *tp = &sk->tp_pinfo.af_tcp;
1320
1321     if (tp->ucopy.task) {
1322         __skb_queue_tail(&tp->ucopy.prequeue, skb);
1323         tp->ucopy.memory += skb->truesize;
1324         if (tp->ucopy.memory > sk->rcvbuf) {
1325             struct sk_buff *skb1;
1326
1327             if (sk->lock.users) BUG();
1328
1329             while ((skb1 = __skb_dequeue(&tp->ucopy.prequeue)) != NULL)
1330                 {
1331                     sk->backlog_rcv(sk, skb1);
1332                     NET_INC_STATS_BH(TCPPrequeueDropped);
1333                 }
1334             tp->ucopy.memory = 0;
1335         } else if (skb_queue_len(&tp->ucopy.prequeue) == 1) {
1336             wake_up_interruptible(sk->sleep);
1337             if (!tcp_ack_scheduled(tp))
1338                 tcp_reset_xmit_timer(sk, TCP_TIME_DACK, (3*
1339                                     TCP_RTO_MIN)/4);
1340         }
1341         return 1;
1342     }
1343     return 0;
1344 }

```

Figure 664: An optimisation

```

3200 /*
3201 *      TCP receive function for the ESTABLISHED state.
3202 *
3203 *      It is split into a fast path and a slow path. The fast path is
3204 *      disabled when:
3205 *      - A zero window was announced from us - zero window probing
3206 *        is only handled properly in the slow path.
3207 *      - Out of order segments arrived.
3208 *      - Urgent data is expected.
3209 *      - There is no buffer space left
3210 *      - Unexpected TCP flags/window values/header lengths are received
3211 *        (detected by checking the TCP header against pred_flags)
3212 *      - Data is sent in both directions. Fast path only supports pure senders
3213 *        or pure receivers (this means either the sequence number or the ack
3214 *        value must stay constant)
3215 *      - Unexpected TCP option.
3216 *
3217 *      When these conditions are not satisfied it drops into a standard
3218 *      receive procedure patterned after RFC793 to handle all cases.
3219 *      The first three cases are guaranteed by proper pred_flags setting,
3220 *      the rest is checked inline. Fast processing is turned on in
3221 *      tcp_data_queue when everything is OK.
3222 */
3223 int tcp_rcv_established(struct sock *sk, struct sk_buff *skb,
3224                       struct tcphdr *th, unsigned len)
3225 {
3226     struct tcp_opt *tp = &(sk->tp_info.af_tcp);
3227
3228     /*
3229      *      Header prediction.
3230      *      The code loosely follows the one in the famous
3231      *      "30 instruction TCP receive" Van Jacobson mail.
3232      *
3233      *      Van's trick is to deposit buffers into socket queue
3234      *      on a device interrupt, to call tcp_rcv function
3235      *      on the receive process context and checksum and copy
3236      *      the buffer to user space. smart...
3237      *
3238      *      Our current scheme is not silly either but we take the
3239      *      extra cost of the net_bh soft interrupt processing...
3240      *      We do checksum and copy also but from device to kernel.
3241      */
3242
3243     tp->saw_tstamp = 0;
3244
3245     /*      pred_flags is 0xS?10 << 16 + snd_wnd
3246      *      if header_prediction is to be made
3247      *      'S' will always be tp->tcp_header_len >> 2
3248      *      '?' will be 0 for the fast path, otherwise pred_flags is 0 to
3249      *      turn it off (when there are holes in the receive

```

Figure 665: Receiving a socket buffer on an established connection (i)

```

3250     *   space for instance)
3251     *   PSH flag is ignored.
3252     */
3253
3254     if ((tcp_flag_word(th) & TCP_HP_BITS) == tp->pred_flags &&
3255         TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
3256         int tcp_header_len = tp->tcp_header_len;
3257
3258         /* Timestamp header prediction: tcp_header_len
3259          * is automatically equal to th->doff*4 due to pred_flags
3260          * match.
3261          */
3262
3263         /* Check timestamp */
3264         if (tcp_header_len == sizeof(struct tcphdr) +
3265             TCPOLEN_TSTAMP_ALIGNED) {
3266             __u32 *ptr = (__u32 *) (th + 1);
3267
3268             /* No? Slow path! */
3269             if (*ptr != __constant_ntohl((TCPOPT_NOP << 24) | (
3270                 TCPOPT_NOP << 16)
3271                 | (TCPOPT_TIMESTAMP << 8) |
3272                 TCPOLEN_TIMESTAMP))
3273                 goto slow_path;
3274
3275             tp->saw_tstamp = 1;
3276             ++ptr;
3277             tp->rcv_tsval = ntohl(*ptr);
3278             ++ptr;
3279             tp->rcv_tsecr = ntohl(*ptr);
3280
3281             /* If PAWS failed, check it more carefully in slow path */
3282             if ((s32)(tp->rcv_tsval - tp->ts_recent) < 0)
3283                 goto slow_path;
3284
3285             /* Predicted packet is in window by definition.
3286              * seq == rcv_nxt and rcv_wup <= rcv_nxt.
3287              * Hence, check seq<=rcv_wup reduces to:
3288              */
3289             if (tp->rcv_nxt == tp->rcv_wup)
3290                 tcp_store_ts_recent(tp);
3291         }
3292     }
3293
3294     if (len <= tcp_header_len) {
3295         /* Bulk data transfer: sender */
3296         if (len == tcp_header_len) {
3297             /* We know that such packets are checksummed
3298              * on entry.
3299              */
3300             tcp_ack(sk, skb, 0);
3301             __kfree_skb(skb);
3302             tcp_data_snd_check(sk);
3303             return 0;
3304         }
3305     }

```

Figure 666: Receiving a socket buffer on an established connection (ii)

```

3300         } else { /* Header too small */
3301             TCP_INC_STATS_BH(TcpInErrs);
3302             goto discard;
3303         }
3304     } else {
3305         int eaten = 0;
3306
3307         if (tp->ucopy.task == current &&
3308             tp->copied_seq == tp->rcv_nxt &&
3309             len - tcp_header_len <= tp->ucopy.len &&
3310             sk->lock.users) {
3311             eaten = 1;
3312
3313             NET_INC_STATS_BH(TCPHPHitsToUser);
3314
3315             __set_current_state (TASK_RUNNING);
3316
3317             if (tcp_copy_to_iovec(sk, skb, tcp_header_len))
3318                 goto csum_error;
3319
3320             __skb_pull(skb, tcp_header_len);
3321
3322             tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
3323         } else {
3324             if (tcp_checksum_complete_user(sk, skb))
3325                 goto csum_error;
3326
3327             if ((int)skb->truesize > sk->forward_alloc)
3328                 goto step5;
3329
3330             NET_INC_STATS_BH(TCPHPHits);
3331
3332             /* Bulk data transfer: receiver */
3333             __skb_pull(skb, tcp_header_len);
3334             __skb_queue_tail(&sk->receive_queue, skb);
3335             tcp_set_owner_r(skb, sk);
3336             tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
3337         }
3338
3339         tcp_event_data_rcv(sk, tp, skb);
3340
3341         if (TCP_SKB_CB(skb)->ack_seq != tp->snd_una) {
3342             /* Well, only one small jumplet in fast path... */
3343             tcp_ack(sk, skb, FLAG_DATA);
3344             tcp_data_snd_check(sk);
3345             if (!tcp_ack_scheduled(tp))
3346                 goto no_ack;
3347         }
3348
3349         if (eaten) {

```

Figure 667: Receiving a socket buffer on an established connection (iii)

```

3350         if (tcp_in_quickack_mode(tp)) {
3351             tcp_send_ack(sk);
3352         } else {
3353             tcp_send_delayed_ack(sk);
3354         }
3355     } else {
3356         _tcp_ack_snd_check(sk, 0);
3357     }
3358
3359 no_ack:
3360     if (eaten)
3361         _kfree_skb(skb);
3362     else
3363         sk->data_ready(sk, 0);
3364     return 0;
3365 }
3366
3367 slow_path:
3368     if (len < (th->doff<<2) || tcp_checksum_complete_user(sk, skb))
3369         goto csum_error;
3370
3371     /*
3372     * RFC1323: H1. Apply PAWS check first.
3373     */
3374     if (tcp_fast_parse_options(skb, th, tp) && tp->saw_tstamp &&
3375         tcp_paws_discard(tp, skb)) {
3376         if (!th->rst) {
3377             NET_INC_STATS_BH(PAWS_Estab_Rejected);
3378             tcp_send_dupack(sk, skb);
3379             goto discard;
3380         }
3381         /* Resets are accepted even if PAWS failed.
3382
3383         ts_recent update must be made after we are sure
3384         that the packet is in window.
3385         */
3386     }
3387
3388     /*
3389     * Standard slow path.
3390     */
3391     if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq))
3392     {
3393         /* RFC793, page 37: "In all states except SYN-SENT, all reset
3394         * (RST) segments are validated by checking their SEQ-fields."
3395         * And page 69: "If an incoming segment is not acceptable,
3396         * an acknowledgment should be sent in reply (unless the RST bit
3397         * is set, if so drop the segment and return)".
3398         */
3399     }

```

Figure 668: Receiving a socket buffer on an established connection (iv)

```

3400         if (!th->rst)
3401             tcp_send_dupack(sk, skb);
3402         goto discard;
3403     }
3404
3405     if (th->rst) {
3406         tcp_reset(sk);
3407         goto discard;
3408     }
3409
3410     tcp_replace_ts_recent(tp, TCP_SKB_CB(skb)->seq);
3411
3412     if (th->syn && !before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
3413         TCP_INC_STATS_BH(TcpInErrs);
3414         NET_INC_STATS_BH(TCPAbortOnSyn);
3415         tcp_reset(sk);
3416         return 1;
3417     }
3418
3419 step5:
3420     if (th->ack)
3421         tcp_ack(sk, skb, FLAG_SLOWPATH);
3422
3423     /* Process urgent data. */
3424     tcp_urg(sk, skb, th);
3425
3426     /* step 7: process the segment text */
3427     tcp_data_queue(sk, skb);
3428
3429     tcp_data_snd_check(sk);
3430     tcp_ack_snd_check(sk);
3431     return 0;
3432
3433 csum_error:
3434     TCP_INC_STATS_BH(TcpInErrs);
3435
3436 discard:
3437     __kfree_skb(skb);
3438     return 0;
3439 }

```

Figure 669: Receiving a socket buffer on an established connection (iv)

```

2175 /* Check segment sequence number for validity.
2176 *
2177 * Segment controls are considered valid, if the segment
2178 * fits to the window after truncation to the window. Acceptability
2179 * of data (and SYN, FIN, of course) is checked separately.
2180 * See tcp_data_queue(), for example.
2181 *
2182 * Also, controls (RST is main one) are accepted using RCV.WUP instead
2183 * of RCV.NXT. Peer still did not advance his SND.UNA when we
2184 * delayed ACK, so that hisSND.UNA<=ourRCV.WUP.
2185 * (borrowed from freebsd)
2186 */
2187
2188 static inline int tcp_sequence(struct tcp_opt *tp, u32 seq, u32 end_seq)
2189 {
2190     return !before(end_seq, tp->rcv_wup) &&
2191           !after(seq, tp->rcv_nxt + tcp_receive_window(tp));
2192 }

```

Figure 670: TCP sequence number check


```
950 /* Compute the actual receive window we are currently advertising.
951 * Rcv_nxt can be after the window if our peer push more data
952 * than the offered window.
953 */
954 static __inline__ u32 tcp_receive_window(struct tcp_opt *tp)
955 {
956     s32 win = tp->rcv_wup + tp->rcv_wnd - tp->rcv_nxt;
957
958     if (win < 0)
959         win = 0;
960     return (u32) win;
961 }
```

Figure 671: TCP receive window determination

```

3006 /*
3007  * Check if sending an ack is needed.
3008  */
3009 static __inline__ void __tcp_ack_snd_check(struct sock *sk, int ofo_possible)
3010 {
3011     struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
3012
3013     /* More than one full frame received... */
3014     if (((tp->rcv_nxt - tp->rcv_wup) > tp->ack.rcv_mss
3015         /* ... and right edge of window advances far enough.
3016          * (tcp_recvmmsg() will send ACK otherwise). Or...
3017          */
3018         && __tcp_select_window(sk) >= tp->rcv_wnd) ||
3019         /* We ACK each frame or... */
3020         tcp_in_quickack_mode(tp) ||
3021         /* We have out of order data. */
3022         (ofo_possible &&
3023          skb_peek(&tp->out_of_order_queue) != NULL)) {
3024         /* Then ack it now */
3025         tcp_send_ack(sk);
3026     } else {
3027         /* Else, send delayed ack. */
3028         tcp_send_delayed_ack(sk);
3029     }
3030 }
3031
3032 static __inline__ void tcp_ack_snd_check(struct sock *sk)
3033 {
3034     struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
3035     if (!tcp_ack_scheduled(tp)) {
3036         /* We sent a data segment already. */
3037         return;
3038     }
3039     __tcp_ack_snd_check(sk, 1);
3040 }

```

Figure 672: Ack sending code

```

1897 /* This routine deals with incoming acks, but not outgoing ones. */
1898 static int tcp_ack(struct sock *sk, struct sk_buff *skb, int flag)
1899 {
1900     struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
1901     u32 prior_snd_una = tp->snd_una;
1902     u32 ack_seq = TCP_SKB_CB(skb)->seq;
1903     u32 ack = TCP_SKB_CB(skb)->ack_seq;
1904     u32 prior_in_flight ;
1905     int prior_packets;
1906
1907     /* If the ack is newer than sent or older than previous acks
1908      * then we can probably ignore it.
1909      */
1910     if (after(ack, tp->snd_nxt))
1911         goto uninteresting_ack;
1912
1913     if (before(ack, prior_snd_una))
1914         goto old_ack;
1915
1916     if (!(flag & FLAG_SLOWPATH) && after(ack, prior_snd_una)) {
1917         /* Window is constant, pure forward advance.
1918          * No more checks are required.
1919          * Note, we use the fact that SND.UNA>=SND.WL2.
1920          */
1921         tcp_update_wl(tp, ack, ack_seq);
1922         tp->snd_una = ack;
1923         flag |= FLAG_WIN_UPDATE;
1924
1925         NET_INC_STATS_BH(TCPHPAcks);
1926     } else {
1927         if (ack_seq != TCP_SKB_CB(skb)->end_seq)
1928             flag |= FLAG_DATA;
1929         else
1930             NET_INC_STATS_BH(TCPPureAcks);
1931
1932         flag |= tcp_ack_update_window(sk, tp, skb, ack, ack_seq);
1933
1934         if (TCP_SKB_CB(skb)->sacked)
1935             flag |= tcp_sacktag_write_queue(sk, skb, prior_snd_una);
1936
1937         if (TCP_ECN_rcv_ecn_echo(tp, skb->h.th))
1938             flag |= FLAG_ECE;
1939     }
1940
1941     /* We passed data and got it acked, remove any soft error
1942      * log. Something worked...
1943      */
1944     sk->err_soft = 0;
1945     tp->rcv_tstamp = tcp_time_stamp;
1946     if ((prior_packets = tp->packets_out) == 0)
1947         goto no_queue;

```

Figure 673: Processing an acknowledgement (i)

```

1848         !((1<<tp->ca_state)&(TCPF_CA_Recovery|TCPF_CA_CWR));
1849     }
1850
1851     /* Check that window update is acceptable.
1852     * The function assumes that snd_una<=ack<=snd_next.
1853     */
1854     static __inline__ int
1855     tcp_may_update_window(struct tcp_opt *tp, u32 ack, u32 ack_seq, u32 nwin)
1856     {
1857         return (after(ack, tp->snd_una) ||
1858             after(ack_seq, tp->snd_wl1) ||
1859             (ack_seq == tp->snd_wl1 && nwin > tp->snd_wnd));
1860     }
1861
1862     /* Update our send window.
1863     *
1864     * Window update algorithm, described in RFC793/RFC1122 (used in linux-2.2
1865     * and in FreeBSD. NetBSD's one is even worse.) is wrong.
1866     */
1867     static int tcp_ack_update_window(struct sock *sk, struct tcp_opt *tp,
1868         struct sk_buff *skb, u32 ack, u32 ack_seq)
1869     {
1870         int flag = 0;
1871         u32 nwin = ntohs(skb->h.th->window) << tp->snd_wsacle;
1872
1873         if (tcp_may_update_window(tp, ack, ack_seq, nwin)) {
1874             flag |= FLAG_WIN_UPDATE;
1875             tcp_update_wl(tp, ack, ack_seq);
1876
1877             if (tp->snd_wnd != nwin) {
1878                 tp->snd_wnd = nwin;
1879
1880                 /* Note, it is the only place, where
1881                 * fast path is recovered for sending TCP.
1882                 */
1883                 tcp_fast_path_check(sk, tp);
1884
1885                 if (nwin > tp->max_window) {
1886                     tp->max_window = nwin;
1887                     tcp_sync_mss(sk, tp->pmtu_cookie);
1888                 }
1889             }
1890         }
1891
1892         tp->snd_una = ack;
1893
1894         return flag;
1895     }
1896
1897     /* This routine deals with incoming acks, but not outgoing ones. */
1898     static int tcp_ack(struct sock *sk, struct sk_buff *skb, int flag)
1899     {
1900         struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
1901         u32 prior_snd_una = tp->snd_una;
1902         u32 ack_seq = TCP_SKB_CB(skb)->seq;
1903         u32 ack = TCP_SKB_CB(skb)->ack_seq;
1904         u32 prior_in_flight;
1905         int prior_packets;
1906
1907         /* If the ack is newer than sent or older than previous acks
1908         * then we can probably ignore it.
1909         */
1910         if (after(ack, tp->snd_nxt))
1911             goto uninteresting_ack;
1912
1913         if (before(ack, prior_snd_una))
1914             goto old_ack;
1915
1916         if (!(flag & FLAG_SLOWPATH) && after(ack, prior_snd_una)) {
1917             /* Window is constant, pure forward advance.
1918             * No more checks are required.
1919             * Note, we use the fact that SND.UNA>=SND.WL2.
1920             */
1921             tcp_update_wl(tp, ack, ack_seq);
1922             tp->snd_una = ack;

```

```

2524 static void tcp_data_queue(struct sock *sk, struct sk_buff *skb)
2525 {
2526     struct tcphdr *th = skb->h.th;
2527     struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
2528     int eaten = -1;
2529
2530     th = skb->h.th;
2531     __skb_pull(skb, th->doff*4);
2532
2533     if (skb->len == 0 && !th->fin)
2534         goto drop;
2535
2536     TCP_ECN_accept_cwr(tp, skb);
2537
2538     if (tp->dsack) {
2539         tp->dsack = 0;
2540         tp->eff_sacks = min(tp->num_sacks, 4-tp->tstamp_ok);
2541     }
2542
2543     /* Queue data for delivery to the user.
2544     * Packets in sequence go to the receive queue.
2545     * Out of sequence packets to the out_of_order_queue.
2546     */
2547     if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
2548         if (tcp_receive_window(tp) == 0)
2549             goto out_of_window;
2550
2551         /* Ok. In sequence. In window. */
2552         if (tp->ucopy.task == current &&
2553             tp->copied_seq == tp->rcv_nxt &&
2554             tp->ucopy.len &&
2555             sk->lock.users &&
2556             !tp->urg_data) {
2557             int chunk = min(skb->len, tp->ucopy.len);
2558
2559             __set_current_state (TASK_RUNNING);
2560
2561             local_bh_enable ();
2562             if (skb_copy_datagram_iovec(skb, 0, tp->ucopy.iov, chunk)) {
2563                 sk->err = EFAULT;
2564                 sk->error_report(sk);
2565             }
2566             local_bh_disable ();
2567             tp->ucopy.len -= chunk;
2568             tp->copied_seq += chunk;
2569             eaten = (chunk == skb->len && !th->fin);
2570         }
2571
2572         if (eaten <= 0) {
2573         queue_and_out:
2574             if (eaten < 0 &&
2575                 (atomic_read(&sk->rmem_alloc) > sk->rcvbuf ||
2576                  !tcp_rmem_schedule(sk, skb))) {
2577                 if (tcp_prune_queue(sk) < 0 || !tcp_rmem_schedule(sk, skb
2578                     ))
2579                     goto drop;

```

Figure 675: processing and queuing incoming data (i)

```

2579         }
2580         tcp_set_owner_r(skb, sk);
2581         __skb_queue_tail(&sk->receive_queue, skb);
2582     }
2583     tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
2584     if (skb->len)
2585         tcp_event_data_rcv(sk, tp, skb);
2586     if (th->fin)
2587         tcp_fin(skb, sk, th);
2588
2589     if (skb_queue_len(&tp->out_of_order_queue)) {
2590         tcp_ofo_queue(sk);
2591
2592         /* RFC2581. 4.2. SHOULD send immediate ACK, when
2593          * gap in queue is filled.
2594          */
2595         if (skb_queue_len(&tp->out_of_order_queue) == 0)
2596             tp->ack.pingpong = 0;
2597     }
2598
2599     if (tp->num_sacks)
2600         tcp_sack_remove(tp);
2601
2602     tcp_fast_path_check(sk, tp);
2603
2604     if (eaten > 0) {
2605         __kfree_skb(skb);
2606     } else if (!sk->dead)
2607         sk->data_ready(sk, 0);
2608     return;
2609 }
2610
2611 if (!after(TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt)) {
2612     /* A retransmit, 2nd most common case. Force an immediate ack. */
2613     NET_INC_STATS_BH(DelayedACKLost);
2614     tcp_enter_quickack_mode(tp);
2615     tcp_dsack_set(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->
2616                 end_seq);
2617
2618 out_of_window:
2619     tcp_schedule_ack(tp);
2620
2621 drop:
2622     __kfree_skb(skb);
2623     return;
2624 }
2625
2626 /* Out of window. F.e. zero window probe. */
2627 if (!before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt+tcp_receive_window(tp)))
2628     goto out_of_window;
2629
2630 tcp_enter_quickack_mode(tp);
2631
2632 if (before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
2633     /* Partial packet, seq < rcv_next < end_seq */
2634     SOCK_DEBUG(sk, "partial_packet:rcv_next_%Xseq_%X-_%X\n",
2635               tp->rcv_nxt, TCP_SKB_CB(skb)->seq,

```

Figure 676: processing and queuing incoming data (ii)

```

2634         TCP_SKB_CB(skb)->end_seq);
2635
2636         tcp_dsack_set(tp, TCP_SKB_CB(skb)->seq, tp->rcv_nxt);
2637
2638         /* If window is closed, drop tail of packet. But after
2639          * remembering D-SACK for its head made in previous line.
2640          */
2641         if (!tcp_receive_window(tp))
2642             goto out_of_window;
2643         goto queue_and_out;
2644     }
2645
2646     TCP_ECN_check_ce(tp, skb);
2647
2648     if (atomic_read(&sk->rmem_alloc) > sk->rcvbuf ||
2649         !tcp_rmem_schedule(sk, skb)) {
2650         if (tcp_prune_queue(sk) < 0 || !tcp_rmem_schedule(sk, skb))
2651             goto drop;
2652     }
2653
2654     /* Disable header prediction. */
2655     tp->pred_flags = 0;
2656     tcp_schedule_ack(tp);
2657
2658     SOCK_DEBUG(sk, "out_of_order_segment:rcv_next_%Xseq_%X-_%X\n",
2659               tp->rcv_nxt, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq
2660               );
2661
2662     tcp_set_owner_r(skb, sk);
2663
2664     if (skb_peek(&tp->out_of_order_queue) == NULL) {
2665         /* Initial out of order segment, build 1 SACK. */
2666         if (tp->sack_ok) {
2667             tp->num_sacks = 1;
2668             tp->dsack = 0;
2669             tp->eff_sacks = 1;
2670             tp->selective_acks[0].start_seq = TCP_SKB_CB(skb)->seq;
2671             tp->selective_acks[0].end_seq = TCP_SKB_CB(skb)->end_seq;
2672         }
2673         __skb_queue_head(&tp->out_of_order_queue, skb);
2674     } else {
2675         struct sk_buff *skb1 = tp->out_of_order_queue.prev;
2676         u32 seq = TCP_SKB_CB(skb)->seq;
2677         u32 end_seq = TCP_SKB_CB(skb)->end_seq;
2678
2679         if (seq == TCP_SKB_CB(skb1)->end_seq) {
2680             __skb_append(skb1, skb);
2681
2682             if (tp->num_sacks == 0 ||
2683                 tp->selective_acks[0].end_seq != seq)
2684                 goto add_sack;
2685
2686             /* Common case: data arrive in order after hole. */
2687             tp->selective_acks[0].end_seq = end_seq;
2688             return;
2689         }
2690     }

```

Figure 677: processing and queuing incoming data (iii)

```

2689
2690      /* Find place to insert this segment. */
2691      do {
2692          if (! after(TCP_SKB_CB(skb1)->seq, seq))
2693              break;
2694      } while ((skb1=skb1->prev) != (struct sk_buff*)&tp->out_of_order_queue
                );
2695
2696      /* Do skb overlap to previous one? */
2697      if (skb1 != (struct sk_buff*)&tp->out_of_order_queue &&
          before(seq, TCP_SKB_CB(skb1)->end_seq)) {
2698          if (! after(end_seq, TCP_SKB_CB(skb1)->end_seq)) {
2699              /* All the bits are present. Drop. */
2700              _kfree_skb(skb);
2701              tcp_dsack_set(tp, seq, end_seq);
2702              goto add_sack;
2703          }
2704          if (after(seq, TCP_SKB_CB(skb1)->seq)) {
2705              /* Partial overlap. */
2706              tcp_dsack_set(tp, seq, TCP_SKB_CB(skb1)->end_seq);
2707          } else {
2708              skb1 = skb1->prev;
2709          }
2710      }
2711      _skb_insert(skb, skb1, skb1->next, &tp->out_of_order_queue);
2712
2713
2714      /* And clean segments covered by new one as whole. */
2715      while ((skb1 = skb->next) != (struct sk_buff*)&tp->out_of_order_queue
              &&
              after(end_seq, TCP_SKB_CB(skb1)->seq)) {
2716          if (before(end_seq, TCP_SKB_CB(skb1)->end_seq)) {
2717              tcp_dsack_extend(tp, TCP_SKB_CB(skb1)->seq, end_seq);
2718              break;
2719          }
2720          _skb_unlink(skb1, skb1->list);
2721          tcp_dsack_extend(tp, TCP_SKB_CB(skb1)->seq, TCP_SKB_CB(
2722              skb1)->end_seq);
2723          _kfree_skb(skb1);
2724      }
2725
2726      add_sack:
2727          if (tp->sack_ok)
2728              tcp_sack_new_ofo_skb(sk, seq, end_seq);
2729      }
2730

```

Figure 678: processing and queuing incoming data (iv)


```

40  /*
41  * Using different timers for retransmit, delayed acks and probes
42  * We may wish use just one timer maintaining a list of expire jiffies
43  * to optimize.
44  */
45
46 void tcp_init_xmit_timers(struct sock *sk)
47 {
48     struct tcp_opt *tp = &sk->tp_pinfo.af_tcp;
49
50     init_timer(&tp->retransmit_timer);
51     tp->retransmit_timer.function=&tcp_write_timer;
52     tp->retransmit_timer.data = (unsigned long) sk;
53     tp->pending = 0;
54
55     init_timer(&tp->delack_timer);
56     tp->delack_timer.function=&tcp_delack_timer;
57     tp->delack_timer.data = (unsigned long) sk;
58     tp->ack.pending = 0;
59
60     init_timer(&sk->timer);
61     sk->timer.function=&tcp_keepalive_timer;
62     sk->timer.data = (unsigned long) sk;
63 }

```

Figure 679: Timer Initialisation

```

1699  /* This is Jacobson's slow start and congestion avoidance.
1700  * SIGCOMM '88, p. 328.
1701  */
1702  static __inline__ void tcp_cong_avoid(struct tcp_opt *tp)
1703  {
1704      if (tp->snd_cwnd <= tp->snd_ssthresh) {
1705          /* In "safe" area, increase. */
1706          if (tp->snd_cwnd < tp->snd_cwnd_clamp)
1707              tp->snd_cwnd++;
1708      } else {
1709          /* In dangerous area, increase slowly.
1710          * In theory this is tp->snd_cwnd += 1 / tp->snd_cwnd
1711          */
1712          if (tp->snd_cwnd_cnt >= tp->snd_cwnd) {
1713              if (tp->snd_cwnd < tp->snd_cwnd_clamp)
1714                  tp->snd_cwnd++;
1715              tp->snd_cwnd_cnt=0;
1716          } else
1717              tp->snd_cwnd_cnt++;
1718      }
1719      tp->snd_cwnd_stamp = tcp_time_stamp;
1720  }

```

Figure 680: TCP congestion avoidance

```

2397 static void tcp_sack_new_ofo_skb(struct sock *sk, u32 seq, u32 end_seq)
2398 {
2399     struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
2400     struct tcp_sack_block *sp = &tp->selective_acks[0];
2401     int cur_sacks = tp->num_sacks;
2402     int this_sack;
2403
2404     if (!cur_sacks)
2405         goto new_sack;
2406
2407     for (this_sack=0; this_sack<cur_sacks; this_sack++, sp++) {
2408         if (tcp_sack_extend(sp, seq, end_seq)) {
2409             /* Rotate this_sack to the first one. */
2410             for (; this_sack>0; this_sack--, sp--)
2411                 tcp_sack_swap(sp, sp-1);
2412             if (cur_sacks > 1)
2413                 tcp_sack_maybe_coalesce(tp);
2414             return;
2415         }
2416     }
2417
2418     /* Could not find an adjacent existing SACK, build a new one,
2419      * put it at the front, and shift everyone else down. We
2420      * always know there is at least one SACK present already here.
2421      *
2422      * If the sack array is full, forget about the last one.
2423      */
2424     if (this_sack >= 4) {
2425         this_sack--;
2426         tp->num_sacks--;
2427         sp--;
2428     }
2429     for (; this_sack > 0; this_sack--, sp--)
2430         *sp = *(sp-1);
2431
2432 new_sack:
2433     /* Build the new head SACK, and we're done. */
2434     sp->start_seq = seq;
2435     sp->end_seq = end_seq;
2436     tp->num_sacks++;
2437     tp->eff_sacks = min(tp->num_sacks+tp->dsack, 4-tp->tstamp_ok);
2438 }

```

```

2298 static __inline__ int
2299 tcp_sack_extend(struct tcp_sack_block *sp, u32 seq, u32 end_seq)
2300 {
2301     if (!after(seq, sp->end_seq) && !after(sp->start_seq, end_seq)) {
2302         if (before(seq, sp->start_seq))
2303             sp->start_seq = seq;
2304         if (after(end_seq, sp->end_seq))
2305             sp->end_seq = end_seq;
2306         return 1;
2307     }
2308     return 0;
2309 }

```

Figure 681: Recording the need to selectively acknowledge an out-of-order segment (i)

```

2355 /* These routines update the SACK block as out-of-order packets arrive or
2356 * in-order packets close up the sequence space.
2357 */
2358 static void tcp_sack_maybe_coalesce(struct tcp_opt *tp)
2359 {
2360     int this_sack;
2361     struct tcp_sack_block *sp = &tp->selective_acks[0];
2362     struct tcp_sack_block *swalk = sp+1;
2363
2364     /* See if the recent change to the first SACK eats into
2365     * or hits the sequence space of other SACK blocks, if so coalesce.
2366     */
2367     for (this_sack = 1; this_sack < tp->num_sacks; ){
2368         if (tcp_sack_extend(sp, swalk->start_seq, swalk->end_seq)) {
2369             int i;
2370
2371             /* Zap SWALK, by moving every further SACK up by one slot.
2372             * Decrease num_sacks.
2373             */
2374             tp->num_sacks--;
2375             tp->eff_sacks = min(tp->num_sacks+tp->dsack, 4-tp->
                tstamp_ok);
2376             for(i=this_sack; i < tp->num_sacks; i++)
2377                 sp[i] = sp[i+1];
2378             continue;
2379         }
2380         this_sack++, swalk++;
2381     }
2382 }
2383
2384 static __inline__ void tcp_sack_swap(struct tcp_sack_block *sack1, struct tcp_sack_block *
sack2)
2385 {
2386     __u32 tmp;
2387
2388     tmp = sack1->start_seq;
2389     sack1->start_seq = sack2->start_seq;
2390     sack2->start_seq = tmp;
2391
2392     tmp = sack1->end_seq;
2393     sack1->end_seq = sack2->end_seq;
2394     sack2->end_seq = tmp;
2395 }

```

Figure 682: Recording the need to selectively acknowledge an out-of-order segment (ii)

```

2311 static __inline__ void tcp_dsack_set(struct tcp_opt *tp, u32 seq, u32 end_seq)
2312 {
2313     if (tp->sack_ok && sysctl_tcp_dsack) {
2314         if (before(seq, tp->rcv_nxt))
2315             NET_INC_STATS_BH(TCPDSACKOldSent);
2316         else
2317             NET_INC_STATS_BH(TCPDSACKOfoSent);
2318
2319         tp->dsack = 1;
2320         tp->duplicate_sack[0].start_seq = seq;
2321         tp->duplicate_sack[0].end_seq = end_seq;
2322         tp->eff_sacks = min(tp->num_sacks+1, 4-tp->tstamp_ok);
2323     }
2324 }
2325
2326 static __inline__ void tcp_dsack_extend(struct tcp_opt *tp, u32 seq, u32 end_seq)
2327 {
2328     if (!tp->dsack)
2329         tcp_dsack_set(tp, seq, end_seq);
2330     else
2331         tcp_sack_extend(tp->duplicate_sack, seq, end_seq);
2332 }

```

Figure 683: Recording the need to send a dsack

```

1406 static __inline__ void tcp_build_and_update_options(__u32 *ptr, struct tcp_opt *tp, __u32
      timestamp)
1407 {
1408     if (tp->timestamp_ok) {
1409         *ptr++ = __constant_htonl((TCPOPT_NOP << 24) |
1410                                 (TCPOPT_NOP << 16) |
1411                                 (TCPOPT_TIMESTAMP << 8) |
1412                                 TCPOLEN_TIMESTAMP);
1413         *ptr++ = htonl(timestamp);
1414         *ptr++ = htonl(tp->ts_recent);
1415     }
1416     if (tp->eff_sacks) {
1417         struct tcp_sack_block *sp = tp->dsack ? tp->duplicate_sack : tp->
            selective_acks;
1418         int this_sack;
1419
1420         *ptr++ = __constant_htonl((TCPOPT_NOP << 24) |
1421                                 (TCPOPT_NOP << 16) |
1422                                 (TCPOPT_SACK << 8) |
1423                                 (TCPOLEN_SACK_BASE +
1424                                 (tp->eff_sacks *
1425                                 TCPOLEN_SACK_PERBLOCK)));
1426         for (this_sack = 0; this_sack < tp->eff_sacks; this_sack++) {
1427             *ptr++ = htonl(sp[this_sack].start_seq);
1428             *ptr++ = htonl(sp[this_sack].end_seq);
1429         }
1430         if (tp->dsack) {
1431             tp->dsack = 0;
1432             tp->eff_sacks--;
1433         }
1434     }

```

Figure 684: Building TCP option fields (timestamp and sack)

```

1028 static int tcp_check_sack_reneging(struct sock *sk, struct tcp_opt *tp)
1029 {
1030     struct sk_buff *skb;
1031
1032     /* If ACK arrived pointing to a remembered SACK,
1033      * it means that our remembered SACKs do not reflect
1034      * real state of receiver i.e.
1035      * receiver _host_ is heavily congested (or buggy).
1036      * Do processing similar to RTO timeout.
1037      */
1038     if ((skb = skb_peek(&sk->write_queue)) != NULL &&
1039         (TCP_SKB_CB(skb)->sacked & TCP_CB_SACKED_ACKED)) {
1040         NET_INC_STATS_BH(TCPSACKReneging);
1041
1042         tcp_enter_loss(sk, 1);
1043         tp->retransmits++;
1044         tcp_retransmit_skb(sk, skb_peek(&sk->write_queue));
1045         tcp_reset_xmit_timer(sk, TCP_TIME_RETRANS, tp->rto);
1046         return 1;
1047     }
1048     return 0;
1049 }

```

Figure 685: Checking for a sack reneging receiver

```

1797 static inline int tcp_paws_check(struct tcp_opt *tp, int rst)
1798 {
1799     if ((s32)(tp->rcv_tsval - tp->ts_recent) >= 0)
1800         return 0;
1801     if (xtime.tv_sec >= tp->ts_recent_stamp + TCP_PAWS_24DAYS)
1802         return 0;
1803
1804     /* RST segments are not recommended to carry timestamp,
1805        and, if they do, it is recommended to ignore PAWS because
1806        "their cleanup function should take precedence over timestamps."
1807        Certainly, it is mistake. It is necessary to understand the reasons
1808        of this constraint to relax it: if peer reboots, clock may go
1809        out-of-sync and half-open connections will not be reset.
1810        Actually, the problem would be not existing if all
1811        the implementations followed draft about maintaining clock
1812        via reboots. Linux-2.2 DOES NOT!
1813
1814        However, we can relax time bounds for RST segments to MSL.
1815        */
1816     if (rst && xtime.tv_sec >= tp->ts_recent_stamp + TCP_PAWS_MSL)
1817         return 0;
1818     return 1;
1819 }

```

Figure 686: PAWS checking


```

409 /* Called to compute a smoothed rtt estimate. The data fed to this
410 * routine either comes from timestamps, or from segments that were
411 * known _not_ to have been retransmitted [see Karn/Partridge
412 * Proceedings SIGCOMM 87]. The algorithm is from the SIGCOMM 88
413 * piece by Van Jacobson.
414 * NOTE: the next three routines used to be one big routine.
415 * To save cycles in the RFC 1323 implementation it was better to break
416 * it up into three procedures. -- erics
417 */
418 static __inline__ void tcp_rtt_estimator(struct tcp_opt *tp, __u32 mrtt)
419 {
420     long m = mrtt; /* RTT */
421
422     /* The following amusing code comes from Jacobson's
423     * article in SIGCOMM '88. Note that rtt and mdev
424     * are scaled versions of rtt and mean deviation.
425     * This is designed to be as fast as possible
426     * m stands for "measurement".
427     *
428     * On a 1990 paper the rto value is changed to:
429     * RTO = rtt + 4 * mdev
430     *
431     * Funny. This algorithm seems to be very broken.
432     * These formulae increase RTO, when it should be decreased, increase
433     * too slowly, when it should be increased fastly, decrease too fastly
434     * etc. I guess in BSD RTO takes ONE value, so that it is absolutely
435     * does not matter how to _calculate_ it. Seems, it was trap
436     * that VJ failed to avoid. 8)
437     */

```

Figure 687: Round Trip Estimation (i)

```

438     if (m == 0)
439         m = 1;
440     if (tp->srtt != 0) {
441         m -= (tp->srtt >> 3); /* m is now error in rtt est */
442         tp->srtt += m;        /* rtt = 7/8 rtt + 1/8 new */
443         if (m < 0) {
444             m = -m;          /* m is now abs(error) */
445             m -= (tp->mdev >> 2); /* similar update on mdev */
446             /* This is similar to one of Eifel findings.
447              * Eifel blocks mdev updates when rtt decreases.
448              * This solution is a bit different: we use finer gain
449              * for mdev in this case (alpha*beta).
450              * Like Eifel it also prevents growth of rto,
451              * but also it limits too fast rto decreases,
452              * happening in pure Eifel.
453              */
454             if (m > 0)
455                 m >>= 3;
456         } else {
457             m -= (tp->mdev >> 2); /* similar update on mdev */
458         }
459         tp->mdev += m;        /* mdev = 3/4 mdev + 1/4 new */
460         if (tp->mdev > tp->mdev_max) {
461             tp->mdev_max = tp->mdev;
462             if (tp->mdev_max > tp->rttvar)
463                 tp->rttvar = tp->mdev_max;
464         }
465         if (after(tp->snd_una, tp->rtt_seq)) {
466             if (tp->mdev_max < tp->rttvar)
467                 tp->rttvar -= (tp->rttvar - tp->mdev_max) >> 2;
468             tp->rtt_seq = tp->snd_una;
469             tp->mdev_max = TCP_RTO_MIN;
470         }
471     } else {
472         /* no previous measure. */
473         tp->srtt = m << 3;    /* take the measured time to be rtt */
474         tp->mdev = m << 2;   /* make sure rto = 3*rtt */
475         tp->mdev_max = tp->rttvar = max(tp->mdev, TCP_RTO_MIN);
476         tp->rtt_seq = tp->snd_nxt;
477     }
478 }

```

Figure 688: Round Trip Estimation (ii)

```

479
480 /* Calculate rto without backoff. This is the second half of Van Jacobson's
481 * routine referred to above.
482 */
483 static __inline__ void tcp_set_rto(struct tcp_opt *tp)
484 {
485     /* Old crap is replaced with new one. 8)
486     *
487     * More seriously:
488     * 1. If rtt variance happened to be less 50msec, it is hallucination.
489     *    It cannot be less due to utterly erratic ACK generation made
490     *    at least by solaris and freebsd. "Erratic ACKs" has _nothing_
491     *    to do with delayed acks, because at cwnd>2 true delack timeout
492     *    is invisible. Actually, Linux-2.4 also generates erratic
493     *    ACKs in some circumstances.
494     */
495     tp->rto = (tp->srtt >> 3) + tp->rttvar;
496
497     /* 2. Fixups made earlier cannot be right.
498     *    If we do not estimate RTO correctly without them,
499     *    all the algo is pure shit and should be replaced
500     *    with correct one. It is exactly, which we pretend to do.
501     */
502 }
503
504 /* NOTE: clamping at TCP_RTO_MIN is not required, current algo
505 * guarantees that rto is higher.
506 */
507 static __inline__ void tcp_bound_rto(struct tcp_opt *tp)
508 {
509     if (tp->rto > TCP_RTO_MAX)
510         tp->rto = TCP_RTO_MAX;
511 }

```

Figure 689: Round Trip Estimation (iii)

```

1436 /* Construct a tcp options header for a SYN or SYN_ACK packet.
1437 * If this is every changed make sure to change the definition of
1438 * MAX_SYN_SIZE to match the new maximum number of options that you
1439 * can generate.
1440 */
1441 static inline void tcp_syn_build_options(__u32 *ptr, int mss, int ts, int sack,
1442                                       int offer_wscale, int wscale, __u32 tstamp,
1443                                       __u32 ts_recent)
1444 {
1445     /* We always get an MSS option.
1446     * The option bytes which will be seen in normal data
1447     * packets should timestamps be used, must be in the MSS
1448     * advertised. But we subtract them from tp->mss_cache so
1449     * that calculations in tcp_sendmsg are simpler etc.
1450     * So account for this fact here if necessary. If we
1451     * don't do this correctly, as a receiver we won't
1452     * recognize data packets as being full sized when we
1453     * should, and thus we won't abide by the delayed ACK
1454     * rules correctly.
1455     * SACKs don't matter, we never delay an ACK when we
1456     * have any of those going out.
1457     */
1458     *ptr++ = htonl((TCPOPT_MSS << 24) | (TCPOLEN_MSS << 16) | mss);
1459     if (ts) {
1460         if (sack)
1461             *ptr++ = __constant_htonl((TCPOPT_SACK_PERM << 24) | (
1462                                     TCPOLEN_SACK_PERM << 16) |
1463                                     (TCPOPT_TIMESTAMP << 8) |
1464                                     TCPOLEN_TIMESTAMP);
1465         else
1466             *ptr++ = __constant_htonl((TCPOPT_NOP << 24) | (
1467                                     TCPOPT_NOP << 16) |
1468                                     (TCPOPT_TIMESTAMP << 8) |
1469                                     TCPOLEN_TIMESTAMP);
1470         *ptr++ = htonl(tstamp);      /* TSVAL */
1471         *ptr++ = htonl(ts_recent);   /* TSECR */
1472     } else if (sack)
1473         *ptr++ = __constant_htonl((TCPOPT_NOP << 24) | (TCPOPT_NOP
1474                                     << 16) |
1475                                     (TCPOPT_SACK_PERM << 8) |
1476                                     TCPOLEN_SACK_PERM);
1477     if (offer_wscale)
1478         *ptr++ = htonl((TCPOPT_NOP << 24) | (TCPOPT_WINDOW << 16)
1479                         | (TCPOLEN_WINDOW << 8) | (wscale));
1480 }

```

Figure 690: Building options on SYN packets

2301

```
if (! after(seq, sp->end_seq) && !after(sp->start_seq, end_seq)) {
```

Figure 691: Parsing the WINDOW option

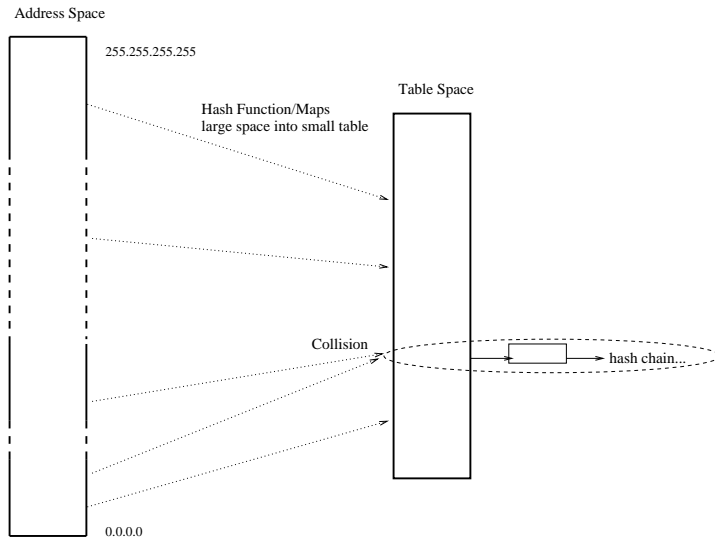


Figure 692: Hash Tables

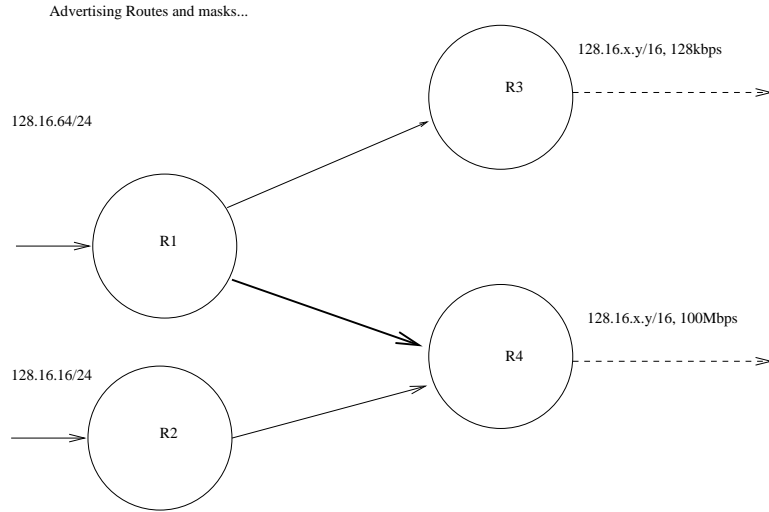


Figure 693: IP Addresses and Masks and Route Advertisements:
`linux{2.4.4/net/ipv4/tcp_input.c`

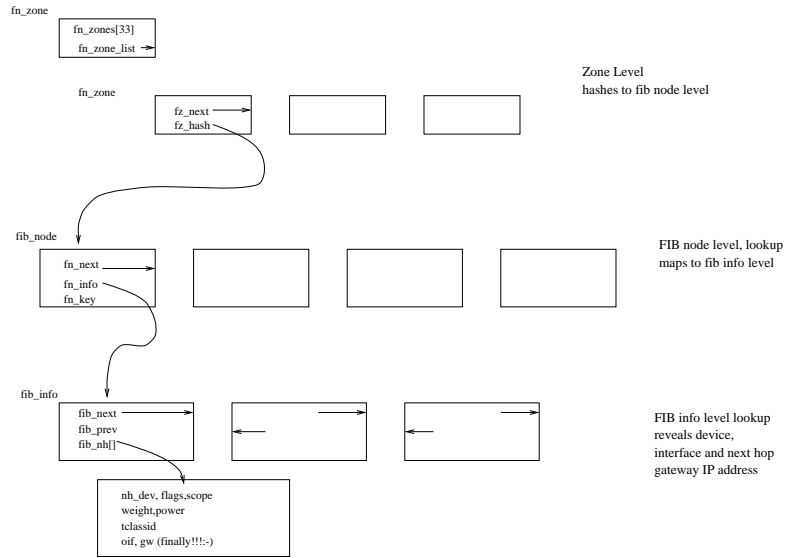


Figure 694: The FIB


```
21 struct kern_rta
22 {
23     void        *rta_dst;
24     void        *rta_src;
25     int         *rta_iif;
26     int         *rta_oif;
27     void        *rta_gw;
28     u32         *rta_priority;
29     void        *rta_prefsrc;
30     struct rtattr *rta_mx;
31     struct rtattr *rta_mp;
32     unsigned char *rta_protoinfo;
33     unsigned char *rta_flow;
34     struct rta_cacheinfo *rta_ci;
35 };
```

Figure 695: FIB Route Table Attribute Entry: `linux/include/net/ip_fib.h`

```
37 struct fib_nh
38 {
39     struct net_device      *nh_dev;
40     unsigned               nh_flags;
41     unsigned char          nh_scope;
42 #ifdef CONFIG_IP_ROUTE_MULTIPATH
43     int                    nh_weight;
44     int                    nh_power;
45 #endif
46 #ifdef CONFIG_NET_CLS_ROUTE
47     __u32                  nh_tclassid;
48 #endif
49     int                    nh_oif;
50     u32                    nh_gw;
51 };
```

Figure 696: FIB Next Hop Data Structure: `linux/include/net/ip_fib.h`

```

57 struct fib_info
58 {
59     struct fib_info    *fib_next ;
60     struct fib_info    *fib_prev ;
61     int                fib_treeref ;
62     atomic_t           fib_clntref ;
63     int                fib_dead ;
64     unsigned           fib_flags ;
65     int                fib_protocol ;
66     u32                fib_prefsrc ;
67     u32                fib_priority ;
68     unsigned           fib_metrics [RTAX_MAX];
69 #define fib_mtu fib_metrics [RTAX_MTU-1]
70 #define fib_window fib_metrics [RTAX_WINDOW-1]
71 #define fib_rtt fib_metrics [RTAX_RTT-1]
72 #define fib_advmss fib_metrics [RTAX_ADVMS-1]
73     int                fib_nhs ;
74 #ifdef CONFIG_IP_ROUTE_MULTIPATH
75     int                fib_power ;
76 #endif
77     struct fib_nh       fib_nh [0];
78 #define fib_dev        fib_nh [0]. nh_dev
79 };

```

Figure 697: The Main FIB Structure: `linux/include/net/ip_fib.h`

```
86 struct fib_result
87 {
88     unsigned char  prefixlen ;
89     unsigned char  nh_sel;
90     unsigned char  type;
91     unsigned char  scope;
92     struct fib_info * fi ;
93 #ifdef CONFIG_IP_MULTIPLE_TABLES
94     struct fib_rule *r;
95 #endif
96 };
```

Figure 698: FIB Lookup Result Structure: `linux/include/net/ip_fib.h`

```

116 struct fib_table
117 {
118     unsigned char  tb_id;
119     unsigned      tb_stamp;
120     int           (*tb_lookup)(struct fib_table *tb, const struct rt_key *key,
121                               struct fib_result *res);
121     int           (*tb_insert)(struct fib_table *table, struct rtmmsg *r,
122                               struct kern_rta *rta, struct nlmsgghdr *n,
123                               struct netlink_skb_parms *req);
124     int           (*tb_delete)(struct fib_table *table, struct rtmmsg *r,
125                               struct kern_rta *rta, struct nlmsgghdr *n,
126                               struct netlink_skb_parms *req);
127     int           (*tb_dump)(struct fib_table *table, struct sk_buff *skb,
128                               struct netlink_callback *cb);
129     int           (*tb_flush)(struct fib_table *table);
130     int           (*tb_get_info)(struct fib_table *table, char *buf,
131                                 int first , int count);
132     void          (*tb_select_default)(struct fib_table *table,
133                                       const struct rt_key *key, struct fib_result *
134                                       res);
134
135     unsigned char  tb_data [0];
136 };

```

Figure 699: FIB Generic Interface: `linux/include/net/ip_fib.h`

```

54 struct fib_table *local_table;
55 struct fib_table *main_table;
56
57 #else
58
59 #define RT_TABLE_MIN 1
60
61 struct fib_table *fib_tables [RT_TABLE_MAX+1];
62
63 struct fib_table *_fib_new_table (int id)
64 {
65     struct fib_table *tb;
66
67     tb = fib_hash_init (id);
68     if (!tb)
69         return NULL;
70     fib_tables [id] = tb;
71     return tb;
72 }

```

Figure 700: FIB Frontend is setting up the table:
linux/net/ipv4/fib_frontend.c

```

78 void fib_flush (void)
79 {
80     int flushed = 0;
81     #ifdef CONFIG_IP_MULTIPLE_TABLES
82         struct fib_table *tb;
83         int id;
84
85         for (id = RT_TABLE_MAX; id>0; id--) {
86             if ((tb = fib_get_table(id))==NULL)
87                 continue;
88             flushed += tb->tb_flush(tb);
89         }
90     #else /* CONFIG_IP_MULTIPLE_TABLES */
91         flushed += main_table->tb_flush(main_table);
92         flushed += local_table->tb_flush(local_table);
93     #endif /* CONFIG_IP_MULTIPLE_TABLES */
94
95     if (flushed)
96         rt_cache_flush (-1);
97 }

```

Figure 701: FIB Frontend i - flushing table: `linux/net/ipv4/fib_frontend.c`

```

100 #ifdef CONFIG_PROC_FS
101
102 /*
103  *   Called from the PROCfs module. This outputs /proc/net/route.
104  *
105  *   It always works in backward compatibility mode.
106  *   The format of the file is not supposed to be changed.
107  */
108
109 static int
110 fib_get_procinfo (char *buffer , char **start , off_t offset , int length)
111 {
112     int first = offset / 128;
113     char *ptr = buffer;
114     int count = (length + 127) / 128;
115     int len;
116
117     *start = buffer + offset % 128;
118
119     if (--first < 0) {
120         sprintf (buffer , "%-127s\n", "Iface\tDestination\tGateway_\tFlags\t
121             \tRefCnt\tUse\tMetric\tMask\t\tMTU\tWindow\tIRTT");
122         --count;
123         ptr += 128;
124         first = 0;
125     }
126
127     if (main_table && count > 0) {
128         int n = main_table->tb_get_info(main_table, ptr, first, count);
129         count -= n;
130         ptr += n * 128;
131     }
132     len = ptr - *start;
133     if (len >= length)
134         return length;
135     if (len >= 0)
136         return len;
137     return 0;
138 }
139 #endif /* CONFIG_PROC_FS */

```

Figure 702: FIB Frontend ProcFS interface: `linux/net/ipv4/fib_frontend.c`


```

141  /*
142  *    Find the first device with a given source address.
143  */
144
145  struct net_device * ip_dev_find(u32 addr)
146  {
147      struct rt_key key;
148      struct fib_result res;
149      struct net_device *dev = NULL;
150
151      memset(&key, 0, sizeof(key));
152      key.dst = addr;
153  #ifdef CONFIG_IP_MULTIPLE_TABLES
154      res.r = NULL;
155  #endif
156
157      if (!local_table || local_table->tb_lookup(local_table, &key, &res)) {
158          return NULL;
159      }
160      if (res.type != RTN_LOCAL)
161          goto out;
162      dev = FIB_RES_DEV(res);
163      if (dev)
164          atomic_inc(&dev->refcnt);
165
166  out:
167      fib_res_put(&res);
168      return dev;
169  }

```

Figure 703: FIB Front end - find device by address:
linux/net/ipv4/fib_frontend.c

```

171 unsigned inet_addr_type(u32 addr)
172 {
173     struct rt_key      key;
174     struct fib_result  res;
175     unsigned ret = RTN_BROADCAST;
176
177     if (ZERONET(addr) || BADCLASS(addr))
178         return RTN_BROADCAST;
179     if (MULTICAST(addr))
180         return RTN_MULTICAST;
181
182     memset(&key, 0, sizeof(key));
183     key.dst = addr;
184 #ifdef CONFIG_IP_MULTIPLE_TABLES
185     res.r = NULL;
186 #endif
187
188     if (local_table) {
189         ret = RTN_UNICAST;
190         if (local_table->tb_lookup(local_table, &key, &res) == 0) {
191             ret = res.type;
192             fib_res_put(&res);
193         }
194     }
195     return ret;
196 }

```

Figure 704: FIB Frontend - find by address type:
linux/net/ipv4/fib_frontend.c

```

198 /* Given (packet source, input interface) and optional (dst, oif, tos):
199 - (main) check, that source is valid i.e. not broadcast or our local
200 address.
201 - figure out what "logical" interface this packet arrived
202 and calculate "specific destination" address.
203 - check, that packet arrived from expected physical interface.
204 */
205
206 int fib_validate_source (u32 src, u32 dst, u8 tos, int oif,
207                        struct net_device *dev, u32 *spec_dst, u32 *itag)
208 {
209     struct in_device *in_dev;
210     struct rt_key key;
211     struct fib_result res;
212     int no_addr, rpf;
213     int ret;
214
215     key.dst = src;
216     key.src = dst;
217     key.tos = tos;
218     key.oif = 0;
219     key.iif = oif;
220     key.scope = RT_SCOPE_UNIVERSE;
221
222     no_addr = rpf = 0;
223     read_lock(&inetdev_lock);
224     in_dev = _in_dev_get(dev);
225     if (in_dev) {
226         no_addr = in_dev->ifa_list == NULL;
227         rpf = IN_DEV_RPFILTER(in_dev);
228     }
229     read_unlock(&inetdev_lock);
230
231     if (in_dev == NULL)
232         goto e_inval;
233
234     if (fib_lookup(&key, &res))
235         goto last_resort;
236     if (res.type != RTN_UNICAST)
237         goto e_inval_res;

```

Figure 705: FIB Frontend check address makes sense i:
linux/net/ipv4/fib_frontend.c

```

238     *spec_dst = FIB_RES_PREFSRC(res);
239     if (itag)
240         fib_combine_itag(itag, &res);
241 #ifdef CONFIG_IP_ROUTE_MULTIPATH
242     if (FIB_RES_DEV(res) == dev || res.fi->fib_nhs > 1)
243 #else
244     if (FIB_RES_DEV(res) == dev)
245 #endif
246     {
247         ret = FIB_RES_NH(res).nh_scope >= RT_SCOPE_HOST;
248         fib_res_put(&res);
249         return ret;
250     }
251     fib_res_put(&res);
252     if (no_addr)
253         goto last_resort;
254     if (rpf)
255         goto e_inval;
256     key.oif = dev->ifindex;
257
258     ret = 0;
259     if (fib_lookup(&key, &res) == 0) {
260         if (res.type == RTN_UNICAST) {
261             *spec_dst = FIB_RES_PREFSRC(res);
262             ret = FIB_RES_NH(res).nh_scope >= RT_SCOPE_HOST;
263         }
264         fib_res_put(&res);
265     }
266     return ret;
267
268 last_resort:
269     if (rpf)
270         goto e_inval;
271     *spec_dst = inet_select_addr(dev, 0, RT_SCOPE_UNIVERSE);
272     *itag = 0;
273     return 0;
274
275 e_inval_res:
276     fib_res_put(&res);
277 e_inval:
278     return -EINVAL;
279 }

```

Figure 706: FIB Frontend check address makes sense ii:
linux/net/ipv4/fib_frontend.c

```

283  /*
284  *   Handle IP routing ioctl calls. These are used to manipulate the routing
      tables
285  */
286
287  int ip_rt_ioctl (unsigned int cmd, void *arg)
288  {
289      int err;
290      struct kern_rta rta;
291      struct rtentry r;
292      struct {
293          struct nlmsgghdr nlh;
294          struct rtmsg rtm;
295      } req;
296
297      switch (cmd) {
298          case SIOCADDRT:      /* Add a route */
299          case SIOCDELRT:     /* Delete a route */
300              if (!capable(CAP_NET_ADMIN))
301                  return -EPERM;
302              if (copy_from_user(&r, arg, sizeof(struct rtentry)))
303                  return -EFAULT;
304              rtnl_lock ();
305              err = fib_convert_rtentry (cmd, &req.nlh, &req.rtm, &rta, &r);
306              if (err == 0) {
307                  if (cmd == SIOCDELRT) {
308                      struct fib_table *tb = fib_get_table (req.rtm.rtm_table);
309                      err = -ESRCH;
310                      if (tb)
311                          err = tb->tb_delete(tb, &req.rtm, &rta, &req.nlh
312                                              , NULL);
313                  } else {
314                      struct fib_table *tb = fib_new_table(req.rtm.rtm_table);
315                      err = -ENOBUFS;
316                      if (tb)
317                          err = tb->tb_insert(tb, &req.rtm, &rta, &req.nlh
318                                              , NULL);
319                  }
320                  if (rta.rta_mx)
321                      kfree(rta.rta_mx);
322              }
323              rtnl_unlock();
324              return err;
325          }
326      return -EINVAL;
327  }

```

Figure 707: FIB Frontend ioctl handler: `linux/net/ipv4/fib_frontend.c`

```

338 static int inet_check_attr(struct rtmsg *r, struct rtattr **rta)
339 {
340     int i;
341
342     for (i=1; i<=RTA_MAX; i++) {
343         struct rtattr *attr = rta[i-1];
344         if (attr) {
345             if (RTA_PAYLOAD(attr) < 4)
346                 return -EINVAL;
347             if (i != RTA_MULTIPATH && i != RTA_METRICS)
348                 rta[i-1] = (struct rtattr*)RTA_DATA(attr);
349         }
350     }
351     return 0;
352 }
353
354 int inet_rtm_delroute(struct sk_buff *skb, struct nlmsg_hdr* nlh, void *arg)
355 {
356     struct fib_table *tb;
357     struct rtattr **rta = arg;
358     struct rtmsg *r = NLMSG_DATA(nlh);
359
360     if (inet_check_attr(r, rta))
361         return -EINVAL;
362
363     tb = fib_get_table(r->rtm_table);
364     if (tb)
365         return tb->tb_delete(tb, r, (struct kern_rta*)rta, nlh, &NETLINK_CB(
366             skb));
367     return -ESRCH;
368 }
369
370 int inet_rtm_newroute(struct sk_buff *skb, struct nlmsg_hdr* nlh, void *arg)
371 {
372     struct fib_table *tb;
373     struct rtattr **rta = arg;
374     struct rtmsg *r = NLMSG_DATA(nlh);
375
376     if (inet_check_attr(r, rta))
377         return -EINVAL;
378
379     tb = fib_new_table(r->rtm_table);
380     if (tb)
381         return tb->tb_insert(tb, r, (struct kern_rta*)rta, nlh, &NETLINK_CB(
382             skb));
383     return -ENOBUFS;
384 }

```

Figure 708: FIB Frontend route find/add/delete:
linux/net/ipv4/fib_frontend.c

```

384 int inet_dump_fib(struct sk_buff *skb, struct netlink_callback *cb)
385 {
386     int t;
387     int s_t;
388     struct fib_table *tb;
389
390     if (NLMSG_PAYLOAD(cb->nlh, 0) >= sizeof(struct rtmsg) &&
391         ((struct rtmsg*)NLMSG_DATA(cb->nlh))->rtm_flags&RTM_F_CLONED)
392         return ip_rt_dump(skb, cb);
393
394     s_t = cb->args[0];
395     if (s_t == 0)
396         s_t = cb->args[0] = RT_TABLE_MIN;
397
398     for (t=s_t; t<=RT_TABLE_MAX; t++) {
399         if (t < s_t) continue;
400         if (t > s_t)
401             memset(&cb->args[1], 0, sizeof(cb->args)-sizeof(cb->args[0]));
402         if ((tb = fib_get_table(t)) == NULL)
403             continue;
404         if (tb->tb_dump(tb, skb, cb) < 0)
405             break;
406     }
407
408     cb->args[0] = t;
409
410     return skb->len;
411 }

```

Figure 709: FIB Frontend dump FIB: `linux/net/ipv4/fib_frontend.c`

```
415 /* Prepare and feed intra-kernel routing request.
416     Really, it should be netlink message, but :( netlink
417     can be not configured, so that we feed it directly
418     to fib engine. It is legal, because all events occur
419     only when netlink is already locked.
420 */
```

Figure 710: FIB Frontend - intra-kernel route message:
linux/net/ipv4/fib_frontend.c


```

422 static void fib_magic(int cmd, int type, u32 dst, int dst_len, struct in_ifaddr *ifa)
423 {
424     struct fib_table *tb;
425     struct {
426         struct nlmsg_hdr nlh;
427         struct rtm_msg rtm;
428     } req;
429     struct kern_rta rta;
430
431     memset(&req.rtm, 0, sizeof(req.rtm));
432     memset(&rta, 0, sizeof(rta));
433
434     if (type == RTN_UNICAST)
435         tb = fib_new_table(RT_TABLE_MAIN);
436     else
437         tb = fib_new_table(RT_TABLE_LOCAL);
438
439     if (tb == NULL)
440         return;
441
442     req.nlh.nlmsg_len = sizeof(req);
443     req.nlh.nlmsg_type = cmd;
444     req.nlh.nlmsg_flags = NLM_F_REQUEST|NLM_F_CREATE|NLM_F_APPEND;
445     req.nlh.nlmsg_pid = 0;
446     req.nlh.nlmsg_seq = 0;
447
448     req.rtm.rtm_dst_len = dst_len;
449     req.rtm.rtm_table = tb->tb_id;
450     req.rtm.rtm_protocol = RTPROT_KERNEL;
451     req.rtm.rtm_scope = (type != RTN_LOCAL ? RT_SCOPE_LINK :
452         RT_SCOPE_HOST);
453     req.rtm.rtm_type = type;

```

Figure 711: FIB Frontend - intra-kernel route message:
linux/net/ipv4/fib_frontend.c

```
454     rta.rta_dst = &dst;
455     rta.rta_prefsrc = &ifa->ifa_local;
456     rta.rta_oif = &ifa->ifa_dev->dev->ifindex;
457
458     if (cmd == RTM_NEWROUTE)
459         tb->tb_insert(tb, &req.rtm, &rta, &req.nlh, NULL);
460     else
461         tb->tb_delete(tb, &req.rtm, &rta, &req.nlh, NULL);
462 }
```

Figure 712: FIB Frontend - intra-kernel route message:
linux/net/ipv4/fib_frontend.c

```

464 static void fib_add_ifaddr(struct in_ifaddr *ifa)
465 {
466     struct in_device *in_dev = ifa->ifa_dev;
467     struct net_device *dev = in_dev->dev;
468     struct in_ifaddr *prim = ifa;
469     u32 mask = ifa->ifa_mask;
470     u32 addr = ifa->ifa_local;
471     u32 prefix = ifa->ifa_address&mask;
472
473     if (ifa->ifa_flags&IFA_F_SECONDARY) {
474         prim = inet_ifa_byprefix(in_dev, prefix, mask);
475         if (prim == NULL) {
476             printk(KERN_DEBUG "fib_add_ifaddr: bug: prim == NULL\n");
477             return;
478         }
479     }
480
481     fib_magic(RTM_NEWROUTE, RTN_LOCAL, addr, 32, prim);
482
483     if (!(dev->flags&IFF_UP))
484         return;
485
486     /* Add broadcast address, if it is explicitly assigned. */
487     if (ifa->ifa_broadcast && ifa->ifa_broadcast != 0xFFFFFFFF)
488         fib_magic(RTM_NEWROUTE, RTN_BROADCAST, ifa->ifa_broadcast
489                 , 32, prim);
490
491     if (!ZERONET(prefix) && !(ifa->ifa_flags&IFA_F_SECONDARY) &&
492         (prefix != addr || ifa->ifa_prefixlen < 32)) {
493         fib_magic(RTM_NEWROUTE, dev->flags&IFF_LOOPBACK ?
494                 RTN_LOCAL :
495                 RTN_UNICAST, prefix, ifa->ifa_prefixlen, prim);
496
497         /* Add network specific broadcasts, when it takes a sense */
498         if (ifa->ifa_prefixlen < 31) {
499             fib_magic(RTM_NEWROUTE, RTN_BROADCAST, prefix, 32,
500                     prim);
501             fib_magic(RTM_NEWROUTE, RTN_BROADCAST, prefix|~mask
502                     , 32, prim);
503         }
504     }
505 }

```

Figure 713: FIB Frontend add entry by interface address:
linux/net/ipv4/fib_frontend.c

```

503 static void fib_del_ifaddr (struct in_ifaddr *ifa)
504 {
505     struct in_device *in_dev = ifa->ifa_dev;
506     struct net_device *dev = in_dev->dev;
507     struct in_ifaddr *ifa1;
508     struct in_ifaddr *prim = ifa;
509     u32 brd = ifa->ifa_address|^ifa->ifa_mask;
510     u32 any = ifa->ifa_address&ifa->ifa_mask;
511     #define LOCAL_OK    1
512     #define BRD_OK      2
513     #define BRD0_OK     4
514     #define BRD1_OK     8
515     unsigned ok = 0;
516
517     if (!(ifa->ifa_flags&IFA_F_SECONDARY))
518         fib_magic(RTM_DELROUTE, dev->flags&IFF_LOOPBACK ?
519                 RTN_LOCAL :
520                 RTN_UNICAST, any, ifa->ifa_prefixlen, prim);
521     else {
522         prim = inet_ifa_byprefix (in_dev, any, ifa->ifa_mask);
523         if (prim == NULL) {
524             printk(KERN_DEBUG "fib_del_ifaddr: bug: prim == NULL\n");
525             return;
526         }
527
528         /* Deletion is more complicated than add.
529            We should take care of not to delete too much :->
530
531            Scan address list to be sure that addresses are really gone.
532            */
533
534         for (ifa1 = in_dev->ifa_list; ifa1; ifa1 = ifa1->ifa_next) {
535             if (ifa->ifa_local == ifa1->ifa_local)
536                 ok |= LOCAL_OK;
537             if (ifa->ifa_broadcast == ifa1->ifa_broadcast)
538                 ok |= BRD_OK;
539             if (brd == ifa1->ifa_broadcast)
540                 ok |= BRD1_OK;
541             if (any == ifa1->ifa_broadcast)
542                 ok |= BRD0_OK;
543         }

```

Figure 714: FIB Frontend delete entry by interface address i:
linux/net/ipv4/fib_frontend.c

```

545     if (!(ok&BRD_OK))
546         fib_magic(RTM_DELROUTE, RTN_BROADCAST, ifa->ifa_broadcast
547                 , 32, prim);
548     if (!(ok&BRD1_OK))
549         fib_magic(RTM_DELROUTE, RTN_BROADCAST, brd, 32, prim);
550     if (!(ok&BRD0_OK))
551         fib_magic(RTM_DELROUTE, RTN_BROADCAST, any, 32, prim);
552     if (!(ok&LOCAL_OK)) {
553         fib_magic(RTM_DELROUTE, RTN_LOCAL, ifa->ifa_local, 32, prim);
554
555         /* Check, that this local address finally disappeared. */
556         if (inet_addr_type(ifa->ifa_local) != RTN_LOCAL) {
557             /* And the last, but not the least thing.
558              * We must flush stray FIB entries.
559
560              * First of all, we scan fib_info list searching
561              * for stray nexthop entries, then ignite fib_flush.
562              */
563             if (fib_sync_down(ifa->ifa_local, NULL, 0))
564                 fib_flush ();
565         }
566     }
567 #undef LOCAL_OK
568 #undef BRD_OK
569 #undef BRD0_OK
570 #undef BRD1_OK
571 }

```

Figure 715: FIB Frontend delete entry by interface address ii:
linux/net/ipv4/fib_frontend.c

```
572 static void fib_disable_ip (struct net_device *dev, int force)
573 {
574     if (fib_sync_down(0, dev, force))
575         fib_flush ();
576     rt_cache_flush (0);
577     arp_ifdown(dev);
578 }
```

Figure 716: FIB Frontend Event Handler: `linux/net/ipv4/fib_frontend.c`

```

580 static int fib_inetaddr_event (struct notifier_block *this, unsigned long event, void *ptr)
581 {
582     struct in_ifaddr *ifa = (struct in_ifaddr*)ptr;
583
584     switch (event) {
585     case NETDEV_UP:
586         fib_add_ifaddr (ifa);
587         rt_cache_flush (-1);
588         break;
589     case NETDEV_DOWN:
590         if (ifa->ifa_dev && ifa->ifa_dev->ifa_list == NULL) {
591             /* Last address was deleted from this interface.
592              * Disable IP.
593              */
594             fib_disable_ip (ifa->ifa_dev->dev, 1);
595         } else {
596             fib_del_ifaddr (ifa);
597             rt_cache_flush (-1);
598         }
599         break;
600     }
601     return NOTIFY_DONE;
602 }

```

Figure 717: FIB Frontend Event Handler: `linux/net/ipv4/fib_frontend.c`

```

604 static int fib_netdev_event(struct notifier_block *this, unsigned long event, void *ptr)
605 {
606     struct net_device *dev = ptr;
607     struct in_device *in_dev = _in_dev_get(dev);
608
609     if (!in_dev)
610         return NOTIFY_DONE;
611
612     switch (event) {
613     case NETDEV_UP:
614         for_ifa(in_dev) {
615             fib_add_ifaddr(ifa);
616         } endfor_ifa(in_dev);
617     #ifdef CONFIG_IP_ROUTE_MULTIPATH
618         fib_sync_up(dev);
619     #endif
620         rt_cache_flush(-1);
621         break;
622     case NETDEV_DOWN:
623         fib_disable_ip(dev, 0);
624         break;
625     case NETDEV_UNREGISTER:
626         fib_disable_ip(dev, 1);
627         break;
628     case NETDEV_CHANGEMTU:
629     case NETDEV_CHANGE:
630         rt_cache_flush(0);
631         break;
632     }
633     return NOTIFY_DONE;
634 }

```

Figure 718: FIB Frontend Event Handler: `linux/net/ipv4/fib_frontend.c`


```

647 void _init ip_fib_init (void)
648 {
649     #ifdef CONFIG_PROC_FS
650         proc_net_create("route",0, fib_get_procinfo );
651     #endif          /* CONFIG_PROC_FS */
652
653     #ifndef CONFIG_IP_MULTIPLE_TABLES
654         local_table = fib_hash_init (RT_TABLE_LOCAL);
655         main_table = fib_hash_init (RT_TABLE_MAIN);
656     #else
657         fib_rules_init ();
658     #endif
659
660     register_netdevice_notifier (& fib_netdev_notifier );
661     register_inetaddr_notifier (& fib_inetaddr_notifier );
662 }
663

```

Figure 719: FIB Frontend initialisation: `linux/net/ipv4/fib_frontend.c`

```

48 static struct fib_info * fib_info_list ;
49 static rwlock_t fib_info_lock = RW_LOCK_UNLOCKED;
50 int fib_info_cnt ;
51
52 #define for_fib_info () { struct fib_info * fi ; \
53     for ( fi = fib_info_list ; fi ; fi = fi->fib_next)
54
55 #define endfor_fib_info () }
56
57 #ifndef CONFIG_IP_ROUTE_MULTIPATH
58
59 #define for_nexthops(fi) { int nhsel; const struct fib_nh * nh; \
60     for (nhsel=0, nh = (fi)->fib_nh; nhsel < (fi)->fib_nhs; nh++, nhsel++)
61
62 #define change_nexthops(fi) { int nhsel; struct fib_nh * nh; \
63     for (nhsel=0, nh = (struct fib_nh*)((fi)->fib_nh); nhsel < (fi)->fib_nhs; nh++, nhsel
64     ++)
65 #else /* CONFIG_IP_ROUTE_MULTIPATH */
66
67 /* Hope, that gcc will optimize it to get rid of dummy loop */
68
69 #define for_nexthops(fi) { int nhsel=0; const struct fib_nh * nh = (fi)->fib_nh; \
70     for (nhsel=0; nhsel < 1; nhsel++)
71
72 #define change_nexthops(fi) { int nhsel=0; struct fib_nh * nh = (struct fib_nh*)((fi)->
73     fib_nh); \
74     for (nhsel=0; nhsel < 1; nhsel++)
75 #endif /* CONFIG_IP_ROUTE_MULTIPATH */
76
77 #define endfor_nexthops(fi) }

```

Figure 720: FIB Semantics - looping macros over fib entries :
linux/net/ipv4/fib_semantics.c

```

80 static struct
81 {
82     int     error;
83     u8     scope;
84 } fib_props[RTA_MAX+1] = {
85     { 0, RT_SCOPE_NOWHERE}, /* RTN_UNSPEC */
86     { 0, RT_SCOPE_UNIVERSE}, /* RTN_UNICAST */
87     { 0, RT_SCOPE_HOST}, /* RTN_LOCAL */
88     { 0, RT_SCOPE_LINK}, /* RTN_BROADCAST */
89     { 0, RT_SCOPE_LINK}, /* RTN_ANYCAST */
90     { 0, RT_SCOPE_UNIVERSE}, /* RTN_MULTICAST */
91     {-EINVAL, RT_SCOPE_UNIVERSE}, /* RTN_BLACKHOLE */
92     {-EHOSTUNREACH, RT_SCOPE_UNIVERSE}, /* RTN_UNREACHABLE */
93     {-EACCES, RT_SCOPE_UNIVERSE}, /* RTN_PROHIBIT */
94     {-EAGAIN, RT_SCOPE_UNIVERSE}, /* RTN_THROW */
95 #ifdef CONFIG_IP_ROUTE_NAT
96     { 0, RT_SCOPE_HOST}, /* RTN_NAT */
97 #else
98     {-EINVAL, RT_SCOPE_NOWHERE}, /* RTN_NAT */
99 #endif
100     {-EINVAL, RT_SCOPE_NOWHERE} /* RTN_XRESOLVE */
101 };

```

Figure 721: FIB Semantics - initial Route Attribute properties:
linux/net/ipv4/fib_semantics.c

```

104 /* Release a nexthop info record */
105
106 void free_fib_info (struct fib_info *fi)
107 {
108     if (fi->fib_dead == 0) {
109         printk("Freeing alive fib_info %p\n", fi);
110         return;
111     }
112     change_nexthops(fi) {
113         if (nh->nh_dev)
114             dev_put(nh->nh_dev);
115         nh->nh_dev = NULL;
116     } endfor_nexthops(fi);
117     fib_info_cnt --;
118     kfree(fi);
119 }
120
121 void fib_release_info (struct fib_info *fi)
122 {
123     write_lock(&fib_info_lock);
124     if (fi && --fi->fib_reref == 0) {
125         if (fi->fib_next)
126             fi->fib_next->fib_prev = fi->fib_prev;
127         if (fi->fib_prev)
128             fi->fib_prev->fib_next = fi->fib_next;
129         if (fi == fib_info_list)
130             fib_info_list = fi->fib_next;
131         fi->fib_dead = 1;
132         fib_info_put(fi);
133     }
134     write_unlock(&fib_info_lock);
135 }

```

Figure 722: FIB Semantics - find and release next hop info:
linux/net/ipv4/fib_semantics.c

```

137 extern _inline_ int nh_comp(const struct fib_info * fi , const struct fib_info * ofi)
138 {
139     const struct fib_nh * onh = ofi->fib_nh;
140
141     for_nexthops(fi) {
142         if (nh->nh_oif != onh->nh_oif ||
143             nh->nh_gw != onh->nh_gw ||
144             nh->nh_scope != onh->nh_scope ||
145             #ifdef CONFIG_IP_ROUTE_MULTIPATH
146                 nh->nh_weight != onh->nh_weight ||
147             #endif
148             #ifdef CONFIG_NET_CLS_ROUTE
149                 nh->nh_tclassid != onh->nh_tclassid ||
150             #endif
151                 ((nh->nh_flags^onh->nh_flags)&~RTNH_F_DEAD))
152             return -1;
153         onh++;
154     } endfor_nexthops(fi);
155     return 0;
156 }
157
158 extern _inline_ struct fib_info * fib_find_info (const struct fib_info * nfi)
159 {
160     for_fib_info () {
161         if ( fi->fib_nhs != nfi->fib_nhs)
162             continue;
163         if ( nfi->fib_protocol == fi->fib_protocol &&
164             nfi->fib_prefsrc == fi->fib_prefsrc &&
165             nfi->fib_priority == fi->fib_priority &&
166             memcmp(nfi->fib_metrics, fi->fib_metrics, sizeof(fi->fib_metrics))
167                 == 0 &&
168             (( nfi->fib_flags^ fi->fib_flags)&~RTNH_F_DEAD) == 0 &&
169             (nfi->fib_nhs == 0 || nh_comp(fi, nfi) == 0))
170                 return fi ;
171     } endfor_fib_info ();
172     return NULL;
173 }

```

Figure 723: FIB Semantics - lookup: `linux/net/ipv4/fib_semantics.c`

```

174 /* Check, that the gateway is already configured.
175    Used only by redirect accept routine.
176 */
177
178 int ip_fib_check_default (u32 gw, struct net_device *dev)
179 {
180     read_lock(& fib_info_lock);
181     for_fib_info () {
182         if (fi->fib_flags & RTNH_F_DEAD)
183             continue;
184         for_nexthops(fi) {
185             if (nh->nh_dev == dev && nh->nh_gw == gw &&
186                 !(nh->nh_flags & RTNH_F_DEAD)) {
187                 read_unlock(& fib_info_lock);
188                 return 0;
189             }
190         } endfor_nexthops(fi);
191     } endfor_fib_info ();
192     read_unlock(& fib_info_lock);
193     return -1;
194 }
195
196 #ifdef CONFIG_IP_ROUTE_MULTIPATH
197
198 static u32 fib_get_attr32 (struct rtattr *attr, int attrlen, int type)
199 {
200     while (RTA_OK(attr, attrlen)) {
201         if (attr->rta_type == type)
202             return *(u32*)RTA_DATA(attr);
203         attr = RTA_NEXT(attr, attrlen);

```

Figure 724: FIB Semantics - set defaults: linux/net/ipv4/fib_semantics.c

```

205     return 0;
206 }
207
208 static int
209 fib_count_nexthops(struct rtattr *rta)
210 {
211     int nhs = 0;
212     struct rt nexthop *nhp = RTA_DATA(rta);
213     int nhlen = RTA_PAYLOAD(rta);
214
215     while (nhlen >= (int)sizeof(struct rt nexthop)) {
216         if ((nhlen -= nhp->rtnh_len) < 0)
217             return 0;
218         nhs++;
219         nhp = RTNH_NEXT(nhp);
220     };
221     return nhs;
222 }
223
224 static int
225 fib_get_nhs(struct fib_info *fi, const struct rtattr *rta, const struct rtmmsg *r)
226 {
227     struct rt nexthop *nhp = RTA_DATA(rta);
228     int nhlen = RTA_PAYLOAD(rta);
229
230     change_nexthops(fi) {
231         int attrlen = nhlen - sizeof(struct rt nexthop);
232         if (attrlen < 0 || (nhlen -= nhp->rtnh_len) < 0)
233             return -EINVAL;
234         nh->nh_flags = (r->rtm_flags & ~0xFFF) | nhp->rtnh_flags;
235         nh->nh_oif = nhp->rtnh_ifindex;
236         nh->nh_weight = nhp->rtnh_hops + 1;
237         if (attrlen) {
238             nh->nh_gw = fib_get_attr32(RTNH_DATA(nhp), attrlen,
239                                     RTA_GATEWAY);
240             #ifdef CONFIG_NET_CLS_ROUTE
241                 nh->nh_classid = fib_get_attr32(RTNH_DATA(nhp), attrlen,
242                                                 RTA_FLOW);
243             #endif
244         }
245         nhp = RTNH_NEXT(nhp);
246     } endfor_nexthops(fi);
247     return 0;
248 }

```

Figure 725: FIB Semantics -add up next hops and get nhs:
linux/net/ipv4/fib_semantics.c

```

250 int fib_nh_match(struct rtmsg *r, struct nlmsg_hdr *nlh, struct kern_rta *rta,
251                 struct fib_info *fi)
252 {
253 #ifdef CONFIG_IP_ROUTE_MULTIPATH
254     struct rt nexthop *nhp;
255     int nhlen;
256 #endif
257
258     if (rta->rta_priority &&
259         *rta->rta_priority != fi->fib_priority)
260         return 1;
261
262     if (rta->rta_oif || rta->rta_gw) {
263         if ((!rta->rta_oif || *rta->rta_oif == fi->fib_nh->nh_oif) &&
264             (!rta->rta_gw || memcmp(rta->rta_gw, &fi->fib_nh->nh_gw, 4)
265                 == 0))
266             return 0;
267         return 1;
268     }

```

Figure 726: FIB Semantics - next hops match:
linux/net/ipv4/fib_semantics.c


```

269 #ifdef CONFIG_IP_ROUTE_MULTIPATH
270     if (rta->rta_mp == NULL)
271         return 0;
272     nhp = RTA_DATA(rta->rta_mp);
273     nhlen = RTA_PAYLOAD(rta->rta_mp);
274
275     for_nexthops(fi) {
276         int attrlen = nhlen - sizeof(struct rt nexthop);
277         u32 gw;
278
279         if (attrlen < 0 || (nhlen -= nhp->rtnh_len) < 0)
280             return -EINVAL;
281         if (nhp->rtnh_ifindex && nhp->rtnh_ifindex != nh->nh_oif)
282             return 1;
283         if (attrlen) {
284             gw = fib_get_attr32(RTNH_DATA(nhp), attrlen, RTA_GATEWAY)
                ;
285             if (gw && gw != nh->nh_gw)
286                 return 1;
287 #ifdef CONFIG_NET_CLS_ROUTE
288             gw = fib_get_attr32(RTNH_DATA(nhp), attrlen, RTA_FLOW);
289             if (gw && gw != nh->nh_tclassid)
290                 return 1;
291 #endif
                }
292         nhp = RTNH_NEXT(nhp);
293     } endfor_nexthops(fi);
294 #endif
295     return 0;
296 }
297

```

Figure 727: FIB Semantics - next hops match.
linux/net/ipv4/fib_semantics.c

```

300  /*
301  Picture
302  -----
303
304  Semantics of nexthop is very messy by historical reasons.
305  We have to take into account, that:
306  a) gateway can be actually local interface address,
307     so that gatewayed route is direct.
308  b) gateway must be on-link address, possibly
309     described not by an ifaddr, but also by a direct route.
310  c) If both gateway and interface are specified, they should not
311     contradict.
312  d) If we use tunnel routes, gateway could be not on-link.
313
314  Attempt to reconcile all of these (alas, self-contradictory) conditions
315  results in pretty ugly and hairy code with obscure logic.
316
317  I choosed to generalized it instead, so that the size
318  of code does not increase practically, but it becomes
319  much more general.
320  Every prefix is assigned a "scope" value: "host" is local address,
321  "link" is direct route,
322  [ ... "site" ... "interior" ... ]
323  and "universe" is true gateway route with global meaning.
324
325  Every prefix refers to a set of "nexthop"s (gw, oif),
326  where gw must have narrower scope. This recursion stops
327  when gw has LOCAL scope or if "nexthop" is declared ONLINK,
328  which means that gw is forced to be on link.
329
330  Code is still hairy, but now it is apparently logically
331  consistent and very flexible. F.e. as by-product it allows
332  to co-exists in peace independent exterior and interior
333  routing processes.
334
335  Normally it looks as following.
336
337  {universe prefix} -> (gw, oif) [scope link]
338                    |
339                    |-> {link prefix} -> (gw, oif) [scope local]
340                    |
341                    |-> {local prefix} (terminal node)
342  */

```

Figure 728: FIB Semantics - the Big Picture:
linux/net/ipv4/fib_semantics.c

```

344 static int fib_check_nh(const struct rtmmsg *r, struct fib_info *fi, struct fib_nh *nh)
345 {
346     int err;
347
348     if (nh->nh_gw) {
349         struct rt_key key;
350         struct fib_result res;
351
352 #ifdef CONFIG_IP_ROUTE_PERVASIVE
353         if (nh->nh_flags&RTNH_F_PERVASIVE)
354             return 0;
355 #endif
356         if (nh->nh_flags&RTNH_F_ONLINK) {
357             struct net_device *dev;
358
359             if (r->rtm_scope >= RT_SCOPE_LINK)
360                 return -EINVAL;
361             if (inet_addr_type(nh->nh_gw) != RTN_UNICAST)
362                 return -EINVAL;
363             if ((dev = _dev_get_by_index(nh->nh_oif)) == NULL)
364                 return -ENODEV;
365             if (!(dev->flags&IFF_UP))
366                 return -ENETDOWN;
367             nh->nh_dev = dev;
368             atomic_inc(&dev->refcnt);
369             nh->nh_scope = RT_SCOPE_LINK;
370             return 0;
371         }
372         memset(&key, 0, sizeof(key));
373         key.dst = nh->nh_gw;
374         key.oif = nh->nh_oif;
375         key.scope = r->rtm_scope + 1;

```

Figure 729: FIB Semantics - Check Next Hop:
linux/net/ipv4/fib_semantics.c

```

377      /* It is not necessary, but requires a bit of thinking */
378      if (key.scope < RT_SCOPE_LINK)
379          key.scope = RT_SCOPE_LINK;
380
381      if ((err = fib_lookup(&key, &res)) != 0)
382          return err;
383      nh->nh_scope = res.scope;
384      nh->nh_oif = FIB_RES_OIF(res);
385      nh->nh_dev = FIB_RES_DEV(res);
386      if (nh->nh_dev)
387          atomic_inc(&nh->nh_dev->refcnt);
388      fib_res_put (&res);
389  } else {
390      struct in_device *in_dev;
391
392      if (nh->nh_flags&(RTNH_F_PERVASIVE|RTNH_F_ONLINK))
393          return -EINVAL;
394
395      in_dev = inetdev_by_index(nh->nh_oif);
396      if (in_dev == NULL)
397          return -ENODEV;
398      if (!(in_dev->dev->flags&IFF_UP)) {
399          in_dev_put(in_dev);
400          return -ENETDOWN;
401      }
402      nh->nh_dev = in_dev->dev;
403      atomic_inc(&nh->nh_dev->refcnt);
404      nh->nh_scope = RT_SCOPE_HOST;
405      in_dev_put(in_dev);
406  }
407  return 0;
408  }

```

Figure 730: FIB Semantics - Check Next Hop:
linux/net/ipv4/fib_semantics.c

```

410 struct fib_info *
411 fib_create_info (const struct rtmmsg *r, struct kern_rta *rta,
412                const struct nlmsg_hdr *nlh, int *errp)
413 {
414     int err;
415     struct fib_info *fi = NULL;
416     struct fib_info *ofi;
417 #ifdef CONFIG_IP_ROUTE_MULTIPATH
418     int nhs = 1;
419 #else
420     const int nhs = 1;
421 #endif
422
423     /* Fast check to catch the most weird cases */
424     if (fib_props[r->rtm_type].scope > r->rtm_scope)
425         goto err_inval;
426
427 #ifdef CONFIG_IP_ROUTE_MULTIPATH
428     if (rta->rt_a_mp) {
429         nhs = fib_count_nexthops(rta->rt_a_mp);
430         if (nhs == 0)
431             goto err_inval;
432     }
433 #endif

```

Figure 731: FIB Semantics - Create FIB Entry - i:
linux/net/ipv4/fib_semantics.c

```

435     fi = kmalloc(sizeof(*fi)+nhs*sizeof(struct fib_nh), GFP_KERNEL);
436     err = -ENOBUFS;
437     if (fi == NULL)
438         goto failure;
439     fib_info_cnt++;
440     memset(fi, 0, sizeof(*fi)+nhs*sizeof(struct fib_nh));
441
442     fi->fib_protocol = r->rtm_protocol;
443     fi->fib_nhs = nhs;
444     fi->fib_flags = r->rtm_flags;
445     if (rta->rta_priority)
446         fi->fib_priority = *rta->rta_priority;
447     if (rta->rta_mx) {
448         int attrlen = RTA_PAYLOAD(rta->rta_mx);
449         struct rtattr *attr = RTA_DATA(rta->rta_mx);
450
451         while (RTA_OK(attr, attrlen)) {
452             unsigned flavor = attr->rta_type;
453             if (flavor) {
454                 if (flavor > RTAX_MAX)
455                     goto err_inval;
456                 fi->fib_metrics[flavor-1] = *(unsigned*)RTA_DATA(attr);
457             }
458             attr = RTA_NEXT(attr, attrlen);
459         }
460     }
461     if (rta->rta_prefsrc)
462         memcpy(&fi->fib_prefsrc, rta->rta_prefsrc, 4);

```

Figure 732: FIB Semantics - Create FIB Entry - i:
linux/net/ipv4/fib_semantics.c

```

464     if (rta->rta_mp) {
465 #ifdef CONFIG_IP_ROUTE_MULTIPATH
466     if ((err = fib_get_nhs(fi, rta->rta_mp, r)) != 0)
467         goto failure;
468     if (rta->rta_oif && fi->fib_nh->nh_oif != *rta->rta_oif)
469         goto err_inval;
470     if (rta->rta_gw && memcmp(&fi->fib_nh->nh_gw, rta->rta_gw, 4))
471         goto err_inval;
472 #ifdef CONFIG_NET_CLS_ROUTE
473     if (rta->rta_flow && memcmp(&fi->fib_nh->nh_tclassid, rta->rta_flow
474         , 4))
475         goto err_inval;
476 #endif
477 #else
478     goto err_inval;
479 #endif
480     } else {
481     struct fib_nh *nh = fi->fib_nh;
482     if (rta->rta_oif)
483         nh->nh_oif = *rta->rta_oif;
484     if (rta->rta_gw)
485         memcpy(&nh->nh_gw, rta->rta_gw, 4);
486 #ifdef CONFIG_NET_CLS_ROUTE
487     if (rta->rta_flow)
488         memcpy(&nh->nh_tclassid, rta->rta_flow, 4);
489 #endif
490     nh->nh_flags = r->rtm_flags;
491 #ifdef CONFIG_IP_ROUTE_MULTIPATH
492     nh->nh_weight = 1;
493 #endif
494     }
495 #ifdef CONFIG_IP_ROUTE_NAT
496     if (r->rtm_type == RTN_NAT) {
497     if (rta->rta_gw == NULL || nhs != 1 || rta->rta_oif)
498         goto err_inval;
499     memcpy(&fi->fib_nh->nh_gw, rta->rta_gw, 4);
500     goto link_it;
501     }
502 #endif
503
504     if (fib_props[r->rtm_type].error) {
505     if (rta->rta_gw || rta->rta_oif || rta->rta_mp)
506         goto err_inval;
507     goto link_it;
508     }

```

Figure 733: FIB Semantics - Create FIB Entry - ii:
linux/net/ipv4/fib_semantics.c

```

510     if (r->rtm_scope > RT_SCOPE_HOST)
511         goto err_inval;
512
513     if (r->rtm_scope == RT_SCOPE_HOST) {
514         struct fib_nh *nh = fi->fib_nh;
515
516         /* Local address is added. */
517         if (nhs != 1 || nh->nh_gw)
518             goto err_inval;
519         nh->nh_scope = RT_SCOPE_NOWHERE;
520         nh->nh_dev = dev_get_by_index(fi->fib_nh->nh_oif);
521         err = -ENODEV;
522         if (nh->nh_dev == NULL)
523             goto failure;
524     } else {
525         change_nexthops(fi) {
526             if ((err = fib_check_nh(r, fi, nh)) != 0)
527                 goto failure;
528             } endfor_nexthops(fi)
529     }
530
531     if (fi->fib_prefsrc) {
532         if (r->rtm_type != RTN_LOCAL || rta->rta_dst == NULL ||
533             memcmp(&fi->fib_prefsrc, rta->rta_dst, 4))
534             if (inet_addr_type(fi->fib_prefsrc) != RTN_LOCAL)
535                 goto err_inval;
536     }

```

Figure 734: FIB Semantics - Create FIB Entry - iii:
linux/net/ipv4/fib_semantics.c


```

538 link_it :
539     if ((ofi = fib_find_info(fi)) != NULL) {
540         fi->fib_dead = 1;
541         free_fib_info(fi);
542         ofi->fib_treeref++;
543         return ofi;
544     }
545
546     fi->fib_treeref++;
547     atomic_inc(&fi->fib_clntref);
548     write_lock(&fib_info_lock);
549     fi->fib_next = fib_info_list;
550     fi->fib_prev = NULL;
551     if (fib_info_list)
552         fib_info_list->fib_prev = fi;
553     fib_info_list = fi;
554     write_unlock(&fib_info_lock);
555     return fi;
556
557 err_inval :
558     err = -EINVAL;
559
560 failure :
561     *errp = err;
562     if (fi) {
563         fi->fib_dead = 1;
564         free_fib_info(fi);
565     }
566     return NULL;
567 }

```

Figure 735: FIB Semantics - Create FIB Entry - iv:
linux/net/ipv4/fib_semantics.c

```

569 int
570 fib_semantic_match(int type, struct fib_info *fi, const struct rt_key *key, struct
      fib_result *res)
571 {
572     int err = fib_props[type].error;
573
574     if (err == 0) {
575         if (fi->fib_flags&RTNH_F_DEAD)
576             return 1;
577
578         res->fi = fi;
579
580         switch (type) {
581 #ifdef CONFIG_IP_ROUTE_NAT
582             case RTN_NAT:
583                 FIB_RES_RESET(*res);
584                 atomic_inc(&fi->fib_clntref);
585                 return 0;
586 #endif
587             case RTN_UNICAST:
588             case RTN_LOCAL:
589             case RTN_BROADCAST:
590             case RTN_ANYCAST:
591             case RTN_MULTICAST:
592                 for_nexthops(fi) {
593                     if (nh->nh_flags&RTNH_F_DEAD)
594                         continue;
595                     if (!key->oif || key->oif == nh->nh_oif)
596                         break;
597                 }

```

Figure 736: FIB Semantics - Meaningful Match?:
linux/net/ipv4/fib_semantics.c

```

598 #ifdef CONFIG_IP_ROUTE_MULTIPATH
599     if (nh_sel < fi->fib_nhs) {
600         res->nh_sel = nh_sel;
601         atomic_inc(&fi->fib_clntref);
602         return 0;
603     }
604 #else
605     if (nh_sel < 1) {
606         atomic_inc(&fi->fib_clntref);
607         return 0;
608     }
609 #endif
610     endfor_nexthops(fi);
611     res->fi = NULL;
612     return 1;
613     default:
614         res->fi = NULL;
615         printk(KERN_DEBUG "impossible_102\n");
616         return -EINVAL;
617     }
618     }
619     return err;
620 }

```

Figure 737: FIB Semantics - Meaningful Match?:
linux/net/ipv4/fib_semantics.c

```
622 /* Find appropriate source address to this destination */
623
624 u32 _fib_res_prefsrc (struct fib_result *res)
625 {
626     return inet_select_addr (FIB_RES_DEV(*res), FIB_RES_GW(*res), res->scope);
627 }
```

Figure 738: FIB Semantics - Dump utilities in:
linux/net/ipv4/fib_semantics.c

```

631 int
632 fib_dump_info(struct sk_buff *skb, u32 pid, u32 seq, int event,
633              u8 tb_id, u8 type, u8 scope, void *dst, int dst_len, u8 tos,
634              struct fib_info *fi)
635 {
636     struct rtmmsg *rtm;
637     struct nlmsg_hdr *nlh;
638     unsigned char *b = skb->tail;
639
640     nlh = NLMSG_PUT(skb, pid, seq, event, sizeof(*rtm));
641     rtm = NLMSG_DATA(nlh);
642     rtm->rtm_family = AF_INET;
643     rtm->rtm_dst_len = dst_len;
644     rtm->rtm_src_len = 0;
645     rtm->rtm_tos = tos;
646     rtm->rtm_table = tb_id;
647     rtm->rtm_type = type;
648     rtm->rtm_flags = fi->fib_flags;
649     rtm->rtm_scope = scope;
650     if (rtm->rtm_dst_len)
651         RTA_PUT(skb, RTA_DST, 4, dst);
652     rtm->rtm_protocol = fi->fib_protocol;
653     if (fi->fib_priority)
654         RTA_PUT(skb, RTA_PRIORITY, 4, &fi->fib_priority);
655 #ifdef CONFIG_NET_CLS_ROUTE
656     if (fi->fib_nh[0].nh_tclassid)
657         RTA_PUT(skb, RTA_FLOW, 4, &fi->fib_nh[0].nh_tclassid);
658 #endif
659     if (rtnetlink_put_metrics(skb, fi->fib_metrics) < 0)
660         goto rtattr_failure;
661     if (fi->fib_prefsrc)
662         RTA_PUT(skb, RTA_PREFSRC, 4, &fi->fib_prefsrc);
663     if (fi->fib_nhs == 1) {
664         if (fi->fib_nh->nh_gw)
665             RTA_PUT(skb, RTA_GATEWAY, 4, &fi->fib_nh->nh_gw);
666         if (fi->fib_nh->nh_oif)
667             RTA_PUT(skb, RTA_OIF, sizeof(int), &fi->fib_nh->nh_oif);
668     }

```

Figure 739: FIB Semantics - Dump utilities ii:
linux/net/ipv4/fib_semantics.c

```

669 #ifdef CONFIG_IP_ROUTE_MULTIPATH
670     if (fi->fib_nhs > 1) {
671         struct rt nexthop *nhp;
672         struct rtattr *mp_head;
673         if (skb_tailroom(skb) <= RTA_SPACE(0))
674             goto rtattr_failure ;
675         mp_head = (struct rtattr*)skb_put(skb, RTA_SPACE(0));
676
677         for_nexthops(fi) {
678             if (skb_tailroom(skb) < RTA_ALIGN(RTA_ALIGN(sizeof(*nhp))
679                 + 4))
680                 goto rtattr_failure ;
681             nhp = (struct rt nexthop*)skb_put(skb, RTA_ALIGN(sizeof(*nhp)))
682                 ;
683             nhp->rtnh_flags = nh->nh_flags & 0xFF;
684             nhp->rtnh_hops = nh->nh_weight - 1;
685             nhp->rtnh_ifindex = nh->nh_oif;
686             if (nh->nh_gw)
687                 RTA_PUT(skb, RTA_GATEWAY, 4, &nh->nh_gw);
688             nhp->rtnh_len = skb->tail - (unsigned char*)nhp;
689         } endfor_nexthops(fi);
690         mp_head->rta_type = RTA_MULTIPATH;
691         mp_head->rta_len = skb->tail - (u8*)mp_head;
692     }
693 #endif
694     nlh->nmsg_len = skb->tail - b;
695     return skb->len;
696
697 rtattr_failure :
698     skb_trim(skb, b - skb->data);
699     return -1;
700 }

```

Figure 740: FIB Semantics - Dump utilities iii:
linux/net/ipv4/fib_semantics.c

```

705 int
706 fib_convert_rtentry (int cmd, struct nlmsg_hdr *nl, struct rtm_msg *rtm,
707                     struct kern_rta *rta, struct rtentry *r)
708 {
709     int    plen;
710     u32    *ptr;
711
712     memset(rtm, 0, sizeof(*rtm));
713     memset(rta, 0, sizeof(*rta));
714
715     if (r->rt_dst.sa_family != AF_INET)
716         return -EAFNOSUPPORT;
717
718     /* Check mask for validity:
719      a) it must be contiguous.
720      b) destination must have all host bits clear.
721      c) if application forgot to set correct family (AF_INET),
722         reject request unless it is absolutely clear i.e.
723         both family and mask are zero.
724     */
725     plen = 32;
726     ptr = &((struct sockaddr_in*)&r->rt_dst)->sin_addr.s_addr;
727     if (!(r->rt_flags&RTF_HOST)) {
728         u32 mask = ((struct sockaddr_in*)&r->rt_genmask)->sin_addr.s_addr;
729         if (r->rt_genmask.sa_family != AF_INET) {
730             if (mask || r->rt_genmask.sa_family)
731                 return -EAFNOSUPPORT;
732         }
733         if (bad_mask(mask, *ptr))
734             return -EINVAL;
735         plen = inet_mask_len(mask);
736     }
737
738     nl->nmsg_flags = NLM_F_REQUEST;
739     nl->nmsg_pid = 0;
740     nl->nmsg_seq = 0;
741     nl->nmsg_len = NLMSG_LENGTH(sizeof(*rtm));
742     if (cmd == SIOCDELRT) {
743         nl->nmsg_type = RTM_DELROUTE;
744         nl->nmsg_flags = 0;
745     } else {
746         nl->nmsg_type = RTM_NEWROUTE;
747         nl->nmsg_flags = NLM_F_REQUEST|NLM_F_CREATE;
748         rtm->rtm_protocol = RTPROT_BOOT;
749     }

```

Figure 741: FIB Semantics - convert RT request:
linux/net/ipv4/fib_semantics.c

```

751     rtm->rtm_dst_len = plen;
752     rta->rta_dst = ptr;
753
754     if (r->rt_metric) {
755         *(u32*)&r->rt_pad3 = r->rt_metric - 1;
756         rta->rta_priority = (u32*)&r->rt_pad3;
757     }
758     if (r->rt_flags&RTF_REJECT) {
759         rtm->rtm_scope = RT_SCOPE_HOST;
760         rtm->rtm_type = RTN_UNREACHABLE;
761         return 0;
762     }
763     rtm->rtm_scope = RT_SCOPE_NOWHERE;
764     rtm->rtm_type = RTN_UNICAST;
765
766     if (r->rt_dev) {
767         char *colon;
768         struct net_device *dev;
769         char devname[IFNAMSIZ];
770
771         if (copy_from_user(devname, r->rt_dev, IFNAMSIZ-1))
772             return -EFAULT;
773         devname[IFNAMSIZ-1] = 0;
774         colon = strchr(devname, ':');
775         if (colon)
776             *colon = 0;
777         dev = __dev_get_by_name(devname);
778         if (!dev)
779             return -ENODEV;
780         rta->rta_oif = &dev->ifindex;
781         if (colon) {
782             struct in_ifaddr *ifa;
783             struct in_device *in_dev = __in_dev_get(dev);
784             if (!in_dev)
785                 return -ENODEV;
786             *colon = ':';
787             for (ifa = in_dev->ifa_list; ifa; ifa = ifa->ifa_next)
788                 if (strcmp(ifa->ifa_label, devname) == 0)
789                     break;
790             if (ifa == NULL)
791                 return -ENODEV;
792             rta->rta_prefsrc = &ifa->ifa_local;
793         }
794     }

```

Figure 742: FIB Semantics - export to RT entry i :
linux/net/ipv4/fib_semantics.c


```

796 ptr = &((struct sockaddr_in*)&r->rt_gateway)->sin_addr.s_addr;
797 if (r->rt_gateway.sa_family == AF_INET && *ptr) {
798     rta->rta_gw = ptr;
799     if (r->rt_flags&RTF_GATEWAY && inet_addr_type(*ptr) ==
RTN_UNICAST)
800         rtm->rtm_scope = RT_SCOPE_UNIVERSE;
801 }
802
803 if (cmd == SIOCDELRT)
804     return 0;
805
806 if (r->rt_flags&RTF_GATEWAY && rta->rta_gw == NULL)
807     return -EINVAL;
808
809 if (rtm->rtm_scope == RT_SCOPE_NOWHERE)
810     rtm->rtm_scope = RT_SCOPE_LINK;
811
812 if (r->rt_flags&(RTF_MTU|RTF_WINDOW|RTF_IRTT)) {
813     struct rtattr *rec;
814     struct rtattr *mx = kmalloc(RTA_LENGTH(3*RTA_LENGTH(4)),
GFP_KERNEL);
815     if (mx == NULL)
816         return -ENOMEM;
817     rta->rta_mx = mx;
818     mx->rta_type = RTA_METRICS;
819     mx->rta_len = RTA_LENGTH(0);
820     if (r->rt_flags&RTF_MTU) {
821         rec = (void*)((char*)mx + RTA_ALIGN(mx->rta_len));
822         rec->rta_type = RTAX_ADVMSS;
823         rec->rta_len = RTA_LENGTH(4);
824         mx->rta_len += RTA_LENGTH(4);
825         *(u32*)RTA_DATA(rec) = r->rt_mtu - 40;
826     }
827     if (r->rt_flags&RTF_WINDOW) {
828         rec = (void*)((char*)mx + RTA_ALIGN(mx->rta_len));
829         rec->rta_type = RTAX_WINDOW;
830         rec->rta_len = RTA_LENGTH(4);
831         mx->rta_len += RTA_LENGTH(4);
832         *(u32*)RTA_DATA(rec) = r->rt_window;
833     }
834     if (r->rt_flags&RTF_IRTT) {
835         rec = (void*)((char*)mx + RTA_ALIGN(mx->rta_len));
836         rec->rta_type = RTAX_IRTT;
837         rec->rta_len = RTA_LENGTH(4);
838         mx->rta_len += RTA_LENGTH(4);
839         *(u32*)RTA_DATA(rec) = r->rt_irtt<<3;
840     }
841 }
842 return 0;
843 }

```

Figure 743: FIB Semantics - export to RT entry ii :
linux/net/ipv4/fib_semantics.c

```
847 /*
848  Update FIB if:
849  - local address disappeared -> we must delete all the entries
850    referring to it.
851  - device went down -> we must shutdown all nexthops going via it.
852  */
```

Figure 744: FIB Semantics - update fib if address or interface down i:
linux/net/ipv4/fib_semantics.c

```

854 int fib_sync_down(u32 local, struct net_device *dev, int force)
855 {
856     int ret = 0;
857     int scope = RT_SCOPE_NOWHERE;
858
859     if (force)
860         scope = -1;
861
862     for_fib_info () {
863         if (local && fi->fib_prefsrc == local) {
864             fi->fib_flags |= RTNH_F_DEAD;
865             ret++;
866         } else if (dev && fi->fib_nhs) {
867             int dead = 0;
868
869             change_next_hops(fi) {
870                 if (nh->nh_flags & RTNH_F_DEAD)
871                     dead++;
872                 else if (nh->nh_dev == dev &&
873                         nh->nh_scope != scope) {
874                     nh->nh_flags |= RTNH_F_DEAD;
875 #ifdef CONFIG_IP_ROUTE_MULTIPATH
876                     fi->fib_power -= nh->nh_power;
877                     nh->nh_power = 0;
878 #endif
879                     dead++;
880                 }
881             } endfor_next_hops(fi)
882             if (dead == fi->fib_nhs) {
883                 fi->fib_flags |= RTNH_F_DEAD;
884                 ret++;
885             }
886         }
887     } endfor_fib_info ();
888     return ret;
889 }

```

Figure 745: FIB Semantics - update fib if address or interface down ii:
linux/net/ipv4/fib_semantics.c

```

893  /*
894   Dead device goes up. We wake up dead nexthops.
895   It takes sense only on multipath routes.
896  */
897
898  int fib_sync_up(struct net_device *dev)
899  {
900      int ret = 0;
901
902      if (!(dev->flags&IFF_UP))
903          return 0;
904
905      for_fib_info () {
906          int alive = 0;
907
908          change_nexthops(fi) {
909              if (!(nh->nh_flags&RTNH_F_DEAD)) {
910                  alive++;
911                  continue;
912              }
913              if (nh->nh_dev == NULL || !(nh->nh_dev->flags&IFF_UP))
914                  continue;
915              if (nh->nh_dev != dev || _in_dev_get(dev) == NULL)
916                  continue;
917              alive++;
918              nh->nh_power = 0;
919              nh->nh_flags &= ~RTNH_F_DEAD;
920          } endfor_nexthops(fi)
921
922          if (alive > 0) {
923              fi->fib_flags &= ~RTNH_F_DEAD;
924              ret++;
925          }
926      } endfor_fib_info ();
927      return ret;
928  }

```

Figure 746: FIB Semantics - update fib if address or interface up!
linux/net/ipv4/fib_semantics.c

```

930  /*
931     The algorithm is suboptimal, but it provides really
932     fair weighted route distribution.
933  */
934
935  void fib_select_multipath (const struct rt_key *key, struct fib_result *res)
936  {
937     struct fib_info *fi = res->fi;
938     int w;
939
940     if (fi->fib_power <= 0) {
941         int power = 0;
942         change_nexthops(fi) {
943             if (!(nh->nh_flags&RTNH_F_DEAD)) {
944                 power += nh->nh_weight;
945                 nh->nh_power = nh->nh_weight;
946             }
947         } endfor_nexthops(fi);
948         fi->fib_power = power;
949
950     #if 1
951         if (power <= 0) {
952             printk(KERN_CRIT "impossible_777\n");
953             return;
954         }
955     #endif
956
957
958     /* w should be random number [0..fi->fib_power-1],
959        it is pretty bad approximation.
960     */
961
962     w = jiffies % fi->fib_power;

```

Figure 747: FIB Semantics - multipath case i:
linux/net/ipv4/fib_semantics.c

```

964     change_nexthops(fi) {
965         if (!(nh->nh_flags&RTNH_F_DEAD) && nh->nh_power) {
966             if ((w -= nh->nh_power) <= 0) {
967                 nh->nh_power--;
968                 fi->fib_power--;
969                 res->nh_sel = nh_sel;
970                 return;
971             }
972         }
973     } endfor_nexthops(fi);
974
975     #if 1
976     printk(KERN_CRIT "impossible_888\n");
977     #endif
978     return;
979 }

```

Figure 748: FIB Semantics - multipath case ii:
linux/net/ipv4/fib_semantics.c

```

985 static unsigned fib_flag_trans (int type, int dead, u32 mask, struct fib_info * fi)
986 {
987     static unsigned type2flags[RTN_MAX+1] = {
988         0, 0, 0, 0, 0, 0, 0, 0,  RTF_REJECT, RTF_REJECT, 0, 0, 0
989     };
990     unsigned flags = type2flags[type];
991
992     if ( fi && fi->fib_nh->nh_gw)
993         flags |= RTF_GATEWAY;
994     if (mask == 0xFFFFFFFF)
995         flags |= RTF_HOST;
996     if (! dead)
997         flags |= RTF_UP;
998     return flags;
999 }
1000
1001 void fib_node_get_info (int type, int dead, struct fib_info * fi , u32 prefix , u32 mask,
1002     char *buffer)
1003 {
1004     int len;
1005     unsigned flags = fib_flag_trans (type, dead, mask, fi);
1006
1007     if ( fi ) {
1008         len = sprintf (buffer, "%s\t%08X\t%08X\t%04X\t%d\t%u\t%d\t%08X\t%08X\t%u\t%u",
1009             fi->fib_dev ? fi->fib_dev->name : "*", prefix,
1010             fi->fib_nh->nh_gw, flags, 0, 0, fi->fib_priority,
1011             mask, fi->fib_advmss+40, fi->fib_window, fi->fib_rtt>>3)
1012         ;
1013     } else {
1014         len = sprintf (buffer, "*\t%08X\t%08X\t%04X\t%d\t%u\t%d\t%08X\t%08X\t%u\t%u",
1015             prefix , 0,
1016             flags , 0, 0, 0,
1017             mask, 0, 0, 0) ;
1018     }
1019     memset (buffer+len, '\0', 127-len);
1020     buffer[127] = '\n';
1021 }

```

Figure 749: FIB Semantics - PROC FS Interface:
linux/net/ipv4/fib_semantics.c

```

52 struct fib_rule
53 {
54     struct fib_rule *r_next;
55     atomic_t        r_clntref;
56     u32             r_preference;
57     unsigned char   r_table;
58     unsigned char   r_action;
59     unsigned char   r_dst_len;
60     unsigned char   r_src_len;
61     u32             r_src;
62     u32             r_srcmask;
63     u32             r_dst;
64     u32             r_dstmask;
65     u32             r_srcmap;
66     u8              r_flags;
67     u8              r_tos;
68 #ifdef CONFIG_IP_ROUTE_FWMARK
69     u32             r_fwmark;
70 #endif
71     int             r_ifindex;
72 #ifdef CONFIG_NET_CLS_ROUTE
73     _u32            r_tclassid;
74 #endif
75     char            r_ifname[IFNAMSIZ];
76     int             r_dead;
77 };
78
79 static struct fib_rule default_rule = { NULL, ATOMIC_INIT(2), 0x7FFF,
80     RT_TABLE_DEFAULT, RTN_UNICAST, };
81 static struct fib_rule main_rule = { &default_rule, ATOMIC_INIT(2), 0x7FFE,
82     RT_TABLE_MAIN, RTN_UNICAST, };
83 static struct fib_rule local_rule = { &main_rule, ATOMIC_INIT(2), 0,
84     RT_TABLE_LOCAL, RTN_UNICAST, };
85
86 static struct fib_rule *fib_rules = &local_rule;
87 static rwlock_t fib_rules_lock = RW_LOCK_UNLOCKED;

```

Figure 750: FIB Rules - globals: `linux/net/ipv4/fib_rules.c`


```

86 int inet_rtm_delrule(struct sk_buff *skb, struct nlmsg_hdr* nlh, void *arg)
87 {
88     struct rtattr **rta = arg;
89     struct rtm_msg *rtm = NLMSG_DATA(nlh);
90     struct fib_rule *r, **rp;
91     int err = -ESRCH;
92
93     for (rp=&fib_rules; (r=*rp) != NULL; rp=&r->r_next) {
94         if ((! rta[RTA_SRC-1] || memcmp(RTA_DATA(rta[RTA_SRC-1]), &r->
95             r_src, 4) == 0) &&
96             rtm->rtm_src_len == r->r_src_len &&
97             rtm->rtm_dst_len == r->r_dst_len &&
98             (! rta[RTA_DST-1] || memcmp(RTA_DATA(rta[RTA_DST-1]), &r->
99                 r_dst, 4) == 0) &&
100             rtm->rtm_tos == r->r_tos &&
101             #ifdef CONFIG_IP_ROUTE_FWMARK
102                 (! rta[RTA_PROTOINFO-1] || memcmp(RTA_DATA(rta[
103                     RTA_PROTOINFO-1]), &r->r_fwmark, 4) == 0) &&
104             #endif
105             (! rtm->rtm_type || rtm->rtm_type == r->r_action) &&
106             (! rta[RTA_PRIORITY-1] || memcmp(RTA_DATA(rta[
107                 RTA_PRIORITY-1]), &r->r_preference, 4) == 0) &&
108             (! rta[RTA_IIF-1] || strcmp(RTA_DATA(rta[RTA_IIF-1]), r->
109                 r_ifname) == 0) &&
110             (! rtm->rtm_table || (r && rtm->rtm_table == r->r_table))) {
111             err = -EPERM;
112             if (r == &local_rule)
113                 break;
114
115             write_lock_bh(& fib_rules_lock);
116             *rp = r->r_next;
117             r->r_dead = 1;
118             write_unlock_bh(& fib_rules_lock);
119             fib_rule_put(r);
120             err = 0;
121             break;
122         }
123     }
124     return err;
125 }

```

Figure 751: FIB Rules i: linux/net/ipv4/fib_rules.c

```

122  /* Allocate new unique table id */
123
124  static struct fib_table *fib_empty_table(void)
125  {
126      int id;
127
128      for (id = 1; id <= RT_TABLE_MAX; id++)
129          if (fib_tables[id] == NULL)
130              return _fib_new_table(id);
131      return NULL;
132  }
133
134  void fib_rule_put (struct fib_rule *r)
135  {
136      if (atomic_dec_and_test(&r->r_clntref)) {
137          if (r->r_dead)
138              kfree(r);
139          else
140              printk("Freeing alive rule %p\n", r);
141      }
142  }

```

Figure 752: FIB Rules - utils: linux/net/ipv4/fib_rules.c

```

144 int inet_rtm_newrule(struct sk_buff *skb, struct nlmsg_hdr* nlh, void *arg)
145 {
146     struct rtattr **rta = arg;
147     struct rtm_msg *rtm = NLMSG_DATA(nlh);
148     struct fib_rule *r, *new_r, **rp;
149     unsigned char table_id;
150
151     if (rtm->rtm_src_len > 32 || rtm->rtm_dst_len > 32 ||
152         (rtm->rtm_tos & ~IPTOS_TOS_MASK))
153         return -EINVAL;
154
155     if (rta[RTA_IIF-1] && RTA_PAYLOAD(rta[RTA_IIF-1]) > IFNAMSIZ)
156         return -EINVAL;
157
158     table_id = rtm->rtm_table;
159     if (table_id == RT_TABLE_UNSPEC) {
160         struct fib_table *table;
161         if (rtm->rtm_type == RTN_UNICAST || rtm->rtm_type == RTN_NAT
162             ) {
163             if ((table = fib_empty_table()) == NULL)
164                 return -ENOBUFS;
165             table_id = table->tb_id;
166         }
167     }

```

Figure 753: FIB Rules - new rule: `linux/net/ipv4/fib_rules.c`

```

168     new_r = kmalloc(sizeof(*new_r), GFP_KERNEL);
169     if (!new_r)
170         return -ENOMEM;
171     memset(new_r, 0, sizeof(*new_r));
172     if (rta[RTA_SRC-1])
173         memcpy(&new_r->r_src, RTA_DATA(rta[RTA_SRC-1]), 4);
174     if (rta[RTA_DST-1])
175         memcpy(&new_r->r_dst, RTA_DATA(rta[RTA_DST-1]), 4);
176     if (rta[RTA_GATEWAY-1])
177         memcpy(&new_r->r_srcmap, RTA_DATA(rta[RTA_GATEWAY-1]), 4);
178     new_r->r_src_len = rtm->rtm_src_len;
179     new_r->r_dst_len = rtm->rtm_dst_len;
180     new_r->r_srcmask = inet_make_mask(rtm->rtm_src_len);
181     new_r->r_dstmask = inet_make_mask(rtm->rtm_dst_len);
182     new_r->r_tos = rtm->rtm_tos;
183 #ifdef CONFIG_IP_ROUTE_FWMARK
184     if (rta[RTA_PROTOINFO-1])
185         memcpy(&new_r->r_fwmark, RTA_DATA(rta[RTA_PROTOINFO-1]), 4);
186 #endif
187     new_r->r_action = rtm->rtm_type;
188     new_r->r_flags = rtm->rtm_flags;
189     if (rta[RTA_PRIORITY-1])
190         memcpy(&new_r->r_preferance, RTA_DATA(rta[RTA_PRIORITY-1]), 4);
191     new_r->r_table = table_id;
192     if (rta[RTA_IIF-1]) {
193         struct net_device *dev;
194         memcpy(new_r->r_ifname, RTA_DATA(rta[RTA_IIF-1]), IFNAMSIZ);
195         new_r->r_ifname[IFNAMSIZ-1] = 0;
196         new_r->r_ifindex = -1;
197         dev = __dev_get_by_name(new_r->r_ifname);
198         if (dev)
199             new_r->r_ifindex = dev->ifindex;
200     }
201 #ifdef CONFIG_NET_CLS_ROUTE
202     if (rta[RTA_FLOW-1])
203         memcpy(&new_r->r_tclassid, RTA_DATA(rta[RTA_FLOW-1]), 4);
204 #endif

```

Figure 754: FIB Rules - new rule: `linux/net/ipv4/fib_rules.c`

```

206     rp = &fib_rules;
207     if (!new_r->r_preference) {
208         r = fib_rules;
209         if (r && (r = r->r_next) != NULL) {
210             rp = &fib_rules->r_next;
211             if (r->r_preference)
212                 new_r->r_preference = r->r_preference - 1;
213         }
214     }
215
216     while ((r = *rp) != NULL){
217         if (r->r_preference > new_r->r_preference)
218             break;
219         rp = &r->r_next;
220     }
221
222     new_r->r_next = r;
223     atomic_inc(&new_r->r_clntref);
224     write_lock_bh(&fib_rules_lock);
225     *rp = new_r;
226     write_unlock_bh(&fib_rules_lock);
227     return 0;
228 }

```

Figure 755: FIB Rules - new rule: `linux/net/ipv4/fib_rules.c`

```

230 u32 fib_rules_map_destination (u32 daddr, struct fib_result *res)
231 {
232     u32 mask = inet_make_mask(res->prefixlen);
233     return (daddr & ~mask) | res->fi->fib_nh->nh_gw;
234 }
235
236 u32 fib_rules_policy (u32 saddr, struct fib_result *res, unsigned *flags)
237 {
238     struct fib_rule *r = res->r;
239
240     if (r->r_action == RTN_NAT) {
241         int addrtype = inet_addr_type(r->r_srcmap);
242
243         if (addrtype == RTN_NAT) {
244             /* Packet is from translated source; remember it */
245             saddr = (saddr & ~r->r_srcmask) | r->r_srcmap;
246             *flags |= RTCF_SNAT;
247         } else if (addrtype == RTN_LOCAL || r->r_srcmap == 0) {
248             /* Packet is from masqueraded source; remember it */
249             saddr = r->r_srcmap;
250             *flags |= RTCF_MASQ;
251         }
252     }
253     return saddr;
254 }

```

Figure 756: FIB Rules - more utils: `linux/net/ipv4/fib_rules.c`

```

256 #ifdef CONFIG_NET_CLS_ROUTE
257 u32 fib_rules_tclass (struct fib_result *res)
258 {
259     if (res->r)
260         return res->r->r_tclassid;
261     return 0;
262 }
263 #endif
264
265
266 static void fib_rules_detach (struct net_device *dev)
267 {
268     struct fib_rule *r;
269
270     for (r=fib_rules; r; r=r->r_next) {
271         if (r->r_ifindex == dev->ifindex) {
272             write_lock_bh(& fib_rules_lock);
273             r->r_ifindex = -1;
274             write_unlock_bh(& fib_rules_lock);
275         }
276     }
277 }
278
279 static void fib_rules_attach (struct net_device *dev)
280 {
281     struct fib_rule *r;
282
283     for (r=fib_rules; r; r=r->r_next) {
284         if (r->r_ifindex == -1 && strcmp(dev->name, r->r_ifname) == 0) {
285             write_lock_bh(& fib_rules_lock);
286             r->r_ifindex = dev->ifindex;
287             write_unlock_bh(& fib_rules_lock);
288         }
289     }
290 }

```

Figure 757: FIB Rules - attach and detach rule: `linux/net/ipv4/fib_rules.c`

```

292 int fib_lookup(const struct rt_key *key, struct fib_result *res)
293 {
294     int err;
295     struct fib_rule *r, *policy;
296     struct fib_table *tb;
297
298     u32 daddr = key->dst;
299     u32 saddr = key->src;
300
301     FRprintk("Lookup: %u.%u.%u.%u < - %u.%u.%u.%u",
302             NIPQUAD(key->dst), NIPQUAD(key->src));
303     read_lock(&fib_rules_lock);
304     for (r = fib_rules; r; r = r->r_next) {
305         if (((saddr ^ r->r_src) & r->r_srcmask) ||
306             ((daddr ^ r->r_dst) & r->r_dstmask) ||
307             #ifdef CONFIG_IP_ROUTE_TOS
308                 (r->r_tos && r->r_tos != key->tos) ||
309             #endif
310             #ifdef CONFIG_IP_ROUTE_FWMARK
311                 (r->r_fwmark && r->r_fwmark != key->fwmark) ||
312             #endif
313                 (r->r_ifindex && r->r_ifindex != key->iif))
314             continue;
315
316     FRprintk("tb %d_r %d", r->r_table, r->r_action);
317     switch (r->r_action) {
318     case RTN_UNICAST:
319     case RTN_NAT:
320         policy = r;
321         break;
322     case RTN_UNREACHABLE:
323         read_unlock(&fib_rules_lock);
324         return -ENETUNREACH;
325     default:
326     case RTN_BLACKHOLE:
327         read_unlock(&fib_rules_lock);
328         return -EINVAL;
329     case RTN_PROHIBIT:
330         read_unlock(&fib_rules_lock);
331         return -EACCES;
332     }

```

Figure 758: FIB Rules -lookup rule: `linux/net/ipv4/fib_rules.c`


```

334         if ((tb = fib_get_table(r->r_table)) == NULL)
335             continue;
336         err = tb->tb_lookup(tb, key, res);
337         if (err == 0) {
338             res->r = policy;
339             if (policy)
340                 atomic_inc(&policy->r_clntref);
341             read_unlock(&fib_rules_lock);
342             return 0;
343         }
344         if (err < 0 && err != -EAGAIN) {
345             read_unlock(&fib_rules_lock);
346             return err;
347         }
348     }
349     FRprintk("FAILURE\n");
350     read_unlock(&fib_rules_lock);
351     return -ENETUNREACH;
352 }

```

Figure 759: FIB Rules -lookup rule: `linux/net/ipv4/fib_rules.c`

```

354 void fib_select_default (const struct rt_key *key, struct fib_result *res)
355 {
356     if (res->r && res->r->r_action == RTN_UNICAST &&
357         FIB_RES_GW(*res) && FIB_RES_NH(*res).nh_scope == RT_SCOPE_LINK) {
358         struct fib_table *tb;
359         if ((tb = fib_get_table(res->r->r_table)) != NULL)
360             tb->tb_select_default(tb, key, res);
361     }
362 }
363
364 static int fib_rules_event (struct notifier_block *this, unsigned long event, void *ptr)
365 {
366     struct net_device *dev = ptr;
367
368     if (event == NETDEV_UNREGISTER)
369         fib_rules_detach (dev);
370     else if (event == NETDEV_REGISTER)
371         fib_rules_attach (dev);
372     return NOTIFY_DONE;
373 }
374
375
376 struct notifier_block fib_rules_notifier = {
377     fib_rules_event,
378     NULL,
379     0
380 };

```

Figure 760: FIB Rules Events: `linux/net/ipv4/fib_rules.c`

```

384 extern _inline_ int inet_fill_rule (struct sk_buff *skb,
385                                   struct fib_rule *r,
386                                   struct netlink_callback *cb)
387 {
388     struct rtmmsg *rtm;
389     struct nlmsg_hdr *nlh;
390     unsigned char *b = skb->tail;
391
392     nlh = NLMSG_PUT(skb, NETLINK_CREDS(cb->skb)->pid, cb->nlh->
393                 nlmseq, RTM_NEWRULE, sizeof(*rtm));
394     rtm = NLMSG_DATA(nlh);
395     rtm->rtm_family = AF_INET;
396     rtm->rtm_dst_len = r->r_dst_len;
397     rtm->rtm_src_len = r->r_src_len;
398     rtm->rtm_tos = r->r_tos;
399 #ifdef CONFIG_IP_ROUTE_FWMARK
400     if (r->r_fwmark)
401         RTA_PUT(skb, RTA_PROTOINFO, 4, &r->r_fwmark);
402 #endif
403     rtm->rtm_table = r->r_table;
404     rtm->rtm_protocol = 0;
405     rtm->rtm_scope = 0;
406     rtm->rtm_type = r->r_action;
407     rtm->rtm_flags = r->r_flags;
408
409     if (r->r_dst_len)
410         RTA_PUT(skb, RTA_DST, 4, &r->r_dst);
411     if (r->r_src_len)
412         RTA_PUT(skb, RTA_SRC, 4, &r->r_src);
413     if (r->r_ifname[0])
414         RTA_PUT(skb, RTA_IIF, IFNAMSIZ, &r->r_ifname);
415     if (r->r_preferance)
416         RTA_PUT(skb, RTA_PRIORITY, 4, &r->r_preferance);
417     if (r->r_srcmap)
418         RTA_PUT(skb, RTA_GATEWAY, 4, &r->r_srcmap);
419 #ifdef CONFIG_NET_CLS_ROUTE
420     if (r->r_tclassid)
421         RTA_PUT(skb, RTA_FLOW, 4, &r->r_tclassid);
422 #endif
423     nlh->nlmsg_len = skb->tail - b;
424     return skb->len;
425
426 nlmmsg_failure:
427 rtattr_failure:
428     skb_put(skb, b - skb->tail);
429     return -1;
430 }

```

Figure 761: FIB Rules - initialisation: linux/net/ipv4/fib_rules.c

```

431 int inet_dump_rules(struct sk_buff *skb, struct netlink_callback *cb)
432 {
433     int idx;
434     int s_idx = cb->args[0];
435     struct fib_rule *r;
436
437     read_lock(&fib_rules_lock);
438     for (r=fib_rules, idx=0; r = r->r_next, idx++) {
439         if (idx < s_idx)
440             continue;
441         if (inet_fill_rule(skb, r, cb) < 0)
442             break;
443     }
444     read_unlock(&fib_rules_lock);
445     cb->args[0] = idx;
446
447     return skb->len;
448 }
449
450 #endif /* CONFIG_RTNETLINK */
451
452 void _init fib_rules_init (void)
453 {
454     register_netdevice_notifier (& fib_rules_notifier );
455 }

```

Figure 762: FIB Rules - initialisation: `linux/net/ipv4/fib_rules.c`

```

51 static kmem_cache_t * fn_hash_kmem;
52
53 /*
54  * These bizarre types are just to force strict type checking.
55  * When I reversed order of bytes and changed to natural mask lengths,
56  * I forgot to make fixes in several places. Now I am lazy to return
57  * it back.
58  */
59
60 typedef struct {
61     u32     datum;
62 } fn_key_t;
63
64 typedef struct {
65     u32     datum;
66 } fn_hash_idx_t;
67
68 struct fib_node
69 {
70     struct fib_node     *fn_next;
71     struct fib_info     *fn_info;
72 #define FIB_INFO(f)    ((f)->fn_info)
73     fn_key_t            fn_key;
74     u8                  fn_tos;
75     u8                  fn_type;
76     u8                  fn_scope;
77     u8                  fn_state;
78 };
79
80 #define FN_S_ZOMBIE 1
81 #define FN_S_ACCESSED 2
82
83 static int fib_hash_zombies;

```

Figure 763: Fast Accessing the FIB through hashing - types in: `linux/net/ipv4/fib_hash.c`

```

85 struct fn_zone
86 {
87     struct fn_zone *fz_next;    /* Next not empty zone */
88     struct fib_node **fz_hash;  /* Hash table pointer */
89     int             fz_nent;    /* Number of entries */
90
91     int             fz_divisor; /* Hash divisor */
92     u32             fz_hashmask; /* (1<<fz_divisor) - 1 */
93 #define FZ_HASHMASK(fz) ((fz)->fz_hashmask)
94
95     int             fz_order;   /* Zone order */
96     u32             fz_mask;
97 #define FZ_MASK(fz) ((fz)->fz_mask)
98 };
99
100 /* NOTE. On fast computers evaluation of fz_hashmask and fz_mask
101     can be cheaper than memory lookup, so that FZ_* macros are used.
102 */
103
104 struct fn_hash
105 {
106     struct fn_zone *fn_zones[33];
107     struct fn_zone *fn_zone_list;
108 };

```

Figure 764: Fast Accessing the FIB through hashing - types ii:
linux/net/ipv4/fib_hash.c

```

110 static __inline__ fn_hash_idx_t fn_hash(fn_key_t key, struct fn_zone *fz)
111 {
112     u32 h = ntohl(key.datum)>>(32 - fz->fz_order);
113     h ^= (h>>20);
114     h ^= (h>>10);
115     h ^= (h>>5);
116     h &= FZ_HASHMASK(fz);
117     return *(fn_hash_idx_t*)&h;
118 }
119
120 #define fz_key_0(key)          ((key).datum = 0)
121 #define fz_prefix(key, fz)    ((key).datum)
122
123 static __inline__ fn_key_t fz_key(u32 dst, struct fn_zone *fz)
124 {
125     fn_key_t k;
126     k.datum = dst & FZ_MASK(fz);
127     return k;
128 }
129
130 static __inline__ struct fib_node ** fz_chain_p(fn_key_t key, struct fn_zone *fz)
131 {
132     return &fz->fz_hash[fn_hash(key, fz).datum];
133 }
134
135 static __inline__ struct fib_node * fz_chain(fn_key_t key, struct fn_zone *fz)
136 {
137     return fz->fz_hash[fn_hash(key, fz).datum];
138 }
139
140 extern __inline__ int fn_key_eq(fn_key_t a, fn_key_t b)
141 {
142     return a.datum == b.datum;
143 }
144
145 extern __inline__ int fn_key_leq(fn_key_t a, fn_key_t b)
146 {
147     return a.datum <= b.datum;
148 }
149
150 static rwlock_t fib_hash_lock = RW_LOCK_UNLOCKED;
151
152 #define FZ_MAX_DIVISOR 1024

```

Figure 765: Fast Accessing the FIB through hashing - inline functions i:
[linux/net/ipv4/fib_hash.c](#)

```

154 #ifdef CONFIG_IP_ROUTE_LARGE_TABLES
155
156 /* The fib hash lock must be held when this is called. */
157 static __inline__ void fn_rebuild_zone(struct fn_zone *fz,
158                                       struct fib_node **old_ht,
159                                       int old_divisor)
160 {
161     int i;
162     struct fib_node *f, **fp, *next;
163
164     for (i=0; i<old_divisor; i++) {
165         for (f=old_ht[i]; f; f=next) {
166             next = f->fn_next;
167             for (fp = fz_chain_p(f->fn_key, fz);
168                 *fp && fn_key_leq((*fp)->fn_key, f->fn_key);
169                 fp = &(*fp)->fn_next)
170                 /* NONE */;
171             f->fn_next = *fp;
172             *fp = f;
173         }
174     }
175 }

```

Figure 766: Fast Accessing the FIB through hashing - inline functions ii:
linux/net/ipv4/fib_hash.c


```

177 static void fn_rehash_zone(struct fn_zone *fz)
178 {
179     struct fib_node **ht, **old_ht;
180     int old_divisor, new_divisor;
181     u32 new_hashmask;
182
183     old_divisor = fz->fz_divisor;
184
185     switch (old_divisor) {
186     case 16:
187         new_divisor = 256;
188         new_hashmask = 0xFF;
189         break;
190     case 256:
191         new_divisor = 1024;
192         new_hashmask = 0x3FF;
193         break;
194     default:
195         printk(KERN_CRIT "route.c:bad_divisor_%d!\n", old_divisor);
196         return;
197     }
198     #if RT_CACHE_DEBUG >= 2
199     printk("fn_rehash_zone:hash_for_zone_%d_grows_from_%d\n", fz->fz_order,
200           old_divisor);
201     #endif
202     ht = kmalloc(new_divisor*sizeof(struct fib_node*), GFP_KERNEL);
203
204     if (ht) {
205         memset(ht, 0, new_divisor*sizeof(struct fib_node*));
206         write_lock_bh(&fib_hash_lock);
207         old_ht = fz->fz_hash;
208         fz->fz_hash = ht;
209         fz->fz_hashmask = new_hashmask;
210         fz->fz_divisor = new_divisor;
211         fn_rebuild_zone(fz, old_ht, old_divisor);
212         write_unlock_bh(&fib_hash_lock);
213         kfree(old_ht);
214     }
215 }
216 #endif /* CONFIG_IP_ROUTE_LARGE_TABLES */

```

Figure 767: Fast Accessing the FIB through hashing - inline functions iii:
linux/net/ipv4/fib_hash.c

```

218 static void fn_free_node(struct fib_node * f)
219 {
220     fib_release_info (FIB_INFO(f));
221     kmem_cache_free(fn_hash_kmem, f);
222 }
223
224
225 static struct fn_zone *
226 fn_new_zone(struct fn_hash *table, int z)
227 {
228     int i;
229     struct fn_zone *fz = kmalloc(sizeof(struct fn_zone), GFP_KERNEL);
230     if (!fz)
231         return NULL;
232
233     memset(fz, 0, sizeof(struct fn_zone));
234     if (z) {
235         fz->fz_divisor = 16;
236         fz->fz_hashmask = 0xF;
237     } else {
238         fz->fz_divisor = 1;
239         fz->fz_hashmask = 0;
240     }
241     fz->fz_hash = kmalloc(fz->fz_divisor*sizeof(struct fib_node*), GFP_KERNEL);
242     if (!fz->fz_hash) {
243         kfree(fz);
244         return NULL;
245     }
246     memset(fz->fz_hash, 0, fz->fz_divisor*sizeof(struct fib_node*));
247     fz->fz_order = z;
248     fz->fz_mask = inet_make_mask(z);
249
250     /* Find the first not empty zone with more specific mask */
251     for (i=z+1; i<=32; i++)
252         if (table->fn_zones[i])
253             break;
254     write_lock_bh(&fib_hash_lock);
255     if (i>32) {
256         /* No more specific masks, we are the first. */
257         fz->fz_next = table->fn_zone_list;
258         table->fn_zone_list = fz;
259     } else {
260         fz->fz_next = table->fn_zones[i]->fz_next;
261         table->fn_zones[i]->fz_next = fz;
262     }
263     table->fn_zones[z] = fz;
264     write_unlock_bh(&fib_hash_lock);
265     return fz;
266 }

```

Figure 768: Fast Accessing the FIB through hashing - zone lookup functions:
linux/net/ipv4/fib_hash.c

```

268 static int
269 fn_hash_lookup(struct fib_table *tb, const struct rt_key *key, struct fib_result *res)
270 {
271     int err;
272     struct fn_zone *fz;
273     struct fn_hash *t = (struct fn_hash*)tb->tb_data;
274
275     read_lock(&fib_hash_lock);
276     for (fz = t->fn_zone_list; fz; fz = fz->fz_next) {
277         struct fib_node *f;
278         fn_key_t k = fz_key(key->dst, fz);
279
280         for (f = fz_chain(k, fz); f; f = f->fn_next) {
281             if (!fn_key_eq(k, f->fn_key)) {
282                 if (fn_key_leq(k, f->fn_key))
283                     break;
284                 else
285                     continue;
286             }
287             #ifdef CONFIG_IP_ROUTE_TOS
288             if (f->fn_tos && f->fn_tos != key->tos)
289                 continue;
290             #endif
291             f->fn_state |= FN_S_ACCESSED;
292
293             if (f->fn_state & FN_S_ZOMBIE)
294                 continue;
295             if (f->fn_scope < key->scope)
296                 continue;
297
298             err = fib_semantic_match(f->fn_type, FIB_INFO(f), key, res);
299             if (err == 0) {
300                 res->type = f->fn_type;
301                 res->scope = f->fn_scope;
302                 res->prefixlen = fz->fz_order;
303                 goto out;
304             }
305             if (err < 0)
306                 goto out;
307         }
308     }
309     err = 1;
310 out:
311     read_unlock(&fib_hash_lock);
312     return err;
313 }

```

Figure 769: Fast Accessing the FIB through hashing - hash lookup function:
linux/net/ipv4/fib_hash.c

```

315 static int fn_hash_last_dflt = -1;
316
317 static int fib_detect_death(struct fib_info *fi, int order,
318                          struct fib_info **last_resort, int *last_idx)
319 {
320     struct neighbour *n;
321     int state = NUD_NONE;
322
323     n = neigh_lookup(&arp_tbl, &fi->fib_nh[0].nh_gw, fi->fib_dev);
324     if (n) {
325         state = n->nud_state;
326         neigh_release(n);
327     }
328     if (state == NUD_REACHABLE)
329         return 0;
330     if ((state & NUD_VALID) && order != fn_hash_last_dflt)
331         return 0;
332     if ((state & NUD_VALID) ||
333         (*last_idx < 0 && order > fn_hash_last_dflt)) {
334         *last_resort = fi;
335         *last_idx = order;
336     }
337     return 1;
338 }

```

Figure 770: Fast Accessing the FIB through hashing - dead entry:
linux/net/ipv4/fib_hash.c

```

340 static void
341 fn_hash_select_default (struct fib_table *tb, const struct rt_key *key, struct fib_result *
      res)
342 {
343     int order, last_idx;
344     struct fib_node *f;
345     struct fib_info *fi = NULL;
346     struct fib_info *last_resort;
347     struct fn_hash *t = (struct fn_hash*)tb->tb_data;
348     struct fn_zone *fz = t->fn_zones[0];
349
350     if (fz == NULL)
351         return;
352
353     last_idx = -1;
354     last_resort = NULL;
355     order = -1;
356
357     read_lock(&fib_hash_lock);
358     for (f = fz->fz_hash[0]; f; f = f->fn_next) {
359         struct fib_info *next_fi = FIB_INFO(f);
360
361         if ((f->fn_state & FN_S_ZOMBIE) ||
362             f->fn_scope != res->scope ||
363             f->fn_type != RTN_UNICAST)
364             continue;
365
366         if (next_fi->fib_priority > res->fi->fib_priority)
367             break;
368         if (!next_fi->fib_nh[0].nh_gw || next_fi->fib_nh[0].nh_scope !=
            RT_SCOPE_LINK)
369             continue;
370         f->fn_state |= FN_S_ACCESSED;
371
372         if (fi == NULL) {
373             if (next_fi != res->fi)
374                 break;
375         } else if (!fib_detect_death (fi, order, &last_resort, &last_idx)) {
376             if (res->fi)
377                 fib_info_put (res->fi);
378             res->fi = fi;
379             atomic_inc(&fi->fib_clntref);
380             fn_hash_last_dflt = order;
381             goto out;
382         }
383         fi = next_fi;
384         order++;
385     }

```

Figure 771: Fast Accessing the FIB through hashing - select default entry:
linux/net/ipv4/fib_hash.c

```

387     if (order<=0 || fi==NULL) {
388         fn_hash_last_dflt = -1;
389         goto out;
390     }
391
392     if (! fib_detect_death ( fi , order, & last_resort , & last_idx )) {
393         if (res->fi)
394             fib_info_put (res->fi);
395         res->fi = fi;
396         atomic_inc(&fi->fib_clntref);
397         fn_hash_last_dflt = order;
398         goto out;
399     }
400
401     if ( last_idx >= 0 ) {
402         if (res->fi)
403             fib_info_put (res->fi);
404         res->fi = last_resort ;
405         if ( last_resort )
406             atomic_inc(&last_resort ->fib_clntref);
407     }
408     fn_hash_last_dflt = last_idx ;
409 out:
410     read_unlock(&fib_hash_lock);
411 }

```

Figure 772: Fast Accessing the FIB through hashing - select default entry:
linux/net/ipv4/fib_hash.c

```

413 #define FIB_SCAN(f, fp) \
414 for ( ; (( f) = *(fp)) != NULL; (fp) = &(f)->fn_next)
415
416 #define FIB_SCAN_KEY(f, fp, key) \
417 for ( ; (( f) = *(fp)) != NULL && fn_key_eq((f)->fn_key, (key)); (fp) = &(f)->fn_next)
418
419 #ifndef CONFIG_IP_ROUTE_TOS
420 #define FIB_SCAN_TOS(f, fp, key, tos) FIB_SCAN_KEY(f, fp, key)
421 #else
422 #define FIB_SCAN_TOS(f, fp, key, tos) \
423 for ( ; (( f) = *(fp)) != NULL && fn_key_eq((f)->fn_key, (key)) && \
424 (f)->fn_tos == (tos); (fp) = &(f)->fn_next)
425 #endif
426
427
428 #ifdef CONFIG_RTNETLINK
429 static void rtmmsg_fib(int , struct fib_node*, int , int ,
430                      struct nlmsg_hdr *n,
431                      struct netlink_skb_parms *);
432 #else
433 #define rtmmsg_fib(a, b, c, d, e, f)
434 #endif
435
436
437 static int
438 fn_hash_insert (struct fib_table *tb, struct rtmmsg *r, struct kern_rta *rta,
439               struct nlmsg_hdr *n, struct netlink_skb_parms *req)
440 {
441     struct fn_hash *table = (struct fn_hash*)tb->tb_data;
442     struct fib_node *new_f, *f, **fp, **del_fp;
443     struct fn_zone *fz;
444     struct fib_info *fi;
445
446     int z = r->rtm_dst_len;
447     int type = r->rtm_type;
448 #ifdef CONFIG_IP_ROUTE_TOS
449     u8 tos = r->rtm_tos;
450 #endif
451     fn_key_t key;
452     int err;

```

Figure 773: Fast Accessing the FIB through hashing - insert entry i:
linux/net/ipv4/fib_hash.c

```

454 FTprint("tb(%d)_insert:_%d_%08x/%d_%d_%08x\n", tb->tb_id, r->rtm_type, rta->
      rta_dst ?
455 *(u32*)rta->rta_dst : 0, z, rta->rta_oif ? *rta->rta_oif : -1,
456 rta->rta_prefsrc ? *(u32*)rta->rta_prefsrc : 0);
457     if (z > 32)
458         return -EINVAL;
459     fz = table->fn_zones[z];
460     if (!fz && !(fz = fn_new_zone(table, z)))
461         return -ENOBUFS;
462
463     fz_key_0(key);
464     if (rta->rta_dst) {
465         u32 dst;
466         memcpy(&dst, rta->rta_dst, 4);
467         if (dst & ~FZ_MASK(fz))
468             return -EINVAL;
469         key = fz_key(dst, fz);
470     }
471
472     if ((fi = fib_create_info(r, rta, n, &err)) == NULL)
473         return err;
474
475     #ifdef CONFIG_IP_ROUTE_LARGE_TABLES
476     if (fz->fn_nent > (fz->fn_divisor << 2) &&
477         fz->fn_divisor < FZ_MAX_DIVISOR &&
478         (z == 32 || (1 << z) > fz->fn_divisor))
479         fn_rehash_zone(fz);
480     #endif
481
482     fp = fz_chain_p(key, fz);
483
484
485     /*
486     * Scan list to find the first route with the same destination
487     */
488     FIB_SCAN(f, fp) {
489         if (fn_key_leq(key, f->fn_key))
490             break;
491     }
492
493     #ifdef CONFIG_IP_ROUTE_TOS
494     /*
495     * Find route with the same destination and tos.
496     */
497     FIB_SCAN_KEY(f, fp, key) {
498         if (f->fn_tos <= tos)
499             break;
500     }
501     #endif

```

Figure 774: Fast Accessing the FIB through hashing - insert entry ii:
linux/net/ipv4/fib_hash.c


```

503     del_fp = NULL;
504
505     if (f && (f->fn_state&FN_S_ZOMBIE) &&
506 #ifdef CONFIG_IP_ROUTE_TOS
507     f->fn_tos == tos &&
508 #endif
509     fn_key_eq(f->fn_key, key)) {
510         del_fp = fp;
511         fp = &f->fn_next;
512         f = *fp;
513         goto create;
514     }
515
516     FIB_SCAN_TOS(f, fp, key, tos) {
517         if (f->fib_priority <= FIB_INFO(f)->fib_priority)
518             break;
519     }

```

Figure 775: Fast Accessing the FIB through hashing - insert entry iii:
linux/net/ipv4/fib_hash.c

```

521     /* Now f==*fp points to the first node with the same
522     keys [prefix,tos,priority], if such key already
523     exists or to the node, before which we will insert new one.
524     */
525
526     if (f &&
527 #ifdef CONFIG_IP_ROUTE_TOS
528         f->fn_tos == tos &&
529 #endif
530         fn_key_eq(f->fn_key, key) &&
531         fi->fib_priority == FIB_INFO(f->fib_priority) {
532         struct fib_node **ins_fp;
533
534         err = -EEXIST;
535         if (n->nmsg_flags&NLM_F_EXCL)
536             goto out;
537
538         if (n->nmsg_flags&NLM_F_REPLACE) {
539             del_fp = fp;
540             fp = &f->fn_next;
541             f = *fp;
542             goto replace;
543         }
544
545         ins_fp = fp;
546         err = -EEXIST;
547
548         FIB_SCAN_TOS(f, fp, key, tos) {
549             if (fi->fib_priority != FIB_INFO(f->fib_priority)
550                 break;
551             if (f->fn_type == type && f->fn_scope == r->rtm_scope
552                 && FIB_INFO(f) == fi)
553                 goto out;
554         }
555
556         if (!(n->nmsg_flags&NLM_F_APPEND)) {
557             fp = ins_fp;
558             f = *fp;
559         }
560     }

```

Figure 776: Fast Accessing the FIB through hashing - insert entry iv:
linux/net/ipv4/fib_hash.c

```

562 create:
563     err = -ENOENT;
564     if (!(n->nmsg_flags&NLM_F_CREATE))
565         goto out;
566
567 replace:
568     err = -ENOBUFFS;
569     new_f = kmem_cache_alloc(fn_hash_kmem, SLAB_KERNEL);
570     if (new_f == NULL)
571         goto out;
572
573     memset(new_f, 0, sizeof(struct fib_node));
574
575     new_f->fn_key = key;
576 #ifdef CONFIG_IP_ROUTE_TOS
577     new_f->fn_tos = tos;
578 #endif
579     new_f->fn_type = type;
580     new_f->fn_scope = r->rtm_scope;
581     FIB_INFO(new_f) = fi;
582
583     /*
584      * Insert new entry to the list.
585      */
586
587     new_f->fn_next = f;
588     write_lock_bh(&fib_hash_lock);
589     *fp = new_f;
590     write_unlock_bh(&fib_hash_lock);
591     fz->fz_nent++;

```

Figure 777: Fast Accessing the FIB through hashing - insert entry v:
linux/net/ipv4/fib_hash.c

```

593     if (del_fp) {
594         f = *del_fp;
595         /* Unlink replaced node */
596         write_lock_bh(&fib_hash_lock);
597         *del_fp = f->fn_next;
598         write_unlock_bh(&fib_hash_lock);
599
600         if (!(f->fn_state&FN_S_ZOMBIE))
601             rtmsg_fib(RTM_DELROUTE, f, z, tb->tb_id, n, req);
602         if (f->fn_state&FN_S_ACCESSED)
603             rt_cache_flush(-1);
604         fn_free_node(f);
605         fz->fz_nent--;
606     } else {
607         rt_cache_flush(-1);
608     }
609     rtmsg_fib(RTM_NEWROUTE, new_f, z, tb->tb_id, n, req);
610     return 0;
611
612 out:
613     fib_release_info(fi);
614     return err;
615 }

```

Figure 778: Fast Accessing the FIB through hashing - insert entry v:
[linux/net/ipv4/fib_hash.c](#)

```

617
618 static int
619 fn_hash_delete(struct fib_table *tb, struct rtmmsg *r, struct kern_rta *rta,
620               struct nlmsg_hdr *n, struct netlink_skb_parms *req)
621 {
622     struct fn_hash *table = (struct fn_hash*)tb->tb_data;
623     struct fib_node **fp, **del_fp, *f;
624     int z = r->rtm_dst_len;
625     struct fn_zone *fz;
626     fn_key_t key;
627     int matched;
628 #ifdef CONFIG_IP_ROUTE_TOS
629     u8 tos = r->rtm_tos;
630 #endif
631
632 FTprint("tb(%d)_delete:_%d_%08x/%d_%d\n", tb->tb_id, r->rtm_type, rta->rta_dst ?
633         *(u32*)rta->rta_dst : 0, z, rta->rta_oif ? *rta->rta_oif : -1);
634     if (z > 32)
635         return -EINVAL;
636     if ((fz = table->fn_zones[z]) == NULL)
637         return -ESRCH;
638
639     fz_key_0(key);
640     if (rta->rta_dst) {
641         u32 dst;
642         memcpy(&dst, rta->rta_dst, 4);
643         if (dst & ~FZ_MASK(fz))
644             return -EINVAL;
645         key = fz_key(dst, fz);
646     }
647
648     fp = fz_chain_p(key, fz);
649
650
651     FIB_SCAN(f, fp) {
652         if (fn_key_eq(f->fn_key, key))
653             break;
654         if (fn_key_leq(key, f->fn_key)) {
655             return -ESRCH;
656         }
657     }

```

Figure 779: Fast Accessing the FIB through hashing - delete entry i:
linux/net/ipv4/fib_hash.c

```

658 #ifdef CONFIG_IP_ROUTE_TOS
659     FIB_SCAN_KEY(f, fp, key) {
660         if (f->fn_tos == tos)
661             break;
662     }
663 #endif
664
665     matched = 0;
666     del_fp = NULL;
667     FIB_SCAN_TOS(f, fp, key, tos) {
668         struct fib_info * fi = FIB_INFO(f);
669
670         if (f->fn_state & FN_S_ZOMBIE) {
671             return -ESRCH;
672         }
673         matched++;
674
675         if (del_fp == NULL &&
676             (!r->rtm_type || f->fn_type == r->rtm_type) &&
677             (r->rtm_scope == RT_SCOPE_NOWHERE || f->fn_scope == r->
678              rtm_scope) &&
679             (!r->rtm_protocol || fi->fib_protocol == r->rtm_protocol) &&
680             fib_nh_match(r, n, rta, fi) == 0)
681             del_fp = fp;
682     }

```

Figure 780: Fast Accessing the FIB through hashing - delete entry ii:
linux/net/ipv4/fib_hash.c

```

683     if (del_fp) {
684         f = *del_fp;
685         rtmsg_fib(RTM_DELROUTE, f, z, tb->tb_id, n, req);
686
687         if (matched != 1) {
688             write_lock_bh(&fib_hash_lock);
689             *del_fp = f->fn_next;
690             write_unlock_bh(&fib_hash_lock);
691
692             if (f->fn_state & FN_S_ACCESSED)
693                 rt_cache_flush(-1);
694             fn_free_node(f);
695             fz->fz_nent--;
696         } else {
697             f->fn_state |= FN_S_ZOMBIE;
698             if (f->fn_state & FN_S_ACCESSED) {
699                 f->fn_state &= ~FN_S_ACCESSED;
700                 rt_cache_flush(-1);
701             }
702             if (++fib_hash_zombies > 128)
703                 fib_flush();
704         }
705
706         return 0;
707     }
708     return -ESRCH;
709 }

```

Figure 781: Fast Accessing the FIB through hashing - delete entry iii:
linux/net/ipv4/fib_hash.c

```

711 extern __inline__ int
712 fn_flush_list (struct fib_node ** fp, int z, struct fn_hash *table)
713 {
714     int found = 0;
715     struct fib_node *f;
716
717     while ((f = *fp) != NULL) {
718         struct fib_info *fi = FIB_INFO(f);
719
720         if ( fi && ((f->fn_state&FN_S_ZOMBIE) || (fi->fib_flags&
721             RTNH_F_DEAD))) {
722             write_lock_bh(&fib_hash_lock);
723             *fp = f->fn_next;
724             write_unlock_bh(&fib_hash_lock);
725
726             fn_free_node(f);
727             found++;
728             continue;
729         }
730         fp = &f->fn_next;
731     }
732     return found;
733 }
734
735 static int fn_hash_flush(struct fib_table *tb)
736 {
737     struct fn_hash *table = (struct fn_hash*)tb->tb_data;
738     struct fn_zone *fz;
739     int found = 0;
740
741     fib_hash_zombies = 0;
742     for (fz = table->fn_zone_list; fz; fz = fz->fz_next) {
743         int i;
744         int tmp = 0;
745         for (i=fz->fz_divisor-1; i>=0; i--)
746             tmp += fn_flush_list(&fz->fz_hash[i], fz->fz_order, table);
747         fz->fz_nent -= tmp;
748         found += tmp;
749     }
750     return found;
751 }

```

Figure 782: Fast Accessing the FIB through hashing - flush entries:
linux/net/ipv4/fib_hash.c


```

755 static int fn_hash_get_info(struct fib_table *tb, char *buffer, int first, int count)
756 {
757     struct fn_hash *table = (struct fn_hash*)tb->tb_data;
758     struct fn_zone *fz;
759     int pos = 0;
760     int n = 0;
761
762     read_lock(&fib_hash_lock);
763     for (fz=table->fn_zone_list; fz; fz = fz->fz_next) {
764         int i;
765         struct fib_node *f;
766         int maxslot = fz->fz_divisor;
767         struct fib_node **fp = fz->fz_hash;
768
769         if (fz->fz_nent == 0)
770             continue;
771
772         if (pos + fz->fz_nent <= first) {
773             pos += fz->fz_nent;
774             continue;
775         }
776
777         for (i=0; i < maxslot; i++, fp++) {
778             for (f = *fp; f; f = f->fn_next) {
779                 if (++pos <= first)
780                     continue;
781                 fib_node_get_info (f->fn_type,
782                                     f->fn_state&FN_S_ZOMBIE,
783                                     FIB_INFO(f),
784                                     fz_prefix (f->fn_key, fz),
785                                     FZ_MASK(fz), buffer);
786
787                 buffer += 128;
788                 if (++n >= count)
789                     goto out;
790             }
791         }
792     }
793 out:
794     read_unlock(&fib_hash_lock);
795     return n;
796 }

```

Figure 783: Accessing the FIB -Proc FS access : linux/net/ipv4/fib_hash.c

```

801 extern __inline__ int
802 fn_hash_dump_bucket(struct sk_buff *skb, struct netlink_callback *cb,
803                   struct fib_table *tb,
804                   struct fn_zone *fz,
805                   struct fib_node *f)
806 {
807     int i, s_i;
808
809     s_i = cb->args[3];
810     for (i=0; f; i++, f=f->fn_next) {
811         if (i < s_i) continue;
812         if (f->fn_state & FN_S_ZOMBIE) continue;
813         if (fib_dump_info(skb, NETLINK_CB(cb->skb).pid, cb->nlh->
814                             nlmsg_seq,
815                             RTM_NEWROUTE,
816                             tb->tb_id, (f->fn_state & FN_S_ZOMBIE) ? 0 : f->
817                                 fn_type, f->fn_scope,
818                                 &f->fn_key, fz->fz_order, f->fn_tos,
819                                 f->fn_info) < 0) {
820             cb->args[3] = i;
821             return -1;
822         }
823     }
824     cb->args[3] = i;
825     return skb->len;
826 }

```

Figure 784: Accessing the FIB - RT access i: `linux/net/ipv4/fib_hash.c`

```

826 extern __inline__ int
827 fn_hash_dump_zone(struct sk_buff *skb, struct netlink_callback *cb,
828                  struct fib_table *tb,
829                  struct fn_zone *fz)
830 {
831     int h, s_h;
832
833     s_h = cb->args[2];
834     for (h=0; h < fz->fz_divisor; h++) {
835         if (h < s_h) continue;
836         if (h > s_h)
837             memset(&cb->args[3], 0, sizeof(cb->args) - 3*sizeof(cb->args
838                [0]));
839         if (fz->fz_hash == NULL || fz->fz_hash[h] == NULL)
840             continue;
841         if (fn_hash_dump_bucket(skb, cb, tb, fz, fz->fz_hash[h]) < 0) {
842             cb->args[2] = h;
843             return -1;
844         }
845     }
846     cb->args[2] = h;
847     return skb->len;
848 }

```

Figure 785: Accessing the FIB - RT access ii: `linux/net/ipv4/fib_hash.c`

```

849 static int fn_hash_dump(struct fib_table *tb, struct sk_buff *skb, struct netlink_callback
      *cb)
850 {
851     int m, s_m;
852     struct fn_zone *fz;
853     struct fn_hash *table = (struct fn_hash*)tb->tb_data;
854
855     s_m = cb->args[1];
856     read_lock(&fib_hash_lock);
857     for (fz = table->fn_zone_list, m=0; fz; fz = fz->fz_next, m++) {
858         if (m < s_m) continue;
859         if (m > s_m)
860             memset(&cb->args[2], 0, sizeof(cb->args) - 2*sizeof(cb->args
      [0]));
861         if (fn_hash_dump_zone(skb, cb, tb, fz) < 0) {
862             cb->args[1] = m;
863             read_unlock(&fib_hash_lock);
864             return -1;
865         }
866     }
867     read_unlock(&fib_hash_lock);
868     cb->args[1] = m;
869     return skb->len;
870 }

```

Figure 786: Accessing the FIB - RT access iii: `linux/net/ipv4/fib_hash.c`

```

872 static void rtmsg_fib(int event, struct fib_node* f, int z, int tb_id,
873                      struct nlmsg_hdr *n, struct netlink_skb_parms *req)
874 {
875     struct sk_buff *skb;
876     u32 pid = req ? req->pid : 0;
877     int size = NLMSG_SPACE(sizeof(struct rtmsg)+256);
878
879     skb = alloc_skb(size, GFP_KERNEL);
880     if (!skb)
881         return;
882
883     if (fib_dump_info(skb, pid, n->nmsg_seq, event, tb_id,
884                     f->fn_type, f->fn_scope, &f->fn_key, z, f->fn_tos,
885                     FIB_INFO(f)) < 0) {
886         kfree_skb(skb);
887         return;
888     }
889     NETLINK_CB(skb).dst_groups = RTMGRP_IPV4_ROUTE;
890     if (n->nmsg_flags & NLM_F_ECHO)
891         atomic_inc(&skb->users);
892     netlink_broadcast(rtnl, skb, pid, RTMGRP_IPV4_ROUTE, GFP_KERNEL);
893     if (n->nmsg_flags & NLM_F_ECHO)
894         netlink_unicast(rtnl, skb, pid, MSG_DONTWAIT);
895 }

```

Figure 787: Accessing the FIB - RT access iv: `linux/net/ipv4/fib_hash.c`

```

899 #ifdef CONFIG_IP_MULTIPLE_TABLES
900 struct fib_table * fib_hash_init (int id)
901 #else
902 struct fib_table * _init fib_hash_init (int id)
903 #endif
904 {
905     struct fib_table *tb;
906
907     if (fn_hash_kmem == NULL)
908         fn_hash_kmem = kmem_cache_create("ip_fib_hash",
909                                         sizeof(struct fib_node),
910                                         0, SLAB_HWCACHE_ALIGN,
911                                         NULL, NULL);
912
913     tb = kmalloc(sizeof(struct fib_table) + sizeof(struct fn_hash), GFP_KERNEL);
914     if (tb == NULL)
915         return NULL;
916
917     tb->tb_id = id;
918     tb->tb_lookup = fn_hash_lookup;
919     tb->tb_insert = fn_hash_insert;
920     tb->tb_delete = fn_hash_delete;
921     tb->tb_flush = fn_hash_flush;
922     tb->tb_select_default = fn_hash_select_default ;
923 #ifdef CONFIG_RTNETLINK
924     tb->tb_dump = fn_hash_dump;
925 #endif
926 #ifdef CONFIG_PROC_FS
927     tb->tb_get_info = fn_hash_get_info;
928 #endif
929     memset(tb->tb_data, 0, sizeof(struct fn_hash));
930     return tb;
931 }

```

Figure 788: Accessing the FIB - Initialisation: `linux/net/ipv4/fib_hash.c`

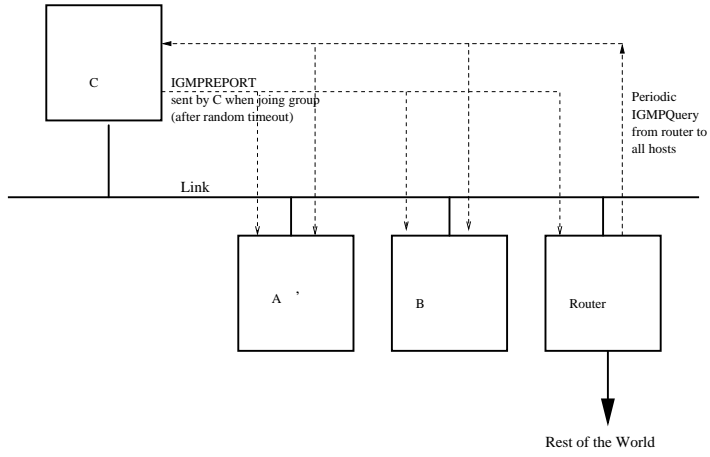


Figure 789: igmp

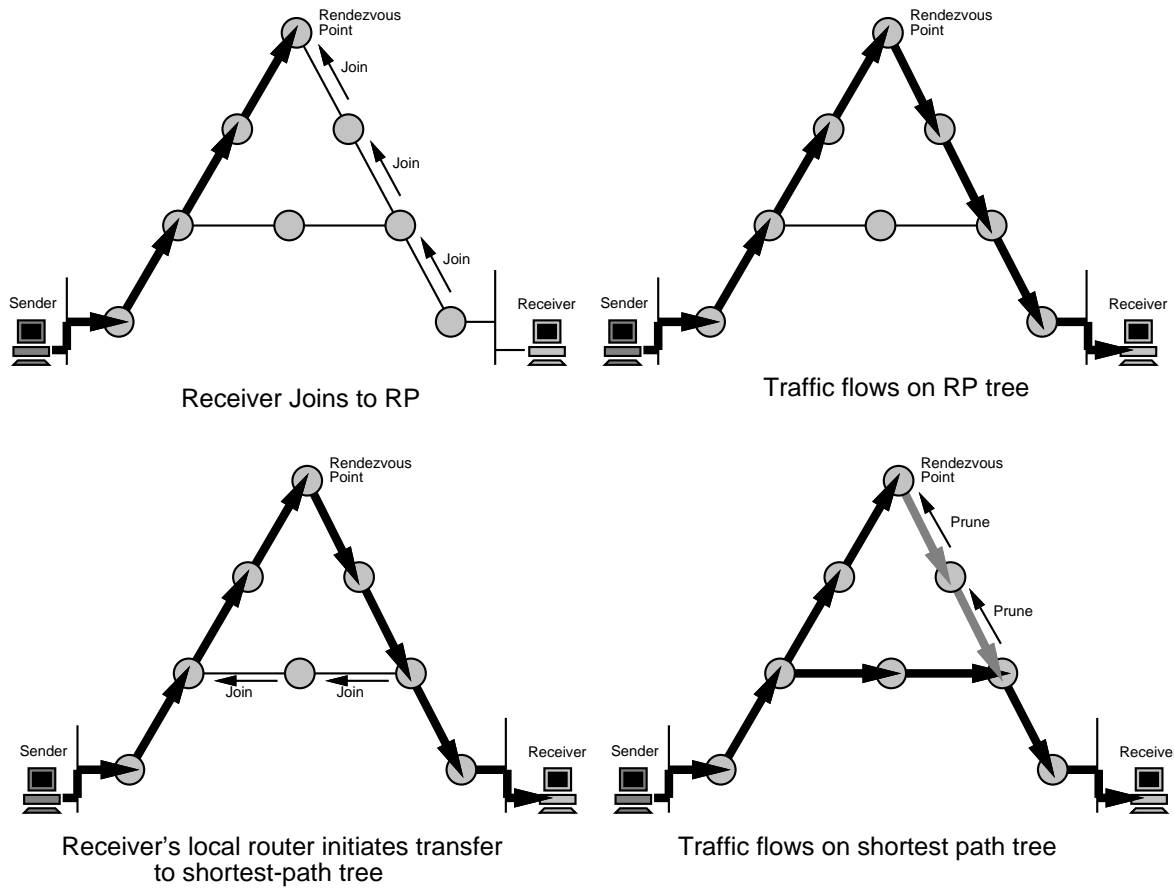


Figure 790: PIM SM


```
36
37 #define RTO_ONLINK 0x01
38 #define RTO_TPROXY 0x80000000
39
40 #define RTO_CONN 0
41
42 struct rt_key
43 {
44     _u32          dst;
45     _u32          src;
46     int          iif;
47     int          oif;
48 #ifdef CONFIG_IP_ROUTE_FWMARK
49     _u32          fwmark;
50 #endif
51     _u8          tos;
52     _u8          scope;
53 };
54
55 struct inet_peer;
```

Figure 791: The Route Interface i: `linux/include/net/route.h`

```

56 struct rtable
57 {
58     union
59     {
60         struct dst_entry    dst;
61         struct rtable      *rt_next;
62     } u;
63
64     unsigned                rt_flags ;
65     unsigned                rt_type;
66
67     _u32                    rt_dst ; /* Path destination */
68     _u32                    rt_src ; /* Path source      */
69     int                     rt_iif ;
70
71     /* Info on neighbour */
72     _u32                    rt_gateway;
73
74     /* Cache lookup keys */
75     struct rt_key           key;
76
77     /* Miscellaneous cached information */
78     _u32                    rt_spec_dst ; /* RFC1122 specific destination */
79     struct inet_peer        *peer; /* long-living peer info */
80
81 #ifdef CONFIG_IP_ROUTE_NAT
82     _u32                    rt_src_map;
83     _u32                    rt_dst_map;
84 #endif
85 };

```

Figure 792: The Route Interface ii: linux/include/net/route.h

```

87 struct ip_rt_acct
88 {
89     __u32   o_bytes;
90     __u32   o_packets;
91     __u32   i_bytes;
92     __u32   i_packets;
93 };
94
95 extern struct ip_rt_acct *ip_rt_acct;
96
97 struct in_device;
98 extern void          ip_rt_init (void);
99 extern void          ip_rt_redirect (u32 old_gw, u32 dst, u32 new_gw,
100                                     u32 src, u8 tos, struct net_device *dev);
101 extern void          ip_rt_advice (struct rtable **rp, int advice);
102 extern void          rt_cache_flush (int how);
103 extern int           ip_route_output_key (struct rtable **, const struct rt_key *key);
104 extern int           ip_route_input (struct sk_buff *, u32 dst, u32 src, u8 tos, struct
    net_device *devin);
105 extern unsigned short ip_rt_frag_needed (struct iphdr *iph, unsigned short new_mtu);
106 extern void          ip_rt_update_pmtu (struct dst_entry *dst, unsigned mtu);
107 extern void          ip_rt_send_redirect (struct sk_buff *skb);
108
109 extern unsigned      inet_addr_type (u32 addr);
110 extern void          ip_rt_multicast_event (struct in_device *);
111 extern int           ip_rt_ioctl (unsigned int cmd, void *arg);
112 extern void          ip_rt_get_source (u8 *src, struct rtable *rt);
113 extern int           ip_rt_dump (struct sk_buff *skb, struct netlink_callback *cb);
114
115 /* Deprecated: use ip_route_output_key directly */

```

Figure 793: The Route Interface iii: `linux/include/net/route.h`

```

100 #define IP_MAX_MTU 0xFFFF0
101
102 #define RT_GC_TIMEOUT (300*HZ)
103
104 int ip_rt_min_delay = 2*HZ;
105 int ip_rt_max_delay = 10*HZ;
106 int ip_rt_max_size;
107 int ip_rt_gc_timeout = RT_GC_TIMEOUT;
108 int ip_rt_gc_interval = 60*HZ;
109 int ip_rt_gc_min_interval = 5*HZ;
110 int ip_rt_redirect_number = 9;
111 int ip_rt_redirect_load = HZ/50;
112 int ip_rt_redirect_silence = ((HZ/50) << (9+1));
113 int ip_rt_error_cost = HZ;
114 int ip_rt_error_burst = 5*HZ;
115 int ip_rt_gc_elasticity = 8;
116 int ip_rt_mtu_expires = 10*60*HZ;
117 int ip_rt_min_pmtu = 512+20+20;
118 int ip_rt_min_advms = 256;
119
120 static unsigned long rt_deadline;
121
122 #define RTprint(a...) printk(KERN_DEBUG a)
123
124 static struct timer_list rt_flush_timer;
125 static struct timer_list rt_periodic_timer;
126
127 /*
128  * Interface to generic destination cache.
129  */
130
131 static struct dst_entry * ipv4_dst_check(struct dst_entry * dst, u32);
132 static struct dst_entry * ipv4_dst_reroute(struct dst_entry * dst,
133                                           struct sk_buff *);
134 static void ipv4_dst_destroy(struct dst_entry * dst);
135 static struct dst_entry * ipv4_negative_advice(struct dst_entry *);
136 static void ipv4_link_failure(struct sk_buff *skb);
137 static int rt_garbage_collect(void);

```

Figure 794: Basic Route Maintenance: linux/net/ipv4/route.c

```

140 struct dst_ops ipv4_dst_ops =
141 {
142     AF_INET,
143     _constant_htons(ETH_P_IP),
144     0,
145
146     rt_garbage_collect ,
147     ipv4_dst_check ,
148     ipv4_dst_reroute ,
149     ipv4_dst_destroy ,
150     ipv4_negative_advice ,
151     ipv4_link_failure ,
152     sizeof (struct rtable),
153 };
154
155 #ifdef CONFIG_INET_ECN
156 #define ECN_OR_COST(class) TC_PRIO_##class
157 #else
158 #define ECN_OR_COST(class) TC_PRIO_FILLER
159 #endif
160
161 _u8 ip_tos2prio [16] = {
162     TC_PRIO_BESTEFFORT,
163     ECN_OR_COST(FILLER),
164     TC_PRIO_BESTEFFORT,
165     ECN_OR_COST(BESTEFFORT),
166     TC_PRIO_BULK,
167     ECN_OR_COST(BULK),
168     TC_PRIO_BULK,
169     ECN_OR_COST(BULK),
170     TC_PRIO_INTERACTIVE,
171     ECN_OR_COST(INTERACTIVE),
172     TC_PRIO_INTERACTIVE,
173     ECN_OR_COST(INTERACTIVE),
174     TC_PRIO_INTERACTIVE_BULK,
175     ECN_OR_COST(INTERACTIVE_BULK),
176     TC_PRIO_INTERACTIVE_BULK,
177     ECN_OR_COST(INTERACTIVE_BULK)
178 };

```

Figure 795: Basic Route Maintenance Ops: `linux/net/ipv4/route.c`

```
53 DECLARE_MUTEX(rtnl_sem);
54
55 void rtnl_lock(void)
56 {
57     rtnl_shlock();
58     rtnl_exlock();
59 }
60
61 void rtnl_unlock(void)
62 {
63     rtnl_exunlock();
64     rtnl_shunlock();
65 }
```

Figure 796: API for Route Maintenance: `linux/net/core/rtnetlink.c`

```
42 struct rt_key
43 {
44     _u32          dst;
45     _u32          src;
46     int           iif;
47     int           oif;
48 #ifdef CONFIG_IP_ROUTE_FWMARK
49     _u32          fwmark;
50 #endif
51     _u8           tos;
52     _u8           scope;
53 };
```

Figure 797: The Route API Hash Key: `linux/include/net/route.h`

```

56 struct rtable
57 {
58     union
59     {
60         struct dst_entry    dst;
61         struct rtable      *rt_next;
62     } u;
63
64     unsigned    rt_flags ;
65     unsigned    rt_type;
66
67     _u32        rt_dst ; /* Path destination */
68     _u32        rt_src ; /* Path source      */
69     int         rt_iif ;
70
71     /* Info on neighbour */
72     _u32        rt_gateway;
73
74     /* Cache lookup keys */
75     struct rt_key    key;
76
77     /* Miscellaneous cached information */
78     _u32        rt_spec_dst ; /* RFC1122 specific destination */
79     struct inet_peer    *peer; /* long-living peer info */
80
81 #ifdef CONFIG_IP_ROUTE_NAT
82     _u32        rt_src_map;
83     _u32        rt_dst_map;
84 #endif
85 };

```

Figure 798: The Route API Table: `linux/include/net/route.h`


```

1893 int ip_route_output_key(struct rtable **rp, const struct rt_key *key)
1894 {
1895     unsigned hash;
1896     struct rtable *rth;
1897
1898     hash = rt_hash_code(key->dst, key->src^(key->oif<<5), key->tos);
1899
1900     read_lock_bh(&rt_hash_table[hash].lock);
1901     for (rth=rt_hash_table[hash].chain; rth; rth=rth->u.rt_next) {
1902         if (rth->key.dst == key->dst &&
1903             rth->key.src == key->src &&
1904             rth->key.iif == 0 &&
1905             rth->key.oif == key->oif &&
1906             #ifdef CONFIG_IP_ROUTE_FWMARK
1907                 rth->key.fwmark == key->fwmark &&
1908             #endif
1909             !((rth->key.tos^key->tos)&(IPTOS_RT_MASK|RTO_ONLINK)) &&
1910             ((key->tos&RTO_TPROXY) || !(rth->rt_flags&RTCF_TPROXY))
1911         ) {
1912             rth->u.dst.lastuse = jiffies ;
1913             dst_hold(&rth->u.dst);
1914             rth->u.dst._use++;
1915             read_unlock_bh(&rt_hash_table[hash].lock);
1916             *rp = rth;
1917             return 0;
1918         }
1919     }
1920     read_unlock_bh(&rt_hash_table[hash].lock);
1921
1922     return ip_route_output_slow(rp, key);
1923 }

```

Figure 799: Route Output Lookup: `linux/net/ipv4/route.c`

```

1613 int ip_route_output_slow(struct rtable **rp, const struct rt_key *oldkey)
1614 {
1615     struct rt_key key;
1616     struct fib_result res;
1617     unsigned flags = 0;
1618     struct rtable *rth;
1619     struct net_device *dev_out = NULL;
1620     unsigned hash;
1621     int free_res = 0;
1622     int err;
1623     u32 tos;
1624
1625     tos = oldkey->tos & (IPTOS_RT_MASK|RTO_ONLINK);
1626     key.dst = oldkey->dst;
1627     key.src = oldkey->src;
1628     key.tos = tos&IPTOS_RT_MASK;
1629     key.iif = loopback_dev.ifindex;
1630     key.oif = oldkey->oif;
1631 #ifdef CONFIG_IP_ROUTE_FWMARK
1632     key.fwmark = oldkey->fwmark;
1633 #endif
1634     key.scope = (tos&RTO_ONLINK) ? RT_SCOPE_LINK : RT_SCOPE_UNIVERSE;
1635     res.fi = NULL;
1636 #ifdef CONFIG_IP_MULTIPLE_TABLES
1637     res.r = NULL;
1638 #endif

```

Figure 800: Route Lookup i : linux/net/ipv4/route.c

```

1640     if (oldkey->src) {
1641         if (MULTICAST(oldkey->src)
1642             || BADCLASS(oldkey->src)
1643             || ZERONET(oldkey->src))
1644             return -EINVAL;
1645
1646         /* It is equivalent to inet_addr_type(saddr) == RTN_LOCAL */
1647         dev_out = ip_dev_find(oldkey->src);
1648         if (dev_out == NULL)
1649             return -EINVAL;
1650
1651         /* I removed check for oif == dev_out->oif here.
1652            It was wrong by three reasons:
1653            1. ip_dev_find(saddr) can return wrong iface, if saddr is
1654               assigned to multiple interfaces.
1655            2. Moreover, we are allowed to send packets with saddr
1656               of another iface. --ANK
1657         */
1658
1659         if (oldkey->oif == 0
1660             && (MULTICAST(oldkey->dst) || oldkey->dst == 0xFFFFFFFF))
1661             {
1662                 /* Special hack: user can direct multicasts
1663                    and limited broadcast via necessary interface
1664                    without fiddling with IP_MULTICAST_IF or IP_PKTINFO.
1665                    This hack is not just for fun, it allows
1666                    vic,vat and friends to work.
1667                    They bind socket to loopback, set ttl to zero
1668                    and expect that it will work.
1669                    From the viewpoint of routing cache they are broken,
1670                    because we are not allowed to build multicast path
1671                    with loopback source addr (look, routing cache
1672                    cannot know, that ttl is zero, so that packet
1673                    will not leave this host and route is valid).
1674                    Luckily, this hack is good workaround.
1675                 */
1676                 key.oif = dev_out->ifindex;
1677                 goto make_route;
1678             }
1679         if (dev_out)
1680             dev_put(dev_out);
1681         dev_out = NULL;
1682     }

```

Figure 801: Route Lookup ii - source case: linux/net/ipv4/route.c

```

1683     if (oldkey->oif) {
1684         dev_out = dev_get_by_index(oldkey->oif);
1685         if (dev_out == NULL)
1686             return -ENODEV;
1687         if (_in_dev_get(dev_out) == NULL) {
1688             dev_put(dev_out);
1689             return -ENODEV; /* Wrong error code */
1690         }
1691
1692         if (LOCAL_MCAST(oldkey->dst) || oldkey->dst == 0xFFFFFFFF) {
1693             if (!key.src)
1694                 key.src = inet_select_addr(dev_out, 0, RT_SCOPE_LINK)
1695                 ;
1696             goto make_route;
1697         }
1698         if (!key.src) {
1699             if (MULTICAST(oldkey->dst))
1700                 key.src = inet_select_addr(dev_out, 0, key.scope);
1701             else if (!oldkey->dst)
1702                 key.src = inet_select_addr(dev_out, 0, RT_SCOPE_HOST
1703                 );
1704         }
1705     }

```

Figure 802: Route Lookup iii - oif case: `linux/net/ipv4/route.c`

```
1705     if (!key.dst) {
1706         key.dst = key.src;
1707         if (!key.dst)
1708             key.dst = key.src = htonl(INADDR_LOOPBACK);
1709         if (dev_out)
1710             dev_put(dev_out);
1711         dev_out = &loopback_dev;
1712         dev_hold(dev_out);
1713         key.oif = loopback_dev.ifindex;
1714         res.type = RTN_LOCAL;
1715         flags |= RTCF_LOCAL;
1716         goto make_route;
1717     }
```

Figure 803: Route Lookup - loopback case: `linux/net/ipv4/route.c`

```

1719     if (fib_lookup(&key, &res)) {
1720         res.fi = NULL;
1721         if (oldkey->oif) {
1722             /* Apparently, routing tables are wrong. Assume,
1723              that the destination is on link.
1724
1725              WHY? DW.
1726              Because we are allowed to send to iface
1727              even if it has NO routes and NO assigned
1728              addresses. When oif is specified, routing
1729              tables are looked up with only one purpose:
1730              to catch if destination is gatewayed, rather than
1731              direct. Moreover, if MSG_DONTROUTE is set,
1732              we send packet, ignoring both routing tables
1733              and ifaddr state. --ANK
1734
1735
1736              We could make it even if oif is unknown,
1737              likely IPv6, but we do not.
1738              */
1739
1740             if (key.src == 0)
1741                 key.src = inet_select_addr(dev_out, 0, RT_SCOPE_LINK)
1742                 ;
1743             res.type = RTN_UNICAST;
1744             goto make_route;
1745         }
1746         if (dev_out)
1747             dev_put(dev_out);
1748         return -ENETUNREACH;
1749     }

```

Figure 804: Route Lookup: oh, need a full FIB
lookuplinux/net/ipv4/route.c

```

1749     free_res = 1;
1750
1751     if (res.type == RTN_NAT)
1752         goto e_inval;
1753
1754     if (res.type == RTN_LOCAL) {
1755         if (!key.src)
1756             key.src = key.dst;
1757         if (dev_out)
1758             dev_put(dev_out);
1759         dev_out = &loopback_dev;
1760         dev_hold(dev_out);
1761         key.oif = dev_out->ifindex;
1762         if (res.fi)
1763             fib_info_put(res.fi);
1764         res.fi = NULL;
1765         flags |= RTCF_LOCAL;
1766         goto make_route;
1767     }
1768
1769     #ifdef CONFIG_IP_ROUTE_MULTIPATH
1770     if (res.fi->fib_nhs > 1 && key.oif == 0)
1771         fib_select_multipath(&key, &res);
1772     else
1773     #endif
1774     if (res.prefixlen == 0 && res.type == RTN_UNICAST && key.oif == 0)
1775         fib_select_default(&key, &res);
1776
1777     if (!key.src)
1778         key.src = FIB_RES_PREFSRC(res);
1779
1780     if (dev_out)
1781         dev_put(dev_out);
1782     dev_out = FIB_RES_DEV(res);
1783     dev_hold(dev_out);
1784     key.oif = dev_out->ifindex;

```

Figure 805: Route Lookup: corner cases `linux/net/ipv4/route.c`

```

1786 make_route:
1787     if (LOOPBACK(key.src) && !(dev_out->flags&IFF_LOOPBACK))
1788         goto e_inval;
1789
1790     if (key.dst == 0xFFFFFFFF)
1791         res.type = RTN_BROADCAST;
1792     else if (MULTICAST(key.dst))
1793         res.type = RTN_MULTICAST;
1794     else if (BADCLASS(key.dst) || ZERONET(key.dst))
1795         goto e_inval;
1796
1797     if (dev_out->flags&IFF_LOOPBACK)
1798         flags |= RTCF_LOCAL;
1799
1800     if (res.type == RTN_BROADCAST) {
1801         flags |= RTCF_BROADCAST|RTCF_LOCAL;
1802         if (res.fi) {
1803             fib_info_put (res.fi);
1804             res.fi = NULL;
1805         }
1806     } else if (res.type == RTN_MULTICAST) {
1807         flags |= RTCF_MULTICAST|RTCF_LOCAL;
1808         read_lock(&inetdev_lock);
1809         if (!_in_dev_get (dev_out) || ! ip_check_mc(_in_dev_get (dev_out), oldkey
1810             ->dst))
1811             flags &= ~RTCF_LOCAL;
1812         read_unlock(&inetdev_lock);
1813         /* If multicast route do not exist use
1814            default one, but do not gateway in this case.
1815            Yes, it is hack.
1816            */
1817         if (res.fi && res.prefixlen < 4) {
1818             fib_info_put (res.fi);
1819             res.fi = NULL;
1820         }
1821     }
1822
1823     rth = dst_alloc(&ipv4_dst_ops);
1824     if (!rth)
1825         goto e_nobufs;
1826
1827     atomic_set(&rth->u.dst._refcnt, 1);
1828     rth->u.dst.flags= DST_HOST;
1829     rth->key.dst = oldkey->dst;
1830     rth->key.tos = tos;
1831     rth->key.src = oldkey->src;
1832     rth->key.iif = 0;
1833     rth->key.oif = oldkey->oif;

```

Figure 806: Route Lookup route request: `linux/net/ipv4/route.c`


```

1833 #ifdef CONFIG_IP_ROUTE_FWMARK
1834     rth->key.fwmark = oldkey->fwmark;
1835 #endif
1836     rth->rt_dst    = key.dst;
1837     rth->rt_src    = key.src;
1838 #ifdef CONFIG_IP_ROUTE_NAT
1839     rth->rt_dst_map = key.dst;
1840     rth->rt_src_map = key.src;
1841 #endif
1842     rth->rt_lif    = oldkey->oif ? : dev_out->ifindex;
1843     rth->u.dst.dev = dev_out;
1844     dev_hold(dev_out);
1845     rth->rt_gateway = key.dst;
1846     rth->rt_spec_dst = key.src;
1847
1848     rth->u.dst.output = ip_output;
1849
1850     if ( flags & RTCF_LOCAL ) {
1851         rth->u.dst.input = ip_local_deliver;
1852         rth->rt_spec_dst = key.dst;
1853     }

```

Figure 807: Route Lookup route request tidy up refs (“put”):
linux/net/ipv4/route.c

```

1854     if ( flags &(RTCF_BROADCAST|RTCF_MULTICAST)) {
1855         rth->rt_spec_dst = key.src;
1856         if ( flags &RTCF_LOCAL && !(dev_out->flags&IFF_LOOPBACK))
1857             rth->u.dst.output = ip_mc_output;
1858 #ifdef CONFIG_IP_MROUTE
1859     if (res.type == RTN_MULTICAST) {
1860         struct in_device *in_dev = in_dev_get(dev_out);
1861         if (in_dev) {
1862             if (IN_DEV_MFORWARD(in_dev) && !LOCAL_MCAST(
1863                 oldkey->dst)) {
1864                 rth->u.dst.input = ip_mr_input;
1865                 rth->u.dst.output = ip_mc_output;
1866             }
1867             in_dev_put(in_dev);
1868         }
1869 #endif
1870     }
1871
1872     rt_set_nexthop(rth, &res, 0);
1873
1874     rth->rt_flags = flags;
1875
1876     hash = rt_hash_code(oldkey->dst, oldkey->src^(oldkey->oif<<5), tos);
1877     err = rt_intern_hash(hash, rth, rp);
1878 done:
1879     if ( free_res )
1880         fib_res_put (&res);
1881     if ( dev_out )
1882         dev_put(dev_out);
1883     return err;
1884
1885 e_inval :
1886     err = -EINVAL;
1887     goto done;
1888 e_nobufs:
1889     err = -ENOBUFS;
1890     goto done;
1891 }

```

Figure 808: Route Lookup route request tidy up refs (“put”):
linux/net/ipv4/route.c

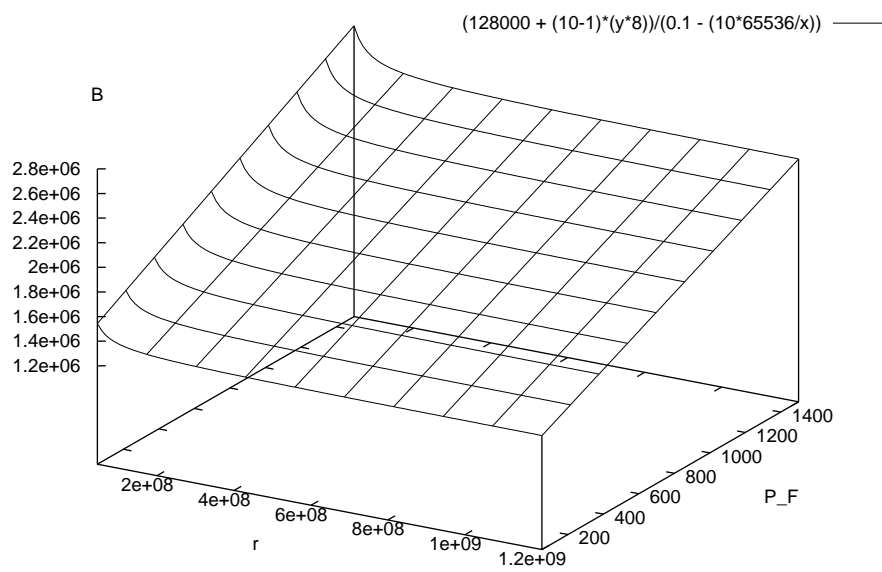


Figure 809: Guaranteed Bounded Delay and Utilisation

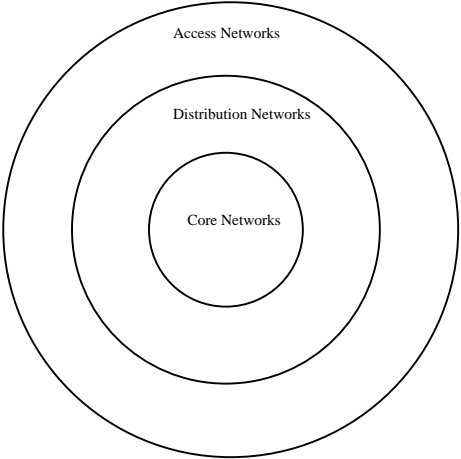


Figure 810: Network Hierarchy

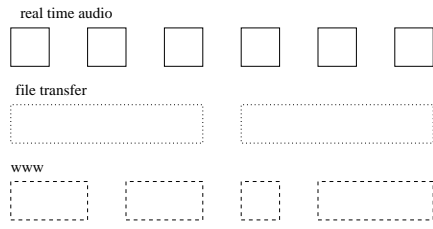


Figure 811: mixing traffic onto one wire

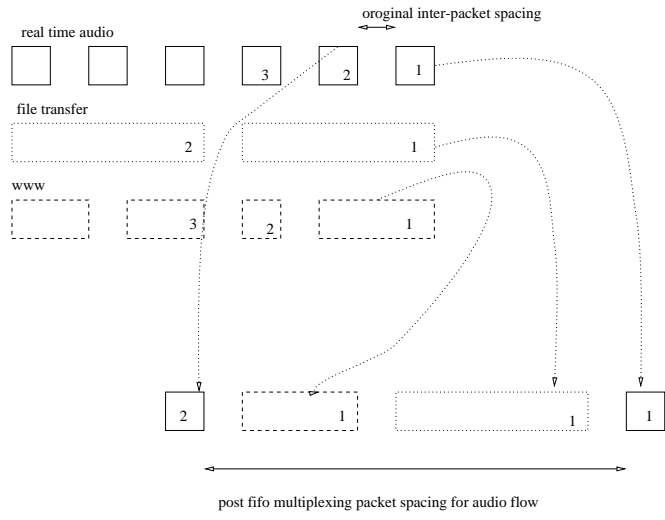


Figure 812: fifo sharing

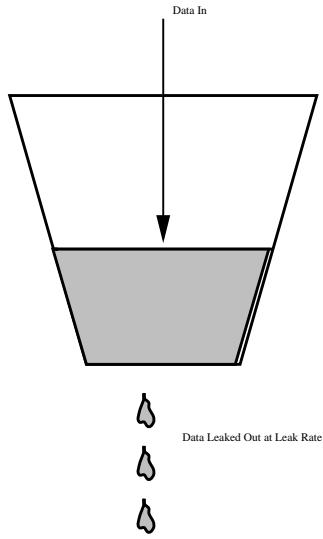


Figure 813: Leaky Bucket Description

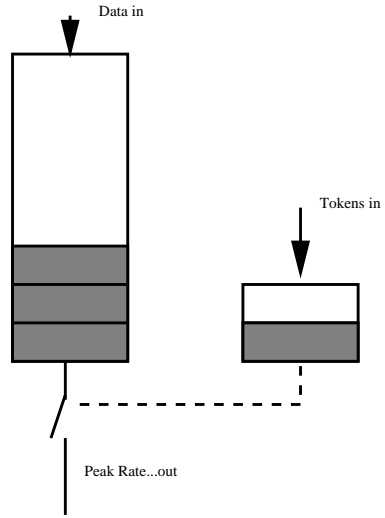


Figure 814: Token Bucket Description

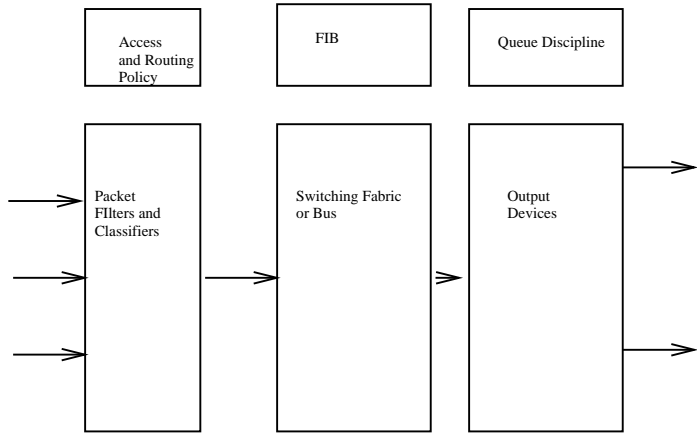


Figure 815: Router Schematic

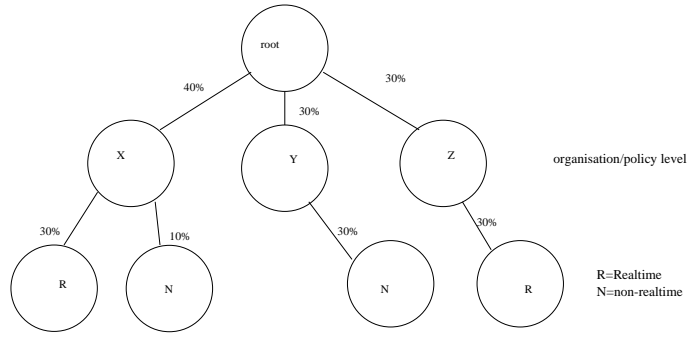


Figure 816: Class Based Queueing

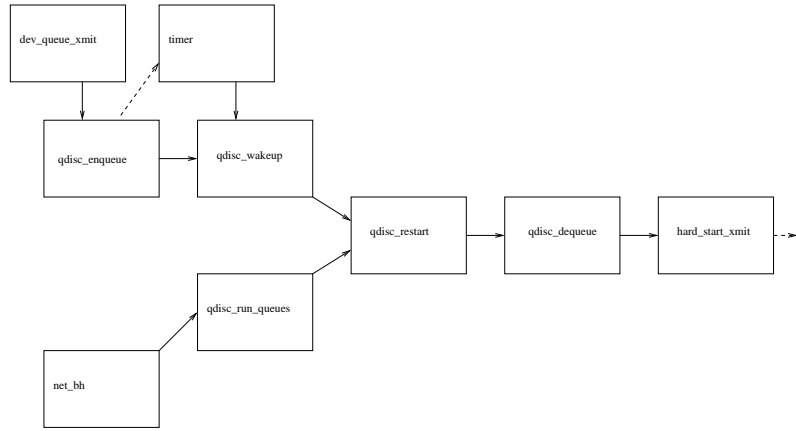


Figure 817: qdisc call graph structure



Figure 818: queueing discipline fine grain structure: classification via filters

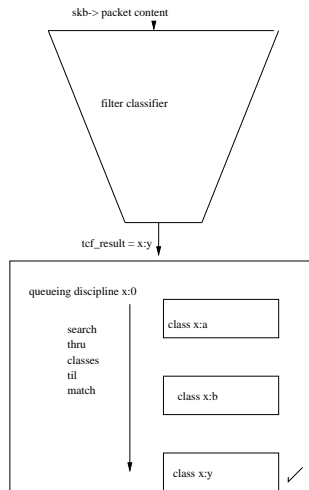


Figure 819: queueing discipline applying a filter

```

38  /* The list of all installed classifier types */
39
40  static struct tcf_proto_ops *tcf_proto_base;
41
42  /* Protects list of registered TC modules. It is pure SMP lock. */
43  static rwlock_t cls_mod_lock = RW_LOCK_UNLOCKED;
44
45  /* Find classifier type by string name */
46
47  struct tcf_proto_ops * tcf_proto_lookup_ops(struct rtattr *kind)
48  {
49      struct tcf_proto_ops *t = NULL;
50
51      if (kind) {
52          read_lock(&cls_mod_lock);
53          for (t = tcf_proto_base; t; t = t->next) {
54              if (rtattr_strcmp(kind, t->kind) == 0)
55                  break;
56          }
57          read_unlock(&cls_mod_lock);
58      }
59      return t;
60  }

```

Figure 820: Classifier: linux/net/sched/cls_api.c

```

62  /* Register(unregister) new classifier type */
63
64  int register_tcf_proto_ops (struct tcf_proto_ops *ops)
65  {
66      struct tcf_proto_ops *t, **tp;
67
68      write_lock(&cls_mod_lock);
69      for (tp = &tcf_proto_base; (t=*tp) != NULL; tp = &t->next) {
70          if (strcmp(ops->kind, t->kind) == 0) {
71              write_unlock(&cls_mod_lock);
72              return -EEXIST;
73          }
74      }
75
76      ops->next = NULL;
77      *tp = ops;
78      write_unlock(&cls_mod_lock);
79      return 0;
80  }

```

Figure 821: Classifier i: `linux/net/sched/cls_api.c`

```

82 int unregister_tcf_proto_ops (struct tcf_proto_ops *ops)
83 {
84     struct tcf_proto_ops *t, **tp;
85
86     write_lock(&cls_mod_lock);
87     for (tp = &tcf_proto_base; (t=*tp) != NULL; tp = &t->next)
88         if (t == ops)
89             break;
90
91     if (!t) {
92         write_unlock(&cls_mod_lock);
93         return -ENOENT;
94     }
95     *tp = t->next;
96     write_unlock(&cls_mod_lock);
97     return 0;
98 }

```

Figure 822: Classifier ii: `linux/net/sched/cls_api.c`


```

100 #ifdef CONFIG_RTNETLINK
101
102 static int tfilter_notify (struct sk_buff *oskb, struct nlmsg_hdr *n,
103                          struct tcf_proto *tp, unsigned long fh, int event);
104
105
106 /* Select new prio value from the range, managed by kernel. */
107
108 static __inline__ u32 tcf_auto_prio(struct tcf_proto *tp)
109 {
110     u32 first = TC_H_MAKE(0xC0000000U,0U);
111
112     if (tp)
113         first = tp->prio-1;
114
115     return first ;
116 }

```

Figure 823: Classifier iii: `linux/net/sched/cls_api.c`

```

118 /* Add/change/delete/get a filter node */
119
120 static int tc_ctl_tfilter (struct sk_buff *skb, struct nlmsg_hdr *n, void *arg)
121 {
122     struct rtattr **tca = arg;
123     struct tcmsg *t = NLMSG_DATA(n);
124     u32 protocol = TC_H_MIN(t->tcmsg_info);
125     u32 prio = TC_H_MAJ(t->tcmsg_info);
126     u32 nprio = prio;
127     u32 parent = t->tcmsg_parent;
128     struct net_device *dev;
129     struct Qdisc *q;
130     struct tcf_proto **back, **chain;
131     struct tcf_proto *tp = NULL;
132     struct tcf_proto_ops *tp_ops;
133     struct Qdisc_class_ops *cops;
134     unsigned long cl = 0;
135     unsigned long fh;
136     int err;
137
138     if (prio == 0) {
139         /* If no priority is given, user wants we allocated it. */
140         if (n->nmsg_type != RTM_NEWTFILTER || !(n->nmsg_flags &
141             NLM_F_CREATE))
142             return -ENOENT;
143         prio = TC_H_MAKE(0x80000000U, 0U);
144     }
145
146     /* Find head of filter chain. */
147
148     /* Find link */
149     if ((dev = __dev_get_by_index(t->tcmsg_ifindex)) == NULL)
150         return -ENODEV;

```

Figure 824: Classifier iv: linux/net/sched/cls_api.c

```

151     /* Find qdisc */
152     if (!parent) {
153         q = dev->qdisc_sleeping;
154         parent = q->handle;
155     } else if ((q = qdisc_lookup(dev, TC_H_MAJ(t->tcm_parent))) == NULL)
156         return -EINVAL;
157
158     /* Is it classful? */
159     if ((cops = q->ops->cl_ops) == NULL)
160         return -EINVAL;
161
162     /* Do we search for filter, attached to class? */
163     if (TC_H_MIN(parent)) {
164         cl = cops->get(q, parent);
165         if (cl == 0)
166             return -ENOENT;
167     }
168
169     /* And the last stroke */
170     chain = cops->tcf_chain(q, cl);
171     err = -EINVAL;
172     if (chain == NULL)
173         goto errout;

```

Figure 825: Classifier iv: `linux/net/sched/cls_api.c`

```

175     /* Check the chain for existence of proto-tcf with this priority */
176     for (back = chain; (tp=*back) != NULL; back = &tp->next) {
177         if (tp->prio >= prio) {
178             if (tp->prio == prio) {
179                 if (!nprio || (tp->protocol != protocol && protocol))
180                     goto errout;
181             } else
182                 tp = NULL;
183             break;
184         }
185     }
186
187     if (tp == NULL) {
188         /* Proto-tcf does not exist, create new one */
189
190         if (tca[TCA_KIND-1] == NULL || !protocol)
191             goto errout;
192
193         err = -ENOENT;
194         if (n->nmsg_type != RTM_NEWTFILTER || !(n->nmsg_flags &
195             NLM_F_CREATE))
196             goto errout;
197
198         /* Create new proto tcf */
199
200         err = -ENOBUFS;
201         if ((tp = kmalloc(sizeof(*tp), GFP_KERNEL)) == NULL)
202             goto errout;

```

Figure 826: Classifier v: linux/net/sched/cls_api.c

```

203         tp_ops = tcf_proto_lookup_ops(tca[TCA_KIND-1]);
204 #ifdef CONFIG_KMOD
205         if (tp_ops==NULL && tca[TCA_KIND-1] != NULL) {
206             struct rtattr *kind = tca[TCA_KIND-1];
207             char module_name[4 + IFNAMSIZ + 1];
208
209             if (RTA_PAYLOAD(kind) <= IFNAMSIZ) {
210                 sprintf(module_name, "cls_%s", (char*)RTA_DATA(kind));
211                 request_module(module_name);
212                 tp_ops = tcf_proto_lookup_ops(kind);
213             }
214         }
215 #endif
216         if (tp_ops == NULL) {
217             err = -EINVAL;
218             kfree(tp);
219             goto errout;
220         }
221         memset(tp, 0, sizeof(*tp));
222         tp->ops = tp_ops;
223         tp->protocol = protocol;
224         tp->prio = nprio ? : tcf_auto_prio(*back);
225         tp->q = q;
226         tp->classify = tp_ops->classify;
227         tp->classid = parent;
228         err = tp_ops->init(tp);
229         if (err) {
230             kfree(tp);
231             goto errout;
232         }
233         write_lock(&qdisc_tree_lock);
234         spin_lock_bh(&dev->queue_lock);
235         tp->next = *back;
236         *back = tp;
237         spin_unlock_bh(&dev->queue_lock);
238         write_unlock(&qdisc_tree_lock);
239     } else if (tca[TCA_KIND-1] && rtattr_strcmp(tca[TCA_KIND-1], tp->ops->
240         kind))
241         goto errout;

```

Figure 827: Classifier v: `linux/net/sched/cls_api.c`

```

242     fh = tp->ops->get(tp, t->tcm_handle);
243
244     if (fh == 0) {
245         if (n->nmsg_type == RTM_DELTFILTER && t->tcm_handle == 0) {
246             write_lock(&qdisc_tree_lock);
247             spin_lock_bh(&dev->queue_lock);
248             *back = tp->next;
249             spin_unlock_bh(&dev->queue_lock);
250             write_unlock(&qdisc_tree_lock);
251
252             tp->ops->destroy(tp);
253             kfree(tp);
254             err = 0;
255             goto errout;
256         }
257
258         err = -ENOENT;
259         if (n->nmsg_type != RTM_NEWTFILTER || !(n->nmsg_flags&
260             NLM_F_CREATE))
261             goto errout;
262     } else {
263         switch (n->nmsg_type) {
264             case RTM_NEWTFILTER:
265                 err = -EEXIST;
266                 if (n->nmsg_flags&NLM_F_EXCL)
267                     goto errout;
268                 break;
269             case RTM_DELTFILTER:
270                 err = tp->ops->delete(tp, fh);
271                 goto errout;
272             case RTM_GETTFILTER:
273                 err = tfilter_notify(skb, n, tp, fh, RTM_NEWTFILTER);
274                 goto errout;
275             default:
276                 err = -EINVAL;
277                 goto errout;
278         }
279     }

```

Figure 828: Classifier vi: linux/net/sched/cls_api.c

```
280     err = tp->ops->change(tp, cl, t->tcn_handle, tca, &fh);
281     if (err == 0)
282         tfilter_notify (skb, n, tp, fh, RTM_NEWTFILTER);
283
284 errout:
285     if (cl)
286         cops->put(q, cl);
287     return err;
288 }
```

Figure 829: Classifier vi: `linux/net/sched/cls_api.c`

```

290 static int
291 tcf_fill_node (struct sk_buff *skb, struct tcf_proto *tp, unsigned long fh,
292               u32 pid, u32 seq, unsigned flags, int event)
293 {
294     struct tcmsg *tcm;
295     struct nlmsg_hdr *nlh;
296     unsigned char *b = skb->tail;
297
298     nlh = NLMSG_PUT(skb, pid, seq, event, sizeof(*tcm));
299     nlh->nlmsg_flags = flags;
300     tcm = NLMSG_DATA(nlh);
301     tcm->tcm_family = AF_UNSPEC;
302     tcm->tcm_ifindex = tp->q->dev->ifindex;
303     tcm->tcm_parent = tp->classid;
304     tcm->tcm_handle = 0;
305     tcm->tcm_info = TC_H_MAKE(tp->prio, tp->protocol);
306     RTA_PUT(skb, TCA_KIND, IFNAMSIZ, tp->ops->kind);
307     if (tp->ops->dump && tp->ops->dump(tp, fh, skb, tcm) < 0)
308         goto rtattr_failure ;
309     nlh->nlmsg_len = skb->tail - b;
310     return skb->len;
311
312 nlmsg_failure:
313 rtattr_failure :
314     skb_trim(skb, b - skb->data);
315     return -1;
316 }

```

Figure 830: Classifier vii: `linux/net/sched/cls_api.c`


```

318 static int tfilter_notify (struct sk_buff *oskb, struct nlmsg_hdr *n,
319                          struct tcf_proto *tp, unsigned long fh, int event)
320 {
321     struct sk_buff *skb;
322     u32 pid = oskb ? NETLINK_CB(oskb).pid : 0;
323
324     skb = alloc_skb(NLMSG_GOODSIZE, GFP_KERNEL);
325     if (!skb)
326         return -ENOBUFS;
327
328     if ( tcf_fill_node (skb, tp, fh, pid, n->nlmsg_seq, 0, event) <= 0) {
329         kfree_skb(skb);
330         return -EINVAL;
331     }
332
333     return rtnetlink_send(skb, pid, RTMGRP_TC, n->nlmsg_flags&NLM_F_ECHO);
334 }

```

Figure 831: Classifier viii: `linux/net/sched/cls_api.c`

```

336 struct tcf_dump_args
337 {
338     struct tcf_walker w;
339     struct sk_buff *skb;
340     struct netlink_callback *cb;
341 };
342
343 static int tcf_node_dump(struct tcf_proto *tp, unsigned long n, struct tcf_walker *arg)
344 {
345     struct tcf_dump_args *a = (void*)arg;
346
347     return tcf_fill_node (a->skb, tp, n, NETLINK_CB(a->cb->skb).pid,
348                          a->cb->nlh->nmsg_seq, NLM_F_MULTI,
349                          RTM_NEWTFILTER);

```

Figure 832: Classifier - Dump Output: `linux/net/sched/cls_api.c`

```

351 static int tc_dump_tfilter(struct sk_buff *skb, struct netlink_callback *cb)
352 {
353     int t;
354     int s_t;
355     struct net_device *dev;
356     struct Qdisc *q;
357     struct tcf_proto *tp, **chain;
358     struct tcmsg *tcm = (struct tcmsg*)NLMSG_DATA(cb->nlh);
359     unsigned long cl = 0;
360     struct Qdisc_class_ops *cops;
361     struct tcf_dump_args arg;
362
363     if (cb->nlh->nmlmsg_len < NLMSG_LENGTH(sizeof(*tcm)))
364         return skb->len;
365     if ((dev = dev_get_by_index(tcm->tcm_ifindex)) == NULL)
366         return skb->len;
367
368     read_lock(&qdisc_tree_lock);
369     if (!tcm->tcm_parent)
370         q = dev->qdisc_sleeping;
371     else
372         q = qdisc_lookup(dev, TC_H_MAJ(tcm->tcm_parent));
373     if (q == NULL) {
374         read_unlock(&qdisc_tree_lock);
375         dev_put(dev);
376         return skb->len;
377     }
378     if ((cops = q->ops->cl_ops) == NULL)
379         goto errout;
380     if (TC_H_MIN(tcm->tcm_parent)) {
381         cl = cops->get(q, tcm->tcm_parent);
382         if (cl == 0)
383             goto errout;
384     }
385     chain = cops->tcf_chain(q, cl);
386     if (chain == NULL)
387         goto errout;
388
389     s_t = cb->args[0];

```

Figure 833: Classifier - Dump Output: linux/net/sched/cls_api.c

```

391     for (tp=*chain, t=0; tp; tp = tp->next, t++) {
392         if (t < s.t) continue;
393         if (TC_H_MAJ(tcm->tcm_info) &&
394             TC_H_MAJ(tcm->tcm_info) != tp->prio)
395             continue;
396         if (TC_H_MIN(tcm->tcm_info) &&
397             TC_H_MIN(tcm->tcm_info) != tp->protocol)
398             continue;
399         if (t > s.t)
400             memset(&cb->args[1], 0, sizeof(cb->args)-sizeof(cb->args[0]));
401         if (cb->args[1] == 0) {
402             if ( tcf_fill_node (skb, tp, 0, NETLINK_CB(cb->skb).pid,
403                             cb->nlh->nmsg_seq, NLM_F_MULTI,
404                             RTM_NEWTFILTER) <= 0) {
405                 break;
406             }
407             cb->args[1] = 1;
408         }
409         if (tp->ops->walk == NULL)
410             continue;
411         arg.w.fn = tcf_node_dump;
412         arg.skbn = skb;

```

Figure 834: Classifier - Dump Output: linux/net/sched/cls_api.c

```

436 int _init tc_filter_init (void)
437 {
438 #ifdef CONFIG_RTNETLINK
439     struct rtnetlink_link *link_p = rtnetlink_links [PF_UNSPEC];
440
441     /* Setup rtnetlink links. It is made here to avoid
442        exporting large number of public symbols.
443        */
444
445     if (link_p) {
446         link_p [RTM_NEWTFILTER - RTM_BASE].doit = tc_ctl_tfilter;
447         link_p [RTM_DELTTFILTER - RTM_BASE].doit = tc_ctl_tfilter;
448         link_p [RTM_GETTFILTER - RTM_BASE].doit = tc_ctl_tfilter;
449         link_p [RTM_GETTFILTER - RTM_BASE].dumpit = tc_dump_tfilter;
450     }
451 #endif
452 #define INIT_TC_FILTER(name) { \
453     extern struct tcf_proto_ops cls_##name##_ops; \
454     register_tcf_proto_ops (&cls_##name##_ops); \
455 }
456
457 #ifdef CONFIG_NET_CLS_U32
458     INIT_TC_FILTER(u32);
459 #endif
460 #ifdef CONFIG_NET_CLS_ROUTE4
461     INIT_TC_FILTER(route4);
462 #endif
463 #ifdef CONFIG_NET_CLS_FW
464     INIT_TC_FILTER(fw);
465 #endif
466 #ifdef CONFIG_NET_CLS_RSVP
467     INIT_TC_FILTER(rsvp);
468 #endif
469 #ifdef CONFIG_NET_CLS_TCINDEX
470     INIT_TC_FILTER(tcindex);
471 #endif
472 #ifdef CONFIG_NET_CLS_RSVP6
473     INIT_TC_FILTER(rsvp6);
474 #endif
475     return 0;
476 }

```

Figure 835: Classifier- Initialisation: linux/net/sched/cls_api.c

```

56 struct tc_u_knode
57 {
58     struct tc_u_knode    *next;
59     u32                   handle;
60     struct tc_u_hnode    *ht_up;
61 #ifdef CONFIG_NET_CLS_POLICE
62     struct tcf_police     *police;
63 #endif
64     struct tcf_result     res;
65     struct tc_u_hnode    *ht_down;
66     struct tc_u32_sel     sel;
67 };
68
69 struct tc_u_hnode
70 {
71     struct tc_u_hnode    *next;
72     u32                   handle;
73     struct tc_u_common    *tp_c;
74     int                   refcnt;
75     unsigned              divisor;
76     u32                   hgenerator;
77     struct tc_u_knode    *ht [1];
78 };
79
80 struct tc_u_common
81 {
82     struct tc_u_common    *next;
83     struct tc_u_hnode    *hlist;
84     struct Qdisc          *q;
85     int                   refcnt;
86     u32                   hgenerator;
87 };
88
89 static struct tc_u_common *u32_list;
90
91 static __inline__ unsigned u32_hash_fold(u32 key, struct tc_u32_sel *sel)
92 {
93     unsigned h = key & sel->hmask;
94
95     h ^= h >> 16;
96     h ^= h >> 8;
97     return h;
98 }

```

Figure 836: Universal 32 bit Classifier API - Global Data:
linux/net/sched/cls_u32.c

```

101 {
102     struct {
103         struct tc_u_knode *knode;
104         u8 *ptr;
105     } stack[TC_U32_MAXDEPTH];
106
107     struct tc_u_hnode *ht = (struct tc_u_hnode*)tp->root;
108     u8 *ptr = skb->nh.raw;
109     struct tc_u_knode *n;
110     int sdepth = 0;
111     int off2 = 0;
112     int sel = 0;
113     int i;
114
115     #if !defined(_i386_) && !defined(_mc68000_)
116         if ((unsigned long)ptr & 3)
117             return -1;
118     #endif
119
120 next_ht:
121     n = ht->ht[sel];
122
123 next_knode:
124     if (n) {
125         struct tc_u32_key *key = n->sel.keys;
126
127         for (i = n->sel.nkeys; i > 0; i--, key++) {
128             if (((u32*)(ptr+key->off+(off2&key->offmask))^key->val)&
129                 key->mask) {
130                 n = n->next;
131                 goto next_knode;
132             }
133             if (n->ht_down == NULL) {
134 check_terminal:
135                 if (n->sel.flags&TC_U32_TERMINAL) {
136                     *res = n->res;
137 #ifdef CONFIG_NET_CLS_POLICE
138                     if (n->police) {
139                         int pol_res = tcf_police(skb, n->police);
140                         if (pol_res >= 0)
141                             return pol_res;
142                     } else
143 #endif
144                     return 0;
145                 }
146                 n = n->next;
147                 goto next_knode;
148             }
149
150             /* PUSH */
151             if (sdepth >= TC_U32_MAXDEPTH)
152                 goto deadlock;

```

Figure 837: u32 API- the Classifier i: `linux/net/sched/cls_u32.c`

```

153     stack[sdepth].knode = n;
154     stack[sdepth].ptr = ptr;
155     sdepth++;
156
157     ht = n->ht_down;
158     sel = 0;
159     if (ht->divisor)
160         sel = ht->divisor&u32_hash_fold(*(u32*)(ptr+n->sel.hoff), &n
->sel);
161
162     if (!(n->sel.flags&(TC_U32_VAROFFSET|TC_U32_OFFSET|
TC_U32_EAT)))
163         goto next_ht;
164
165     if (n->sel.flags&(TC_U32_EAT|TC_U32_VAROFFSET)) {
166         off2 = n->sel.off + 3;
167         if (n->sel.flags&TC_U32_VAROFFSET)
168             off2 += ntohs(n->sel.offmask & *(u16*)(ptr+n->sel.
offoff)) >>n->sel.offshift;
169
170         off2 &= ~3;
171     }
172     if (n->sel.flags&TC_U32_EAT) {
173         ptr += off2;
174         off2 = 0;
175     }
176     if (ptr < skb->tail)
177         goto next_ht;
178 }
179
180 /* POP */
181 if (sdepth--> {
182     n = stack[sdepth].knode;
183     ht = n->ht_up;
184     ptr = stack[sdepth].ptr;
185     goto check_terminal;
186 }
187 return -1;
188
189 deadlock:
190 if (net_ratelimit ())
191     printk("cls_u32: _dead_loop\n");
192 return -1;
193 }

```

Figure 838: u32 API- the Classifier ii: linux/net/sched/cls_u32.c


```

195 static __inline__ struct tc_u_hnode *
196 u32_lookup_ht(struct tc_u_common *tp_c, u32 handle)
197 {
198     struct tc_u_hnode *ht;
199
200     for (ht = tp_c->hlist; ht; ht = ht->next)
201         if (ht->handle == handle)
202             break;
203
204     return ht;
205 }
206
207 static __inline__ struct tc_u_knode *
208 u32_lookup_key(struct tc_u_hnode *ht, u32 handle)
209 {
210     unsigned sel;
211     struct tc_u_knode *n;
212
213     sel = TC_U32_HASH(handle);
214     if (sel > ht->divisor)
215         return 0;
216
217     for (n = ht->ht[sel]; n; n = n->next)
218         if (n->handle == handle)
219             return n;
220
221     return NULL;
222 }

```

Figure 839: u32 API - lookup functions: `linux/net/sched/cls_u32.c`

```

225 static unsigned long u32_get(struct tcf_proto *tp, u32 handle)
226 {
227     struct tc_u_hnode *ht;
228     struct tc_u_common *tp_c = tp->data;
229
230     if (TC_U32_HTID(handle) == TC_U32_ROOT)
231         ht = tp->root;
232     else
233         ht = u32_lookup_ht(tp_c, TC_U32_HTID(handle));
234
235     if (!ht)
236         return 0;
237
238     if (TC_U32_KEY(handle) == 0)
239         return (unsigned long)ht;
240
241     return (unsigned long)u32_lookup_key(ht, handle);
242 }
243
244 static void u32_put(struct tcf_proto *tp, unsigned long f)
245 {
246 }
247
248 static u32 gen_new_htid(struct tc_u_common *tp_c)
249 {
250     int i = 0x800;
251
252     do {
253         if (++tp_c->hgenerator == 0x7FF)
254             tp_c->hgenerator = 1;
255     } while (--i>0 && u32_lookup_ht(tp_c, (tp_c->hgenerator|0x800)<<20));
256
257     return i > 0 ? (tp_c->hgenerator|0x800)<<20 : 0;
258 }

```

Figure 840: u32 API - Misc Support: `linux/net/sched/cls_u32.c`

```

260 static int u32_init(struct tcf_proto *tp)
261 {
262     struct tc_u_hnode *root_ht;
263     struct tc_u_common *tp_c;
264
265     MOD_INC_USE_COUNT;
266
267     for (tp_c = u32_list; tp_c; tp_c = tp_c->next)
268         if (tp_c->q == tp->q)
269             break;
270
271     root_ht = kmalloc(sizeof(*root_ht), GFP_KERNEL);
272     if (root_ht == NULL) {
273         MOD_DEC_USE_COUNT;
274         return -ENOBUFS;
275     }
276     memset(root_ht, 0, sizeof(*root_ht));
277     root_ht->divisor = 0;
278     root_ht->refcnt++;
279     root_ht->handle = tp_c ? gen_new_htid(tp_c) : 0x80000000;
280
281     if (tp_c == NULL) {
282         tp_c = kmalloc(sizeof(*tp_c), GFP_KERNEL);
283         if (tp_c == NULL) {
284             kfree(root_ht);
285             MOD_DEC_USE_COUNT;
286             return -ENOBUFS;
287         }
288         memset(tp_c, 0, sizeof(*tp_c));
289         tp_c->q = tp->q;
290         tp_c->next = u32_list;
291         u32_list = tp_c;
292     }
293
294     tp_c->refcnt++;
295     root_ht->next = tp_c->hlist;
296     tp_c->hlist = root_ht;
297     root_ht->tp_c = tp_c;
298
299     tp->root = root_ht;
300     tp->data = tp_c;
301     return 0;
302 }

```

Figure 841: u32 API - Initialisation: linux/net/sched/cls_u32.c

```

304 static int u32_destroy_key(struct tcf_proto *tp, struct tc_u_knode *n)
305 {
306     unsigned long cl;
307
308     if ((cl = _cls_set_class (&n->res.class, 0)) != 0)
309         tp->q->ops->cl_ops->unbind_tcf(tp->q, cl);
310 #ifdef CONFIG_NET_CLS_POLICE
311     tcf_police_release (n->police);
312 #endif
313     if (n->ht_down)
314         n->ht_down->refcnt--;
315     kfree(n);
316     return 0;
317 }
318
319 static int u32_delete_key(struct tcf_proto *tp, struct tc_u_knode* key)
320 {
321     struct tc_u_knode **kp;
322     struct tc_u_hnode *ht = key->ht_up;
323
324     if (ht) {
325         for (kp = &ht->ht[TC_U32_HASH(key->handle)]; *kp; kp = &(*kp)->
326             next) {
327             if (*kp == key) {
328                 tcf_tree_lock (tp);
329                 *kp = key->next;
330                 tcf_tree_unlock (tp);
331
332                 u32_destroy_key(tp, key);
333                 return 0;
334             }
335         }
336     }
337     BUG_TRAP(0);
338     return 0;
339 }
340
341 static void u32_clear_hnode(struct tcf_proto *tp, struct tc_u_hnode *ht)
342 {
343     struct tc_u_knode *n;
344     unsigned h;
345
346     for (h=0; h<=ht->divisor; h++) {
347         while ((n = ht->ht[h]) != NULL) {
348             ht->ht[h] = n->next;
349             u32_destroy_key(tp, n);
350         }
351     }
352 }

```

Figure 842: u32 API - Destroy/Delete Key: linux/net/sched/cls_u32.c

```

354 static int u32_destroy_hnode(struct tcf_proto *tp, struct tc_u_hnode *ht)
355 {
356     struct tc_u_common *tp_c = tp->data;
357     struct tc_u_hnode **hn;
358
359     BUG_TRAP(!ht->refcnt);
360
361     u32_clear_hnode(tp, ht);
362
363     for (hn = &tp_c->hlist; *hn; hn = &(*hn)->next) {
364         if (*hn == ht) {
365             *hn = ht->next;
366             kfree(ht);
367             return 0;
368         }
369     }
370
371     BUG_TRAP(0);
372     return -ENOENT;
373 }
374
375 static void u32_destroy(struct tcf_proto *tp)
376 {
377     struct tc_u_common *tp_c = tp->data;
378     struct tc_u_hnode *root_ht = xchg(&tp->root, NULL);
379
380     BUG_TRAP(root_ht != NULL);
381
382     if (root_ht && --root_ht->refcnt == 0)
383         u32_destroy_hnode(tp, root_ht);
384
385     if (--tp_c->refcnt == 0) {
386         struct tc_u_hnode *ht;
387         struct tc_u_common **tp_cp;
388
389         for (tp_cp = &u32_list; *tp_cp; tp_cp = &(*tp_cp)->next) {
390             if (*tp_cp == tp_c) {
391                 *tp_cp = tp_c->next;
392                 break;
393             }
394         }
395
396         for (ht = tp_c->hlist; ht; ht = ht->next)
397             u32_clear_hnode(tp, ht);
398
399         while ((ht = tp_c->hlist) != NULL) {
400             tp_c->hlist = ht->next;
401
402             BUG_TRAP(ht->refcnt == 0);
403
404             kfree(ht);
405         };
406
407         kfree(tp_c);
408     }
409
410     MOD_DEC_USE_COUNT;
411     tp->data = NULL;
412 }

```

Figure 843: u32 API - Destroy/Delete Classifier: `linux/net/sched/cfs_u32.c`

```

414 static int u32_delete(struct tcf_proto *tp, unsigned long arg)
415 {
416     struct tc_u_hnode *ht = (struct tc_u_hnode*)arg;
417
418     if (ht == NULL)
419         return 0;
420
421     if (TC_U32_KEY(ht->handle))
422         return u32_delete_key(tp, (struct tc_u_knode*)ht);
423
424     if (tp->root == ht)
425         return -EINVAL;
426
427     if (--ht->refcnt == 0)
428         u32_destroy_hnode(tp, ht);
429
430     return 0;
431 }
432
433 static u32 gen_new_kid(struct tc_u_hnode *ht, u32 handle)
434 {
435     struct tc_u_knode *n;
436     unsigned i = 0x7FFF;
437
438     for (n=ht->ht[TC_U32_HASH(handle)]; n; n = n->next)
439         if (i < TC_U32_NODE(n->handle))
440             i = TC_U32_NODE(n->handle);
441     i++;
442
443     return handle|(i>0xFFFF ? 0xFFFF : i);
444 }

```

Figure 844: u32 API - New Key Id: `linux/net/sched/cls_u32.c`

```

446 static int u32_set_parms(struct Qdisc *q, unsigned long base,
447                        struct tc_u_hnode *ht,
448                        struct tc_u_knode *n, struct rtattr **tb,
449                        struct rtattr *est)
450 {
451     if (tb[TCA_U32_LINK-1]) {
452         u32 handle = *(u32*)RTA_DATA(tb[TCA_U32_LINK-1]);
453         struct tc_u_hnode *ht_down = NULL;
454
455         if (TC_U32_KEY(handle))
456             return -EINVAL;
457
458         if (handle) {
459             ht_down = u32_lookup_ht(ht->tp_c, handle);
460
461             if (ht_down == NULL)
462                 return -EINVAL;
463             ht_down->refcnt++;
464         }
465
466         sch_tree_lock(q);
467         ht_down = xchg(&n->ht_down, ht_down);
468         sch_tree_unlock(q);
469
470         if (ht_down)
471             ht_down->refcnt--;
472     }
473     if (tb[TCA_U32_CLASSID-1]) {
474         unsigned long cl;
475
476         n->res.classid = *(u32*)RTA_DATA(tb[TCA_U32_CLASSID-1]);
477         sch_tree_lock(q);
478         cl = _cls_set_class (&n->res.class, q->ops->cl_ops->bind_tcf(q, base,
479                                n->res.classid));
480         sch_tree_unlock(q);
481         if (cl)
482             q->ops->cl_ops->unbind_tcf(q, cl);
483     }
484     #ifdef CONFIG_NET_CLS_POLICE
485     if (tb[TCA_U32_POLICE-1]) {
486         struct tcf_police *police = tcf_police_locate (tb[TCA_U32_POLICE-1],
487                                                       est);
488
489         sch_tree_lock(q);
490         police = xchg(&n->police, police);
491         sch_tree_unlock(q);
492
493         tcf_police_release (police);
494     }
495     #endif
496     return 0;
497 }

```

Figure 845: u32 API - Set Parameters: `linux/net/sched/cls_u32.c`

```

497 static int u32_change(struct tcf_proto *tp, unsigned long base, u32 handle,
498                     struct rtattr **tca,
499                     unsigned long *arg)
500 {
501     struct tc_u_common *tp_c = tp->data;
502     struct tc_u_hnode *ht;
503     struct tc_u_knode *n;
504     struct tc_u32_sel *s;
505     struct rtattr *opt = tca[TCA_OPTIONS-1];
506     struct rtattr *tb[TCA_U32_MAX];
507     u32 htid;
508     int err;
509
510     if (opt == NULL)
511         return handle ? -EINVAL : 0;
512
513     if (rtattr_parse(tb, TCA_U32_MAX, RTA_DATA(opt), RTA_PAYLOAD(opt))
514         < 0)
515         return -EINVAL;
516
517     if ((n = (struct tc_u_knode*)*arg) != NULL) {
518         if (TC_U32_KEY(n->handle) == 0)
519             return -EINVAL;
520
521         return u32_set_parms(tp->q, base, n->ht_up, n, tb, tca[TCA_RATE-1]);
522     }
523
524     if (tb[TCA_U32_DIVISOR-1]) {
525         unsigned divisor = *(unsigned*)RTA_DATA(tb[TCA_U32_DIVISOR-1]);
526
527         if (--divisor > 0x100)
528             return -EINVAL;
529         if (TC_U32_KEY(handle))
530             return -EINVAL;
531         if (handle == 0) {
532             handle = gen_new_htid(tp->data);
533             if (handle == 0)
534                 return -ENOMEM;
535         }
536         ht = kmalloc(sizeof(*ht) + divisor*sizeof(void*), GFP_KERNEL);
537         if (ht == NULL)
538             return -ENOBUFS;
539         memset(ht, 0, sizeof(*ht) + divisor*sizeof(void*));
540         ht->tp_c = tp_c;
541         ht->refcnt = 0;
542         ht->divisor = divisor;
543         ht->handle = handle;
544         ht->next = tp_c->hlist;
545         tp_c->hlist = ht;
546         *arg = (unsigned long)ht;
547         return 0;
548     }
549 }

```

Figure 846: u32 API - Change Parameters: linux/net/sched/c1s_u32.c


```

549     if (tb[TCA_U32_HASH-1]) {
550         htid = *(unsigned*)RTA_DATA(tb[TCA_U32_HASH-1]);
551         if (TC_U32_HTID(htid) == TC_U32_ROOT) {
552             ht = tp->root;
553             htid = ht->handle;
554         } else {
555             ht = u32_lookup_ht(tp->data, TC_U32_HTID(htid));
556             if (ht == NULL)
557                 return -EINVAL;
558         }
559     } else {
560         ht = tp->root;
561         htid = ht->handle;
562     }
563
564     if (ht->divisor < TC_U32_HASH(htid))
565         return -EINVAL;
566
567     if (handle) {
568         if (TC_U32_HTID(handle) && TC_U32_HTID(handle^htid))
569             return -EINVAL;
570         handle = htid | TC_U32_NODE(handle);
571     } else
572         handle = gen_new_kid(ht, htid);
573
574     if (tb[TCA_U32_SEL-1] == 0 ||
575         RTA_PAYLOAD(tb[TCA_U32_SEL-1]) < sizeof(struct tc_u32_sel))
576         return -EINVAL;
577
578     s = RTA_DATA(tb[TCA_U32_SEL-1]);
579     n = kmalloc(sizeof(*n) + s->nkeys*sizeof(struct tc_u32_key), GFP_KERNEL);
580     if (n == NULL)
581         return -ENOBUFS;
582     memset(n, 0, sizeof(*n) + s->nkeys*sizeof(struct tc_u32_key));
583     memcpy(&n->sel, s, sizeof(*s) + s->nkeys*sizeof(struct tc_u32_key));
584     n->ht_up = ht;
585     n->handle = handle;
586     err = u32_set_parms(tp->q, base, ht, n, tb, tca[TCA_RATE-1]);
587     if (err == 0) {
588         struct tc_u_knode **ins;
589         for (ins = &ht->ht[TC_U32_HASH(handle)]; *ins; ins = &(*ins)->next)
590             if (TC_U32_NODE(handle) < TC_U32_NODE((*ins)->handle))
591                 break;
592
593         n->next = *ins;
594         wmb();
595         *ins = n;
596
597         *arg = (unsigned long)n;
598         return 0;
599     }
600     kfree(n);
601     return err;
602 }

```

Figure 847: u32 API - Change Parameters: `linux/net/sched/cls_u32.c`

```

604 static void u32_walk(struct tcf_proto *tp, struct tcf_walker *arg)
605 {
606     struct tc_u_common *tp_c = tp->data;
607     struct tc_u_hnode *ht;
608     struct tc_u_knode *n;
609     unsigned h;
610
611     if (arg->stop)
612         return;
613
614     for (ht = tp_c->hlist; ht; ht = ht->next) {
615         if (arg->count >= arg->skip) {
616             if (arg->fn(tp, (unsigned long)ht, arg) < 0) {
617                 arg->stop = 1;
618                 return;
619             }
620         }
621         arg->count++;
622         for (h = 0; h <= ht->divisor; h++) {
623             for (n = ht->ht[h]; n; n = n->next) {
624                 if (arg->count < arg->skip) {
625                     arg->count++;
626                     continue;
627                 }
628                 if (arg->fn(tp, (unsigned long)n, arg) < 0) {
629                     arg->stop = 1;
630                     return;
631                 }
632                 arg->count++;
633             }
634         }
635     }
636 }

```

Figure 848: u32 API - Walk Tree: `linux/net/sched/cls_u32.c`

```

639 static int u32_dump(struct tcf_proto *tp, unsigned long fh,
640                  struct sk_buff *skb, struct tcmsg *)
641 {
642     struct tc_u_knode *n = (struct tc_u_knode*)fh;
643     unsigned char *b = skb->tail;
644     struct rtattr *rta;
645
646     if (n == NULL)
647         return skb->len;
648
649     t->tcn_handle = n->handle;
650
651     rta = (struct rtattr*)b;
652     RTA_PUT(skb, TCA_OPTIONS, 0, NULL);
653
654     if (TC_U32_KEY(n->handle) == 0) {
655         struct tc_u_hnode *ht = (struct tc_u_hnode*)fh;
656         u32 divisor = ht->divisor+1;
657         RTA_PUT(skb, TCA_U32_DIVISOR, 4, &divisor);
658     } else {
659         RTA_PUT(skb, TCA_U32_SEL,
660                sizeof(n->sel) + n->sel.nkeys*sizeof(struct tc_u32_key),
661                &n->sel);
662         if (n->ht_up) {
663             u32 htid = n->handle & 0xFFFFF000;
664             RTA_PUT(skb, TCA_U32_HASH, 4, &htid);
665         }
666         if (n->res.classid)
667             RTA_PUT(skb, TCA_U32_CLASSID, 4, &n->res.classid);
668         if (n->ht_down)
669             RTA_PUT(skb, TCA_U32_LINK, 4, &n->ht_down->handle);
670 #ifdef CONFIG_NET_CLS_POLICE
671         if (n->police) {
672             struct rtattr *p_rta = (struct rtattr*)skb->tail;
673
674             RTA_PUT(skb, TCA_U32_POLICE, 0, NULL);
675
676             if (tcf_police_dump(skb, n->police) < 0)
677                 goto rtattr_failure ;
678
679             p_rta->rta_len = skb->tail - (u8*)p_rta;
680         }
681 #endif
682     }
683
684     rta->rta_len = skb->tail - b;
685 #ifdef CONFIG_NET_CLS_POLICE
686     if (TC_U32_KEY(n->handle) && n->police) {
687         if (qdisc_copy_stats(skb, &n->police->stats))
688             goto rtattr_failure ;
689     }
690 #endif
691     return skb->len;
692
693 rtattr_failure :
694     skb_trim(skb, b - skb->data);
695     return -1;
696 }

```

Figure 849: u32 API - RTNETLINK Comms: `linux/net/sched/cls_u32.c`

```

698
699 struct tcf_proto_ops cls_u32_ops = {
700     NULL,
701     "u32",
702     u32_classify ,
703     u32_init ,
704     u32_destroy,
705
706     u32_get,
707     u32_put,
708     u32_change,
709     u32_delete,
710     u32_walk,
711 #ifdef CONFIG_RTNETLINK
712     u32_dump
713 #else
714     NULL
715 #endif
716 };
717
718 #ifdef MODULE
719 int init_module(void)
720 {
721     return register_tcf_proto_ops (&cls_u32_ops);
722 }
723
724 void cleanup_module(void)
725 {
726     unregister_tcf_proto_ops (&cls_u32_ops);
727 }

```

Figure 850: u32 API - TCF Operationlinux/net/sched/Registration:
linux/net/sched/cls_u32.c

```

12  /*
13     Comparing to general packet classification problem,
14     RSVP needs only several relatively simple rules:
15
16     * (dst, protocol) are always specified,
17     so that we are able to hash them.
18     * src may be exact, or may be wildcard, so that
19     we can keep a hash table plus one wildcard entry.
20     * source port (or flow label) is important only if src is given.
21
22     IMPLEMENTATION.
23
24     We use a two level hash table: The top level is keyed by
25     destination address and protocol ID, every bucket contains a list
26     of "rsvp sessions", identified by destination address, protocol and
27     DPI(="Destination Port ID"): triple (key, mask, offset).
28
29     Every bucket has a smaller hash table keyed by source address
30     (cf. RSVP flowspec) and one wildcard entry for wildcard reservations.
31     Every bucket is again a list of "RSVP flows", selected by
32     source address and SPI(="Source Port ID" here rather than
33     "security parameter index"): triple (key, mask, offset).
34
35
36     NOTE 1. All the packets with IPv6 extension headers (but AH and ESP)
37     and all fragmented packets go to the best-effort traffic class.
38
39
40     NOTE 2. Two "port id"'s seems to be redundant, rfc2207 requires
41     only one "Generalized Port Identifier". So that for classic
42     ah, esp (and udp,tcp) both *pi should coincide or one of them
43     should be wildcard.
44
45     At first sight, this redundancy is just a waste of CPU
46     resources. But DPI and SPI add the possibility to assign different
47     priorities to GPIs. Look also at note 4 about tunnels below.
48
49
50     NOTE 3. One complication is the case of tunneled packets.
51     We implement it as following: if the first lookup
52     matches a special session with "tunnelhdr" value not zero,
53     flowid doesn't contain the true flow ID, but the tunnel ID (1..255).
54     In this case, we pull tunnelhdr bytes and restart lookup
55     with tunnel ID added to the list of keys. Simple and stupid 8)8)
56     It's enough for PIMREG and IPIP.
57
58
59     NOTE 4. Two GPIs make it possible to parse even GRE packets.
60     F.e. DPI can select ETH_P_IP (and necessary flags to make
61     tunnelhdr correct) in GRE protocol field and SPI matches
62     GRE key. Is it not nice? 8)8)
63
64
65     Well, as result, despite its simplicity, we get a pretty
66     powerful classification engine. */

```

Figure 851: RSVP Classifier Interface - Opening Comment:
linux/net/sched/cls_rsvp.h

```

70 struct rsvp_head
71 {
72     u32          tmap[256/32];
73     u32          hgenerator;
74     u8           tgenerator;
75     struct rsvp_session *ht [256];
76 };
77
78 struct rsvp_session
79 {
80     struct rsvp_session *next;
81     u32          dst [RSVP_DST_LEN];
82     struct tc_rsvp_gpi  dpi;
83     u8           protocol;
84     u8           tunnelid;
85     /* 16 (src, sport) hash slots, and one wildcard source slot */
86     struct rsvp_filter *ht[16+1];
87 };
88
89
90 struct rsvp_filter
91 {
92     struct rsvp_filter *next;
93     u32          src [RSVP_DST_LEN];
94     struct tc_rsvp_gpi  spi;
95     u8           tunnelhdr;
96
97     struct tcf_result  res;
98 #ifdef CONFIG_NET_CLS_POLICE
99     struct tcf_police *police;
100 #endif
101
102     u32          handle;
103     struct rsvp_session *sess;
104 };

```

Figure 852: RSVP - Structures: linux/net/sched/cls_rsvp.h

```

106 static __inline__ unsigned hash_dst(u32 *dst, u8 protocol, u8 tunnelid)
107 {
108     unsigned h = dst[RSVP_DST_LEN-1];
109     h ^= h>>16;
110     h ^= h>>8;
111     return (h ^ protocol ^ tunnelid) & 0xFF;
112 }
113
114 static __inline__ unsigned hash_src(u32 *src)
115 {
116     unsigned h = src[RSVP_DST_LEN-1];
117     h ^= h>>16;
118     h ^= h>>8;
119     h ^= h>>4;
120     return h & 0xF;
121 }
122
123 #ifdef CONFIG_NET_CLS_POLICE
124 #define RSVP_POLICE() \
125 if (f->police) { \
126     int pol_res = tcf_police(skb, f->police); \
127     if (pol_res < 0) continue; \
128     if (pol_res) return pol_res; \
129 }
130 #else
131 #define RSVP_POLICE()
132 #endif

```

Figure 853: RSVP - Police function: `linux/net/sched/cls_rsvp.h`

```

135 static int rsvp_classify (struct sk_buff *skb, struct tcf_proto *tp,
136                          struct tcf_result *res)
137 {
138     struct rsvp_session **sht = ((struct rsvp_head*)tp->root)->ht;
139     struct rsvp_session *s;
140     struct rsvp_filter *f;
141     unsigned h1, h2;
142     u32 *dst, *src;
143     u8 protocol;
144     u8 tunnelid = 0;
145     u8 *xprt;
146     #if RSVP_DST_LEN == 4
147         struct ipv6hdr *nhptr = skb->nh.ipv6h;
148     #else
149         struct iphdr *nhptr = skb->nh.iph;
150     #endif
151
152     #if !defined( __i386__ ) && !defined( _mc68000_ )
153         if ((unsigned long)nhptr & 3)
154             return -1;
155     #endif
156
157     restart :
158
159     #if RSVP_DST_LEN == 4
160         src = &nhptr->saddr.s6_addr32[0];
161         dst = &nhptr->daddr.s6_addr32[0];
162         protocol = nhptr->nexthdr;
163         xprt = ((u8*)nhptr) + sizeof(struct ipv6hdr);
164     #else
165         src = &nhptr->saddr;
166         dst = &nhptr->daddr;
167         protocol = nhptr->protocol;
168         xprt = ((u8*)nhptr) + (nhptr->ihl<<2);
169         if (nhptr->frag_off&_constant_htons(IP_MF|IP_OFFSET))
170             return -1;
171     #endif

```

Figure 854: RSVP - Classifier Function i: `linux/net/sched/cls_rsvp.h`


```

173     h1 = hash_dst(dst, protocol, tunnelid);
174     h2 = hash_src(src);
175
176     for (s = sht[h1]; s; s = s->next) {
177         if (dst[RSVP_DST_LEN-1] == s->dst[RSVP_DST_LEN-1] &&
178             protocol == s->protocol &&
179             !(s->dpi.mask & (*(u32*)(xprt+s->dpi.offset)^s->dpi.key))
180 #if RSVP_DST_LEN == 4
181             && dst[0] == s->dst[0]
182             && dst[1] == s->dst[1]
183             && dst[2] == s->dst[2]
184 #endif
185             && tunnelid == s->tunnelid) {
186
187                 for (f = s->ht[h2]; f; f = f->next) {
188                     if (src[RSVP_DST_LEN-1] == f->src[RSVP_DST_LEN
189                         -1] &&
190                         !(f->spi.mask & (*(u32*)(xprt+f->spi.offset)^f->
191                             spi.key))
192 #if RSVP_DST_LEN == 4
193                         && src[0] == f->src[0]
194                         && src[1] == f->src[1]
195                         && src[2] == f->src[2]
196 #endif
197                         ) {
198                             *res = f->res;
199                             RSVP_POLICE();
200 matched:
201                             if (f->tunnelhdr == 0)
202                                 return 0;
203
204                             tunnelid = f->res.classid;
205                             nhptr = (void*)(xprt + f->tunnelhdr - sizeof(*
206                                 nhptr));
207                             goto restart;
208                         }
209
210                     /* And wildcard bucket... */
211                     for (f = s->ht[16]; f; f = f->next) {
212                         *res = f->res;
213                         RSVP_POLICE();
214                         goto matched;
215                     }
216                     return -1;
217                 }
218             }
219     }
220 }

```

Figure 855: RSVP - Classifier Function ii: `linux/net/sched/cls_rsvp.h`

```

222 static unsigned long rsvp_get(struct tcf_proto *tp, u32 handle)
223 {
224     struct rsvp_session **sht = ((struct rsvp_head*)tp->root)->ht;
225     struct rsvp_session *s;
226     struct rsvp_filter *f;
227     unsigned h1 = handle&0xFF;
228     unsigned h2 = (handle>>8)&0xFF;
229
230     if (h2 > 16)
231         return 0;
232
233     for (s = sht[h1]; s; s = s->next) {
234         for (f = s->ht[h2]; f; f = f->next) {
235             if (f->handle == handle)
236                 return (unsigned long)f;
237         }
238     }
239     return 0;
240 }
241
242 static void rsvp_put(struct tcf_proto *tp, unsigned long f)
243 {
244 }
245
246 static int rsvp_init(struct tcf_proto *tp)
247 {
248     struct rsvp_head *data;
249
250     MOD_INC_USE_COUNT;
251     data = kmalloc(sizeof(struct rsvp_head), GFP_KERNEL);
252     if (data) {
253         memset(data, 0, sizeof(struct rsvp_head));
254         tp->root = data;
255         return 0;
256     }
257     MOD_DEC_USE_COUNT;
258     return -ENOBUFS;
259 }

```

Figure 856: RSVP - Misc Functions: `linux/net/sched/cls_rsvp.h`

```

261 static void rsvp_destroy(struct tcf_proto *tp)
262 {
263     struct rsvp_head *data = xchg(&tp->root, NULL);
264     struct rsvp_session **sht;
265     int h1, h2;
266
267     if (data == NULL)
268         return;
269
270     sht = data->ht;
271
272     for (h1=0; h1<256; h1++) {
273         struct rsvp_session *s;
274
275         while ((s = sht[h1]) != NULL) {
276             sht[h1] = s->next;
277
278             for (h2=0; h2<=16; h2++) {
279                 struct rsvp_filter *f;
280
281                 while ((f = s->ht[h2]) != NULL) {
282                     unsigned long cl;
283
284                     s->ht[h2] = f->next;
285                     if ((cl = _cls_set_class (&f->res.class, 0)) != 0)
286                         tp->q->ops->cl_ops->unbind_tcf(tp
287                             ->q, cl);
288
289                     #ifdef CONFIG_NET_CLS_POLICE
290                         tcf_police_release (f->police);
291                     #endif
292
293                     kfree(f);
294                 }
295             }
296             kfree(s);
297         }
298     }
299     kfree(data);
300     MOD_DEC_USE_COUNT;
301 }

```

Figure 857: RSVP - Delete: linux/net/sched/cls_rsvp.h

```

300 static int rsvp_delete(struct tcf_proto *tp, unsigned long arg)
301 {
302     struct rsvp_filter **fp, *f = (struct rsvp_filter *)arg;
303     unsigned h = f->handle;
304     struct rsvp_session **sp;
305     struct rsvp_session *s = f->sess;
306     int i;
307
308     for (fp = &s->ht[(h>>8)&0xFF]; *fp; fp = &(*fp)->next) {
309         if (*fp == f) {
310             unsigned long cl;
311
312
313             tcf_tree_lock (tp);
314             *fp = f->next;
315             tcf_tree_unlock (tp);
316
317             if ((cl = cls_set_class (tp, &f->res.class, 0)) != 0)
318                 tp->q->ops->cl_ops->unbind_tcf(tp->q, cl);
319
320 #ifdef CONFIG_NET_CLS_POLICE
321                 tcf_police_release (f->police);
322 #endif
323
324                 kfree(f);
325
326                 /* Strip tree */
327
328                 for (i=0; i<=16; i++)
329                     if (s->ht[i])
330                         return 0;
331
332                 /* OK, session has no flows */
333                 for (sp = &((struct rsvp_head*)tp->root)->ht[h&0xFF];
334                     *sp; sp = &(*sp)->next) {
335                     if (*sp == s) {
336                         tcf_tree_lock (tp);
337                         *sp = s->next;
338                         tcf_tree_unlock (tp);
339
340                         kfree(s);
341                         return 0;
342                     }
343                 }
344                 return 0;
345             }
346         }
347     }
348     return 0;
349 }

```

Figure 858: RSVP - Destroy: linux/net/sched/cls_rsvp.h

```

351 static unsigned gen_handle(struct tcf_proto *tp, unsigned salt)
352 {
353     struct rsvp_head *data = tp->root;
354     int i = 0xFFFF;
355
356     while (i-- > 0) {
357         u32 h;
358         if ((data->hgenerator += 0x10000) == 0)
359             data->hgenerator = 0x10000;
360         h = data->hgenerator|salt;
361         if (rsvp_get(tp, h) == 0)
362             return h;
363     }
364     return 0;
365 }
366
367 static int tunnel_bts(struct rsvp_head *data)
368 {
369     int n = data->tgenerator>>5;
370     u32 b = 1<<(data->tgenerator&0x1F);
371
372     if (data->tmap[n]&b)
373         return 0;
374     data->tmap[n] |= b;
375     return 1;
376 }

```

Figure 859: RSVP - general bits: `linux/net/sched/cls_rsvp.h`

```

378 static void tunnel_recycle(struct rsvp_head *data)
379 {
380     struct rsvp_session **sht = data->ht;
381     u32 tmap[256/32];
382     int h1, h2;
383
384     memset(tmap, 0, sizeof(tmap));
385
386     for (h1=0; h1<256; h1++) {
387         struct rsvp_session *s;
388         for (s = sht[h1]; s; s = s->next) {
389             for (h2=0; h2<=16; h2++) {
390                 struct rsvp_filter *f;
391
392                 for (f = s->ht[h2]; f; f = f->next) {
393                     if (f->tunnelhdr == 0)
394                         continue;
395                     data->tgenerator = f->res.classid;
396                     tunnel_bts(data);
397                 }
398             }
399         }
400     }
401
402     memcpy(data->tmap, tmap, sizeof(tmap));
403 }
404
405 static u32 gen_tunnel(struct rsvp_head *data)
406 {
407     int i, k;
408
409     for (k=0; k<2; k++) {
410         for (i=255; i>0; i--) {
411             if (++data->tgenerator == 0)
412                 data->tgenerator = 1;
413             if (tunnel_bts(data))
414                 return data->tgenerator;
415         }
416         tunnel_recycle(data);
417     }
418     return 0;
419 }

```

Figure 860: RSVP - recycle or generate tunnel: `linux/net/sched/cls_rsvp.h`

```

421 static int rsvp_change(struct tcf_proto *tp, unsigned long base,
422                       u32 handle,
423                       struct rtattr **tca,
424                       unsigned long *arg)
425 {
426     struct rsvp_head *data = tp->root;
427     struct rsvp_filter *f, **fp;
428     struct rsvp_session *s, **sp;
429     struct tc_rsvp_pinfo *pinfo = NULL;
430     struct rtattr *opt = tca[TCA_OPTIONS-1];
431     struct rtattr *tb[TCA_RSVP_MAX];
432     unsigned h1, h2;
433     u32 *dst;
434     int err;
435
436     if (opt == NULL)
437         return handle ? -EINVAL : 0;
438
439     if (rtattr_parse(tb, TCA_RSVP_MAX, RTA_DATA(opt), RTA_PAYLOAD(opt))
440         < 0)
441         return -EINVAL;

```

Figure 861: RSVP - Change Parameters i: `linux/net/sched/cls_rsvp.h`

```

442     if ((f = (struct rsvp_filter *)*arg) != NULL) {
443         /* Node exists: adjust only classid */
444
445         if (f->handle != handle && handle)
446             return -EINVAL;
447         if (tb[TCA_RSVP_CLASSID-1]) {
448             unsigned long cl;
449
450             f->res.classid = *(u32*)RTA_DATA(tb[TCA_RSVP_CLASSID
451             -1]);
452             cl = cls_set_class (tp, &f->res.class, tp->q->ops->cl_ops->
453             bind_tcf(tp->q, base, f->res.classid));
454             if (cl)
455                 tp->q->ops->cl_ops->unbind_tcf(tp->q, cl);
456         }
457     #ifdef CONFIG_NET_CLS_POLICE
458         if (tb[TCA_RSVP_POLICE-1]) {
459             struct tcf_police *police = tcf_police_locate (tb[
460             TCA_RSVP_POLICE-1], tca[TCA_RATE-1]);
461
462             tcf_tree_lock (tp);
463             police = xchg(&f->police, police);
464             tcf_tree_unlock (tp);
465
466             tcf_police_release (police);
467         }
468     #endif
469     return 0;
470 }
471
472 /* Now more serious part... */
473 if (handle)
474     return -EINVAL;
475 if (tb[TCA_RSVP_DST-1] == NULL)
476     return -EINVAL;
477
478 f = kmalloc(sizeof(struct rsvp_filter), GFP_KERNEL);
479 if (f == NULL)
480     return -ENOBUFS;

```

Figure 862: RSVP - Change Parameters ii: linux/net/sched/cls_rsvp.h


```

479     memset(f, 0, sizeof(*f));
480     h2 = 16;
481     if (tb[TCA_RSVP_SRC-1]) {
482         err = -EINVAL;
483         if (RTA_PAYLOAD(tb[TCA_RSVP_SRC-1]) != sizeof(f->src))
484             goto errout;
485         memcpy(f->src, RTA_DATA(tb[TCA_RSVP_SRC-1]), sizeof(f->src));
486         h2 = hash_src(f->src);
487     }
488     if (tb[TCA_RSVP_PINFO-1]) {
489         err = -EINVAL;
490         if (RTA_PAYLOAD(tb[TCA_RSVP_PINFO-1]) < sizeof(struct
491             tc_rsvp_pinfo))
492             goto errout;
493         pinfo = RTA_DATA(tb[TCA_RSVP_PINFO-1]);
494         f->spi = pinfo->spi;
495         f->tunnelhdr = pinfo->tunnelhdr;
496     }
497     if (tb[TCA_RSVP_CLASSID-1]) {
498         err = -EINVAL;
499         if (RTA_PAYLOAD(tb[TCA_RSVP_CLASSID-1]) != 4)
500             goto errout;
501         f->res.classid = *(u32*)RTA_DATA(tb[TCA_RSVP_CLASSID-1]);
502     }
503     err = -EINVAL;
504     if (RTA_PAYLOAD(tb[TCA_RSVP_DST-1]) != sizeof(f->src))
505         goto errout;
506     dst = RTA_DATA(tb[TCA_RSVP_DST-1]);
507     h1 = hash_dst(dst, pinfo ? pinfo->protocol : 0, pinfo ? pinfo->tunnelid : 0);
508
509     err = -ENOMEM;
510     if ((f->handle = gen_handle(tp, h1 | (h2<<8))) == 0)
511         goto errout;

```

Figure 863: RSVP - Change Parameters iii: `linux/net/sched/cls_rsvp.h`

```

513     if (f->tunnelhdr) {
514         err = -EINVAL;
515         if (f->res.classid > 255)
516             goto errout;
517
518         err = -ENOMEM;
519         if (f->res.classid == 0 &&
520             (f->res.classid = gen_tunnel(data)) == 0)
521             goto errout;
522     }
523
524     for (sp = &data->ht[h1]; (s=*sp) != NULL; sp = &s->next) {
525         if (dst[RSVP_DST_LEN-1] == s->dst[RSVP_DST_LEN-1] &&
526             pinfo->protocol == s->protocol &&
527             memcmp(&pinfo->dpi, &s->dpi, sizeof(s->dpi)) == 0
528 #if RSVP_DST_LEN == 4
529             && dst[0] == s->dst[0]
530             && dst[1] == s->dst[1]
531             && dst[2] == s->dst[2]
532 #endif
533             && pinfo->tunnelid == s->tunnelid) {
534
535         insert :
536
537             /* OK, we found appropriate session */
538
539             fp = &s->ht[h2];
540
541             f->sess = s;
542             if (f->tunnelhdr == 0)
543                 cls_set_class(tp, &f->res.class, tp->q->ops->cl_ops
544                     ->bind_tcf(tp->q, base, f->res.classid));
545 #ifdef CONFIG_NET_CLS_POLICE
546             if (tb[TCA_RSVP_POLICE-1])
547                 f->police = tcf_police_locate(tb[TCA_RSVP_POLICE
548                     -1], tca[TCA_RATE-1]);
549 #endif
550
551             for (fp = &s->ht[h2]; *fp; fp = &(*fp)->next)
552                 if (((*fp)->spi.mask & f->spi.mask) != f->spi.mask)
553                     break;
554
555             f->next = *fp;
556             wmb();
557             *fp = f;
558
559             *arg = (unsigned long)f;
560             return 0;
561         }
562     }
563 }

```

Figure 864: RSVP - Change Parameters iv: linux/net/sched/cls_rsvp.h

```

560     /* No session found. Create new one. */
561
562     err = -ENOBUFS;
563     s = kmalloc(sizeof(struct rsvp_session), GFP_KERNEL);
564     if (s == NULL)
565         goto errout;
566     memset(s, 0, sizeof(*s));
567     memcpy(s->dst, dst, sizeof(s->dst));
568     s->dpi = pinfo->dpi;
569     s->protocol = pinfo->protocol;
570     s->tunnelid = pinfo->tunnelid;
571     for (sp = &data->ht[h1]; *sp; sp = &(*sp)->next) {
572         if (((*sp)->dpi.mask & s->dpi.mask) != s->dpi.mask)
573             break;
574     }
575     s->next = *sp;
576     wmb();
577     *sp = s;
578
579     goto insert;
580
581 errout:
582     if (f)
583         kfree(f);
584     return err;
585 }

```

Figure 865: RSVP - Change Parameters v: linux/net/sched/cls_rsvp.h

```

587 static void rsvp_walk(struct tcf_proto *tp, struct tcf_walker *arg)
588 {
589     struct rsvp_head *head = tp->root;
590     unsigned h, h1;
591
592     if (arg->stop)
593         return;
594
595     for (h = 0; h < 256; h++) {
596         struct rsvp_session *s;
597
598         for (s = head->ht[h]; s; s = s->next) {
599             for (h1 = 0; h1 <= 16; h1++) {
600                 struct rsvp_filter *f;
601
602                 for (f = s->ht[h1]; f; f = f->next) {
603                     if (arg->count < arg->skip) {
604                         arg->count++;
605                         continue;
606                     }
607                     if (arg->fn(tp, (unsigned long)f, arg) < 0) {
608                         arg->stop = 1;
609                         break;
610                     }
611                     arg->count++;
612                 }
613             }
614         }
615     }
616 }

```

Figure 866: RSVP - Walk RSVP tree: `linux/net/sched/cls_rsvp.h`

```
41 #include "cls_rsvp.h"
```

Figure 867: RSVP Classifier API: `linux/net/sched/cls_rsvp.c`

```

38  /*
39  1. For now we assume that route tags < 256.
40     It allows to use direct table lookups, instead of hash tables.
41  2. For now we assume that "from TAG" and "fromdev DEV" statements
42     are mutually exclusive.
43  3. "to TAG from ANY" has higher priority, than "to ANY from XXX"
44  */
45
46  struct route4_fastmap
47  {
48      struct route4_filter    * filter ;
49      u32                      id ;
50      int                      iif ;
51  };
52
53  struct route4_head
54  {
55      struct route4_fastmap    fastmap[16];
56      struct route4_bucket     *table[256+1];
57  };
58
59  struct route4_bucket
60  {
61      struct route4_filter     *ht[16+16+1];
62  };
63
64  struct route4_filter
65  {
66      struct route4_filter     *next;
67      u32                      id;
68      int                      iif ;
69
70      struct tcf_result        res;
71  #ifdef CONFIG_NET_CLS_POLICE
72      struct tcf_police        *police;
73  #endif
74
75      u32                      handle;
76      struct route4_bucket     *bkt;
77  };
78
79  #define ROUTE4_FAILURE ((struct route4_filter*)(-1L))

```

Figure 868: Route Classifier API - Structures: `linux/net/sched/cls_route.c`

```

81 static __inline__ int route4_fastmap_hash(u32 id, int iif)
82 {
83     return id&0xF;
84 }
85
86 static void route4_reset_fastmap(struct net_device *dev, struct route4_head *head, u32 id
87 )
88 {
89     spin_lock_bh(&dev->queue_lock);
90     memset(head->fastmap, 0, sizeof(head->fastmap));
91     spin_unlock_bh(&dev->queue_lock);
92 }
93
94 static void __inline__
95 route4_set_fastmap(struct route4_head *head, u32 id, int iif,
96                   struct route4_filter *f)
97 {
98     int h = route4_fastmap_hash(id, iif);
99     head->fastmap[h].id = id;
100    head->fastmap[h].iif = iif;
101    head->fastmap[h].filter = f;
102 }
103
104 static __inline__ int route4_hash_to(u32 id)
105 {
106     return id&0xFF;
107 }
108
109 static __inline__ int route4_hash_from(u32 id)
110 {
111     return (id>>16)&0xF;
112 }
113
114 static __inline__ int route4_hash_iif(int iif)
115 {
116     return 16 + ((iif >>16)&0xF);
117 }
118
119 static __inline__ int route4_hash_wild(void)
120 {
121     return 32;
122 }
123
124 #ifdef CONFIG_NET_CLS_POLICE
125 #define IF_ROUTE_POLICE \
126 if (f->police) { \
127     int pol_res = tcf_police(skb, f->police); \
128     if (pol_res >= 0) return pol_res; \
129     dont_cache = 1; \
130     continue; \
131 } \
132 if (!dont_cache)
133 #define IF_ROUTE_POLICE
134 #endif

```

Figure 869: Route Classifier API - Functions including Policer:
linux/net/sched/cls_route.c

```

137 static int route4_classify (struct sk_buff *skb, struct tcf_proto *tp,
138                          struct tcf_result *res)
139 {
140     struct route4_head *head = (struct route4_head*)tp->root;
141     struct dst_entry *dst;
142     struct route4_bucket *b;
143     struct route4_filter *f;
144     #ifdef CONFIG_NET_CLS_POLICE
145     int dont_cache = 0;
146     #endif
147     u32 id, h;
148     int iif;
149
150     if ((dst = skb->dst) == NULL)
151         goto failure;
152
153     id = dst->tclassid;
154     if (head == NULL)
155         goto old_method;
156
157     iif = ((struct rtable*)dst)->key.iif;
158
159     h = route4_fastmap_hash(id, iif);
160     if (id == head->fastmap[h].id &&
161         iif == head->fastmap[h].iif &&
162         (f = head->fastmap[h].filter) != NULL) {
163         if (f == ROUTE4_FAILURE)
164             goto failure;
165
166         *res = f->res;
167         return 0;
168     }
169
170     h = route4_hash_to(id);

```

Figure 870: Route Classifier API - Classifier Function i:
linux/net/sched/cls_route.c


```

172 restart :
173     if ((b = head->table[h]) != NULL) {
174         f = b->ht[route4_hash_from(id)];
175
176         for (; f; f = f->next) {
177             if (f->id == id) {
178                 *res = f->res;
179                 IF_ROUTE_POLICE route4_set_fastmap(head, id, iif, f);
180                 return 0;
181             }
182         }
183
184         for (f = b->ht[route4_hash_iif(iif)]; f; f = f->next) {
185             if (f->iif == iif) {
186                 *res = f->res;
187                 IF_ROUTE_POLICE route4_set_fastmap(head, id, iif, f);
188                 return 0;
189             }
190         }
191
192         for (f = b->ht[route4_hash_wild()]; f; f = f->next) {
193             *res = f->res;
194             IF_ROUTE_POLICE route4_set_fastmap(head, id, iif, f);
195             return 0;
196         }
197     }
198 }
199 if (h < 256) {
200     h = 256;
201     id &= ~0xFFFF;
202     goto restart ;
203 }

```

Figure 871: Route Classifier API - Classifier Function ii:
linux/net/sched/cls_route.c

```

204
205 #ifdef CONFIG_NET_CLS_POLICE
206     if (!dont_cache)
207 #endif
208         route4_set_fastmap(head, id, iif, ROUTE4_FAILURE);
209 failure:
210     return -1;
211
212 old_method:
213     if (id && (TC_H_MAJ(id) == 0 ||
214             !(TC_H_MAJ(id) ^ tp->q->handle))) {
215         res->classid = id;
216         res->class = 0;
217         return 0;
218     }
219     return -1;
220 }

```

Figure 872: Route Classifier API - Classifier Function iii:
linux/net/sched/cls_route.c

```

222 static u32 to_hash(u32 id)
223 {
224     u32 h = id&0xFF;
225     if (id&0x8000)
226         h += 256;
227     return h;
228 }
229
230 static u32 from_hash(u32 id)
231 {
232     id &= 0xFFFF;
233     if (id == 0xFFFF)
234         return 32;
235     if (!(id & 0x8000)) {
236         if (id > 255)
237             return 256;
238         return id&0xF;
239     }
240     return 16 + (id&0xF);
241 }
242
243 static unsigned long route4_get(struct tcf_proto *tp, u32 handle)
244 {
245     struct route4_head *head = (struct route4_head*)tp->root;
246     struct route4_bucket *b;
247     struct route4_filter *f;
248     unsigned h1, h2;
249
250     if (!head)
251         return 0;
252
253     h1 = to_hash(handle);
254     if (h1 > 256)
255         return 0;
256
257     h2 = from_hash(handle>>16);
258     if (h2 > 32)
259         return 0;
260
261     if ((b = head->table[h1]) != NULL) {
262         for (f = b->ht[h2]; f; f = f->next)
263             if (f->handle == handle)
264                 return (unsigned long)f;
265     }
266     return 0;
267 }
268
269 static void route4_put(struct tcf_proto *tp, unsigned long f)
270 {
271 }
272
273 static int route4_init(struct tcf_proto *tp)
274 {
275     MOD_INC_USE_COUNT;
276     return 0;
277 }

```

Figure 873: Route Classifier API - Misc: `linux/net/sched/cls_route.c`

```

279 static void route4_destroy(struct tcf_proto *tp)
280 {
281     struct route4_head *head = xchg(&tp->root, NULL);
282     int h1, h2;
283
284     if (head == NULL) {
285         MOD_DEC_USE_COUNT;
286         return;
287     }
288
289     for (h1=0; h1<=256; h1++) {
290         struct route4_bucket *b;
291
292         if ((b = head->table[h1]) != NULL) {
293             for (h2=0; h2<=32; h2++) {
294                 struct route4_filter *f;
295
296                 while ((f = b->ht[h2]) != NULL) {
297                     unsigned long cl;
298
299                     b->ht[h2] = f->next;
300                     if ((cl = _cls_set_class (&f->res.class, 0)) != 0)
301                         tp->q->ops->cl_ops->unbind_tcf(tp
302                             ->q, cl);
303
304                     #ifdef CONFIG_NET_CLS_POLICE
305                         tcf_police_release (f->police);
306                     #endif
307
308                     kfree(f);
309                 }
310             }
311             kfree(b);
312         }
313     }
314     kfree(head);
315     MOD_DEC_USE_COUNT;
316 }

```

Figure 874: Route Classifier API - Destroy Route Classifier:
linux/net/sched/cls_route.c

```

315 static int route4_delete(struct tcf_proto *tp, unsigned long arg)
316 {
317     struct route4_head *head = (struct route4_head*)tp->root;
318     struct route4_filter **fp, *f = (struct route4_filter *)arg;
319     unsigned h = f->handle;
320     struct route4_bucket *b;
321     int i;
322
323     if (!head || !f)
324         return -EINVAL;
325
326     b = f->bkt;
327
328     for (fp = &b->ht[from_hash(h>>16)]; *fp; fp = &(*fp)->next) {
329         if (*fp == f) {
330             unsigned long cl;
331
332             tcf_tree_lock (tp);
333             *fp = f->next;
334             tcf_tree_unlock (tp);
335
336             route4_reset_fastmap(tp->q->dev, head, f->id);
337
338             if ((cl = cls_set_class (tp, &f->res.class, 0)) != 0)
339                 tp->q->ops->cl_ops->unbind_tcf(tp->q, cl);
340
341 #ifdef CONFIG_NET_CLS_POLICE
342                 tcf_police_release (f->police);
343 #endif
344                 kfree(f);
345
346                 /* Strip tree */
347
348                 for (i=0; i<=32; i++)
349                     if (b->ht[i])
350                         return 0;
351
352                 /* OK, session has no flows */
353                 tcf_tree_lock (tp);
354                 head->table[to_hash(h)] = NULL;
355                 tcf_tree_unlock (tp);
356
357                 kfree(b);
358                 return 0;
359             }
360         }
361     }
362     return 0;

```

Figure 875: Route Classifier API - Delete Entry:
linux/net/sched/cls_route.c

```

43 struct fw_head
44 {
45     struct fw_filter *ht [256];
46 };
47
48 struct fw_filter
49 {
50     struct fw_filter *next;
51     u32 id;
52     struct tcf_result res;
53 #ifdef CONFIG_NET_CLS_POLICE
54     struct tcf_police *police;
55 #endif
56 };
57
58 static __inline__ int fw_hash(u32 handle)
59 {
60     return handle&0xFF;
61 }

```

Figure 876: Firewall Classifier API - Structures: `linux/net/sched/cls_fw.c`

```

63 static int fw_classify (struct sk_buff *skb, struct tcf_proto *tp,
64                       struct tcf_result *res)
65 {
66     struct fw_head *head = (struct fw_head*)tp->root;
67     struct fw_filter *f;
68     #ifdef CONFIG_NETFILTER
69         u32 id = skb->nfmark;
70     #else
71         u32 id = 0;
72     #endif
73
74     if (head == NULL)
75         goto old_method;
76
77     for (f=head->ht[fw_hash(id)]; f=f->next) {
78         if (f->id == id) {
79             *res = f->res;
80             #ifdef CONFIG_NET_CLS_POLICE
81                 if (f->police)
82                     return tcf_police (skb, f->police);
83             #endif
84             return 0;
85         }
86     }
87     return -1;
88
89 old_method:
90     if (id && (TC_H_MAJ(id) == 0 ||
91              !(TC_H_MAJ(id)*tp->q->handle))) {
92         res->classid = id;
93         res->class = 0;
94         return 0;
95     }
96     return -1;
97 }

```

Figure 877: Firewall Classifier API - Classifier: `linux/net/sched/cls_fw.c`

```

99 static unsigned long fw_get(struct tcf_proto *tp, u32 handle)
100 {
101     struct fw_head *head = (struct fw_head*)tp->root;
102     struct fw_filter *f;
103
104     if (head == NULL)
105         return 0;
106
107     for (f=head->ht[fw_hash(handle)]; f=f->next) {
108         if (f->id == handle)
109             return (unsigned long)f;
110     }
111     return 0;
112 }
113
114 static void fw_put(struct tcf_proto *tp, unsigned long f)
115 {
116 }
117
118 static int fw_init(struct tcf_proto *tp)
119 {
120     MOD_INC_USE_COUNT;
121     return 0;
122 }

```

Figure 878: Firewall Classifier API - Misc: `linux/net/sched/cls_fw.c`


```

124 static void fw_destroy(struct tcf_proto *tp)
125 {
126     struct fw_head *head = (struct fw_head*)xchg(&tp->root, NULL);
127     struct fw_filter *f;
128     int h;
129
130     if (head == NULL) {
131         MOD_DEC_USE_COUNT;
132         return;
133     }
134
135     for (h=0; h<256; h++) {
136         while ((f=head->ht[h]) != NULL) {
137             unsigned long cl;
138             head->ht[h] = f->next;
139
140             if ((cl = _cls_set_class (&f->res.class, 0)) != 0)
141                 tp->q->ops->cl_ops->unbind_tcf(tp->q, cl);
142 #ifdef CONFIG_NET_CLS_POLICE
143                 tcf_police_release (f->police);
144 #endif
145                 kfree(f);
146         }
147     }
148     kfree(head);
149     MOD_DEC_USE_COUNT;
150 }

```

Figure 879: Firewall Classifier API - Destroy Classifier:
linux/net/sched/cls_fw.c

```

152 static int fw_delete(struct tcf_proto *tp, unsigned long arg)
153 {
154     struct fw_head *head = (struct fw_head*)tp->root;
155     struct fw_filter *f = (struct fw_filter *)arg;
156     struct fw_filter **fp;
157
158     if (head == NULL || f == NULL)
159         return -EINVAL;
160
161     for (fp=&head->ht[fw_hash(f->id)]; *fp; fp = &(*fp)->next) {
162         if (*fp == f) {
163             unsigned long cl;
164
165             tcf_tree_lock (tp);
166             *fp = f->next;
167             tcf_tree_unlock (tp);
168
169             if ((cl = cls_set_class (tp, &f->res.class, 0)) != 0)
170                 tp->q->ops->cl_ops->unbind_tcf(tp->q, cl);
171 #ifdef CONFIG_NET_CLS_POLICE
172                 tcf_police_release (f->police);
173 #endif
174                 kfree(f);
175                 return 0;
176             }
177         }
178     return -EINVAL;
179 }

```

Figure 880: Firewall Classifier API Delete Classifier:
linux/net/sched/cls_fw.c

```

181 static int fw_change(struct tcf_proto *tp, unsigned long base,
182                    u32 handle,
183                    struct rtattr **tca,
184                    unsigned long *arg)
185 {
186     struct fw_head *head = (struct fw_head*)tp->root;
187     struct fw_filter *f;
188     struct rtattr *opt = tca[TCA_OPTIONS-1];
189     struct rtattr *tb[TCA_FW_MAX];
190     int err;
191
192     if (!opt)
193         return handle ? -EINVAL : 0;
194
195     if (rtattr_parse(tb, TCA_FW_MAX, RTA_DATA(opt), RTA_PAYLOAD(opt))
196         < 0)
197         return -EINVAL;

```

Figure 881: Firewall Classifier API - Change Classifier:
linux/net/sched/cls_fw.c

```

197
198     if ((f = (struct fw_filter *)arg) != NULL) {
199         /* Node exists: adjust only classid */
200
201         if (f->id != handle && handle)
202             return -EINVAL;
203         if (tb[TCA_FW_CLASSID-1]) {
204             unsigned long cl;
205
206             f->res.classid = *(u32*)RTA_DATA(tb[TCA_FW_CLASSID-1]);
207             cl = tp->q->ops->cl_ops->bind_tcf(tp->q, base, f->res.
                classid);
208             cl = cls_set_class (tp, &f->res.class, cl);
209             if (cl)
210                 tp->q->ops->cl_ops->unbind_tcf(tp->q, cl);
211         }
212 #ifdef CONFIG_NET_CLS_POLICE
213         if (tb[TCA_FW_POLICE-1]) {
214             struct tcf_police *police = tcf_police_locate (tb[
                TCA_FW_POLICE-1], tca[TCA_RATE-1]);
215
216             tcf_tree_lock (tp);
217             police = xchg(&f->police, police);
218             tcf_tree_unlock (tp);
219
220             tcf_police_release (police);
221         }
222 #endif
223         return 0;
224     }

```

Figure 882: Firewall Classifier API - Change Classifier ii:
linux/net/sched/cls_fw.c

```

226     if (!handle)
227         return -EINVAL;
228
229     if (head == NULL) {
230         head = kmalloc(sizeof(struct fw_head), GFP_KERNEL);
231         if (head == NULL)
232             return -ENOBUFS;
233         memset(head, 0, sizeof(*head));
234
235         tcf_tree_lock(tp);
236         tp->root = head;
237         tcf_tree_unlock(tp);
238     }
239
240     f = kmalloc(sizeof(struct fw_filter), GFP_KERNEL);
241     if (f == NULL)
242         return -ENOBUFS;
243     memset(f, 0, sizeof(*f));
244
245     f->id = handle;

```

Figure 883: Firewall Classifier API - Change Classifier iii:
linux/net/sched/cls_fw.c

```

247     if (tb[TCA_FW_CLASSID-1]) {
248         err = -EINVAL;
249         if (RTA_PAYLOAD(tb[TCA_FW_CLASSID-1]) != 4)
250             goto errout;
251         f->res.classid = *(u32*)RTA_DATA(tb[TCA_FW_CLASSID-1]);
252         cls_set_class (tp, &f->res.class, tp->q->ops->cl_ops->bind_tcf(tp->q
                , base, f->res.classid));
253     }
254
255 #ifdef CONFIG_NET_CLS_POLICE
256     if (tb[TCA_FW_POLICE-1])
257         f->police = tcf_police_locate(tb[TCA_FW_POLICE-1], tca[TCA_RATE
                -1]);
258 #endif
259
260     f->next = head->ht[fw_hash(handle)];
261     tcf_tree_lock (tp);
262     head->ht[fw_hash(handle)] = f;
263     tcf_tree_unlock (tp);
264
265     *arg = (unsigned long)f;
266     return 0;
267
268 errout:
269     if (f)
270         kfree(f);
271     return err;
272 }

```

Figure 884: Firewall Classifier API - Change Classifier iv:
linux/net/sched/cls_fw.c

```

19  /*
20  * Not quite sure if we need all the xchgs Alexey uses when accessing things.
21  * Can always add them later ... :)
22  */
23
24  /*
25  * Passing parameters to the root seems to be done more awkwardly than really
26  * necessary. At least, u32 doesn't seem to use such dirty hacks. To be
27  * verified. FIXME.
28  */
29
30  #define PERFECT_HASH_THRESHOLD 64 /* use perfect hash if not bigger */
31  #define DEFAULT_HASH_SIZE 64 /* optimized for diffserv */
32
33
34  #if 1 /* control */
35  #define DPRINTK(format,args...) printk(KERN_DEBUG format,##args)
36  #else
37  #define DPRINTK(format,args...)
38  #endif
39
40  #if 0 /* data */
41  #define D2PRINTK(format,args...) printk(KERN_DEBUG format,##args)
42  #else
43  #define D2PRINTK(format,args...)
44  #endif
45
46
47  #define PRIV(tp) ((struct tcindex_data *) (tp)->root)
48
49
50  struct tcindex_filter_result {
51      struct tcf_police *police;
52      struct tcf_result res;
53  };
54
55  struct tcindex_filter {
56      _u16 key;
57      struct tcindex_filter_result result;
58      struct tcindex_filter *next;
59  };
60
61
62  struct tcindex_data {
63      struct tcindex_filter_result *perfect; /* perfect hash; NULL if none */
64      struct tcindex_filter **h; /* imperfect hash; only used if !perfect;
65      NULL if unused */
66      _u16 mask; /* AND key with mask */
67      int shift; /* shift ANDed key to the right */
68      int hash; /* hash table size; 0 if undefined */
69      int alloc_hash; /* allocated size */
70      int fall_through; /* 0: only classify if explicit match */
71  };

```

Figure 885: TC Classifier API - Defines: linux/net/sched/cls_tcindex.c

```

74 static struct tcindex_filter_result *lookup(struct tcindex_data *p, _u16 key)
75 {
76     struct tcindex_filter *f;
77
78     if (p->perfect)
79         return p->perfect[key].res.class ? p->perfect+key : NULL;
80     if (!p->h)
81         return NULL;
82     for (f = p->h[key % p->hash]; f; f = f->next) {
83         if (f->key == key)
84             return &f->result;
85     }
86     return NULL;
87 }

```

Figure 886: TC Classifier API - lookup: `linux/net/sched/cls_tcindex.c`


```

90 static int tcindex_classify(struct sk_buff *skb, struct tcf_proto *tp,
91                          struct tcf_result *res)
92 {
93     struct tcindex_data *p = PRIV(tp);
94     struct tcindex_filter_result *f;
95
96     D2PRINTK("tcindex_classify(skb=%p,tp=%p,res=%p),p=%p\n",skb,tp,res,p);
97
98     f = lookup(p,(skb->tc_index & p->mask) >> p->shift);
99     if (!f) {
100         if (!p->fall_through)
101             return -1;
102         res->classid = TC_H_MAKE(TC_H_MAJ(tp->q->handle),
103                                (skb->tc_index & p->mask) >> p->shift);
104         res->class = 0;
105         D2PRINTK("alg_0x%x\n",res->classid);
106         return 0;
107     }
108     *res = f->res;
109     D2PRINTK("map_0x%x\n",res->classid);
110 #ifdef CONFIG_NET_CLS_POLICE
111     if (f->police) {
112         int result;
113
114         result = tcf_police(skb,f->police);
115         D2PRINTK("police_%d\n",res);
116         return result;
117     }
118 #endif
119     return 0;
120 }

```

Figure 887: TC Classifier API - Classifier: `linux/net/sched/cls_tcindex.c`

```

122
123 static unsigned long tcindex_get(struct tcf_proto *tp, u32 handle)
124 {
125     struct tcindex_data *p = PRIV(tp);
126     struct tcindex_filter_result *r;
127
128     DPRINTK("tcindex_get(tp=%p,handle=0x%08x)\n",tp,handle);
129     if (p->perfect && handle >= p->alloc_hash)
130         return 0;
131     r = lookup(PRIV(tp),handle);
132     return r && r->res.class ? (unsigned long) r : 0;
133 }
134
135
136 static void tcindex_put(struct tcf_proto *tp, unsigned long f)
137 {
138     DPRINTK("tcindex_put(tp=%p,f=0x%lx)\n",tp,f);
139 }
140
141
142 static int tcindex_init(struct tcf_proto *tp)
143 {
144     struct tcindex_data *p;
145
146     DPRINTK("tcindex_init(tp=%p)\n",tp);
147     MOD_INC_USE_COUNT;
148     p = kmalloc(sizeof(struct tcindex_data),GFP_KERNEL);
149     if (!p) {
150         MOD_DEC_USE_COUNT;
151         return -ENOMEM;
152     }
153     tp->root = p;
154     p->perfect = NULL;
155     p->h = NULL;
156     p->hash = 0;

```

Figure 888: TC Classifier API - Misc: linux/net/sched/cls_tcindex.c

```

158     p->shift = 0;
159     p->fallthrough = 1;
160     return 0;
161 }
162
163
164 static int tcindex_delete(struct tcf_proto *tp, unsigned long arg)
165 {
166     struct tcindex_data *p = PRIV(tp);
167     struct tcindex_filter_result *r = (struct tcindex_filter_result *) arg;
168     struct tcindex_filter *f = NULL;
169     unsigned long cl;
170
171     DPRINTK("tcindex_delete(tp,%p,arg,0x%lx),p,%p,f,%p\n",tp,arg,p,f);
172     if (p->perfect) {
173         if (!r->res.class)
174             return -ENOENT;
175     } else {
176         int i;
177         struct tcindex_filter **walk = NULL;
178
179         for (i = 0; i < p->hash; i++)
180             for (walk = p->h+i; *walk; walk = &(*walk)->next)
181                 if (&(*walk)->result == r)
182                     goto found;
183         return -ENOENT;
184
185 found:
186         f = *walk;
187         tcf_tree_lock(tp);
188         *walk = f->next;
189         tcf_tree_unlock(tp);
190     }
191     cl = _cls_set_class(&r->res.class,0);
192     if (cl)
193         tp->q->ops->cl_ops->unbind_tcf(tp->q,cl);
194 #ifdef CONFIG_NET_CLS_POLICE

```

Figure 889: TC Classifier API - delete classifier:
linux/net/sched/cls_tcindex.c

```

377     }
378   }
379   if (!p->h)
380     return;
381   for (i = 0; i < p->hash; i++) {
382     for (f = p->h[i]; f; f = next) {
383       next = f->next;
384       if (walker->count >= walker->skip) {
385         if (walker->fn(tp,(unsigned long) &f->result,
386           walker) < 0) {
387           walker->stop = 1;
388           return;
389         }
390       }
391       walker->count++;
392     }
393   }
394 }
395
396
397 static int tcindex_destroy_element(struct tcf_proto *tp,
398   unsigned long arg, struct tcf_walker *walker)
399 {
400     return tcindex_delete(tp,arg);
401 }

```

Figure 890: TC Classifier API - Destroy Classifier:
 linux/net/sched/cls_tcindex.c

```

202
203 /*
204  * There are no parameters for tcindex_init, so we overload tcindex_change
205  */
206
207
208 static int tcindex_change(struct tcf_proto *tp,unsigned long base,u32 handle,
209      struct rtattr **tca,unsigned long *arg)
210 {
211     struct tcindex_filter_result new_filter_result = {
212         NULL,          /* no policing */
213         { 0,0 },      /* no classification */
214     };
215     struct rtattr *opt = tca[TCA_OPTIONS-1];
216     struct rtattr *tb[TCA_TCINDEX_MAX];
217     struct tcindex_data *p = PRIV(tp);
218     struct tcindex_filter *f;
219     struct tcindex_filter_result *r = (struct tcindex_filter_result *) *arg;
220     struct tcindex_filter **walk;
221     int hash, shift ;
222     __u16 mask;
223
224     DPRINTK("tcindex_change(tp,%p,handle,0x%08x,tca,%p,arg,%p),opt,%p,"
225            "p,%p,r,%p\n",tp,handle,tca,arg,opt,p,r);
226     if (arg)
227         DPRINTK("*arg=%0x%lx\n",*arg);
228     if (!opt)
229         return 0;
230     if ( rtattr_parse (tb,TCA_TCINDEX_MAX,RTA_DATA(opt),RTA_PAYLOAD(opt))
231          < 0)
232         return -EINVAL;
233     if (!tb[TCA_TCINDEX_HASH-1]) {
234         hash = p->hash;
235     } else {
236         if (RTA_PAYLOAD(tb[TCA_TCINDEX_HASH-1]) < sizeof(int))
237             return -EINVAL;
238         hash = *(int *) RTA_DATA(tb[TCA_TCINDEX_HASH-1]);
239     }
240     if (!tb[TCA_TCINDEX_MASK-1]) {
241         mask = p->mask;
242     } else {
243         if (RTA_PAYLOAD(tb[TCA_TCINDEX_MASK-1]) < sizeof(__u16))
244             return -EINVAL;

```

Figure 891: TC Classifier API - change classifier i:
linux/net/sched/cls_tcindex.c

```

244         mask = *(_u16 *) RTA_DATA(tb[TCA_TCINDEX_MASK-1]);
245     }
246     if (!tb[TCA_TCINDEX_SHIFT-1])
247         shift = p->shift;
248     else {
249         if (RTA_PAYLOAD(tb[TCA_TCINDEX_SHIFT-1]) < sizeof(_u16))
250             return -EINVAL;
251         shift = *(int *) RTA_DATA(tb[TCA_TCINDEX_SHIFT-1]);
252     }
253     if (p->perfect && hash <= (mask >> shift))
254         return -EBUSY;
255     if (p->perfect && hash > p->alloc_hash)
256         return -EBUSY;
257     if (p->h && hash != p->alloc_hash)
258         return -EBUSY;
259     p->hash = hash;
260     p->mask = mask;
261     p->shift = shift;
262     if (tb[TCA_TCINDEX_FALL_THROUGH-1]) {
263         if (RTA_PAYLOAD(tb[TCA_TCINDEX_FALL_THROUGH-1]) < sizeof(
                int))
264             return -EINVAL;
265         p->fall_through =
266             *(int *) RTA_DATA(tb[TCA_TCINDEX_FALL_THROUGH-1]);
267     }

```

Figure 892: TC Classifier API - change classifier ii:
linux/net/sched/cls_tcindex.c

```

268     DPRINTK("classid/police_%p/%p\n",tb[TCA_TCINDEX_CLASSID-1],
269            tb[TCA_TCINDEX_POLICE-1]);
270     if (!tb[TCA_TCINDEX_CLASSID-1] && !tb[TCA_TCINDEX_POLICE-1])
271         return 0;
272     if (!hash) {
273         if ((mask >> shift) < PERFECT_HASH_THRESHOLD) {
274             p->hash = (mask >> shift)+1;
275         } else {
276             p->hash = DEFAULT_HASH_SIZE;
277         }
278     }
279     if (!p->perfect && !p->h) {
280         p->alloc_hash = p->hash;
281         DPRINTK("hash_%d_mask_%d\n",p->hash,p->mask);
282         if (p->hash > (mask >> shift)) {
283             p->perfect = kmalloc(p->hash*
284                                sizeof(struct tcindex_filter_result ),GFP_KERNEL);
285             if (!p->perfect)
286                 return -ENOMEM;
287             memset(p->perfect, 0,
288                   p->hash * sizeof(struct tcindex_filter_result ));
289         } else {
290             p->h = kmalloc(p->hash*sizeof(struct tcindex_filter *),
291                           GFP_KERNEL);
292             if (!p->h)
293                 return -ENOMEM;
294             memset(p->h, 0, p->hash*sizeof(struct tcindex_filter *));
295         }
296     }

```

Figure 893: TC Classifier API - change classifier iii:
linux/net/sched/cls_tcindex.c

```

297     /*
298     * Note: this could be as restrictive as
299     * if (handle & ~(mask >> shift))
300     * but then, we'd fail handles that may become valid after some
301     * future mask change. While this is extremely unlikely to ever
302     * matter, the check below is safer (and also more
303     * backwards-compatible).
304     */
305     if (p->perfect && handle >= p->alloc_hash)
306         return -EINVAL;
307     if (p->perfect) {
308         r = p->perfect+handle;
309     } else {
310         r = lookup(p,handle);
311         DPRINTK("r=%p\n",r);
312         if (!r)
313             r = &new_filter_result ;
314     }
315     DPRINTK("r=%p\n",r);
316     if (tb[TCA_TCINDEX_CLASSID-1]) {
317         unsigned long cl = cls_set_class (tp,&r->res.class,0);
318
319         if (cl)
320             tp->q->ops->cl_ops->unbind_tcf(tp->q,cl);
321         r->res.classid = *(_u32 *) RTA_DATA(tb[TCA_TCINDEX_CLASSID-1])
322             ;
323         r->res.class = tp->q->ops->cl_ops->bind_tcf(tp->q,base,
324             r->res.classid);
325         if (!r->res.class) {
326             r->res.classid = 0;
327             return -ENOENT;
328         }
329     }
330 #ifdef CONFIG_NET_CLS_POLICE
331     {
332         struct tcf_police *police;
333
334         police = tb[TCA_TCINDEX_POLICE-1] ?
335             tcf_police_locate (tb[TCA_TCINDEX_POLICE-1],NULL) : NULL;
336         tcf_tree_lock (tp);
337         police = xchg(&r->police,police);

```

Figure 894: TC Classifier API - change classifier iii: linux/net/sched/cls_tcindex.c


```

51  /*
52
53  Short review.
54  -----
55
56  This file consists of two interrelated parts:
57
58  1. queueing disciplines manager frontend.
59  2. traffic classes manager frontend.
60
61  Generally, queueing discipline ("qdisc") is a black box,
62  which is able to enqueue packets and to dequeue them (when
63  device is ready to send something) in order and at times
64  determined by algorithm hidden in it.
65
66  qdisc's are divided to two categories:
67  - "queues", which have no internal structure visible from outside.
68  - "schedulers", which split all the packets to "traffic classes",
69    using "packet classifiers" (look at cls_api.c)
70
71  In turn, classes may have child qdiscs (as rule, queues)
72  attached to them etc. etc. etc.
73
74  The goal of the routines in this file is to translate
75  information supplied by user in the form of handles
76  to more intelligible for kernel form, to make some sanity
77  checks and part of work, which is common to all qdiscs
78  and to provide rtnetlink notifications.
79
80  All real intelligent work is done inside qdisc modules.
81
82
83
84  Every discipline has two major routines: enqueue and dequeue.
85
86  ---dequeue
87
88  dequeue usually returns a skb to send. It is allowed to return NULL,
89  but it does not mean that queue is empty, it just means that
90  discipline does not want to send anything this time.
91  Queue is really empty if q->q.qlen == 0.
92  For complicated disciplines with multiple queues q->q is not
93  real packet queue, but however q->q.qlen must be valid.
94
95  ---enqueue
96
97  enqueue returns 0, if packet was enqueued successfully.
98  If packet (this one or another one) was dropped, it returns
99  not zero error code.
100 NET_XMIT_DROP      - this packet dropped
101   Expected action: do not backoff, but wait until queue will clear.
102 NET_XMIT_CN        - probably this packet enqueued, but another one dropped.
103   Expected action: backoff or ignore
104 NET_XMIT_POLICED   - dropped by police.
105   Expected action: backoff or error to real-time apps.

```

Figure 895: Scheduler API i: linux/net/sched/sch_api.c

```

106
107 Auxiliary routines:
108
109 ---requeue
110
111 requeues once dequeued packet. It is used for non-standard or
112 just buggy devices, which can defer output even if dev->tbusy=0.
113
114 ---reset
115
116 returns qdisc to initial state: purge all buffers, clear all
117 timers, counters (except for statistics) etc.
118
119 ---init
120
121 initializes newly created qdisc.
122
123 ---destroy
124
125 destroys resources allocated by init and during lifetime of qdisc.
126
127 ---change
128
129 changes qdisc parameters.
130 */
131
132 /* Protects list of registered TC modules. It is pure SMP lock. */
133 static rwlock_t qdisc_mod_lock = RW_LOCK_UNLOCKED;

```

Figure 896: Scheduler API auxiliary routines: `linux/net/sched/sch_api.c`

```
136 /*****
137  *   Queueing disciplines manipulation. *
138  *****/
139
140
141 /* The list of all installed queueing disciplines. */
142
143 static struct Qdisc_ops *qdisc_base = NULL;
```

Figure 897: Scheduler API all the disciplines: `linux/net/sched/sch_api.c`

```

147 int register_qdisc (struct Qdisc_ops *qops)
148 {
149     struct Qdisc_ops *q, **qp;
150
151     write_lock(&qdisc_mod_lock);
152     for (qp = &qdisc_base; (q=*qp)!=NULL; qp = &q->next) {
153         if (strcmp(qops->id, q->id) == 0) {
154             write_unlock(&qdisc_mod_lock);
155             return -EEXIST;
156         }
157     }
158
159     if (qops->enqueue == NULL)
160         qops->enqueue = noop_qdisc_ops.enqueue;
161     if (qops->requeue == NULL)
162         qops->requeue = noop_qdisc_ops.requeue;
163     if (qops->dequeue == NULL)
164         qops->dequeue = noop_qdisc_ops.dequeue;
165
166     qops->next = NULL;
167     *qp = qops;
168     write_unlock(&qdisc_mod_lock);
169     return 0;
170 }
171
172 int unregister_qdisc (struct Qdisc_ops *qops)
173 {
174     struct Qdisc_ops *q, **qp;
175     int err = -ENOENT;
176
177     write_lock(&qdisc_mod_lock);
178     for (qp = &qdisc_base; (q=*qp)!=NULL; qp = &q->next)
179         if (q == qops)
180             break;
181     if (q) {
182         *qp = q->next;
183         q->next = NULL;
184         err = 0;
185     }
186     write_unlock(&qdisc_mod_lock);
187     return err;
188 }

```

Figure 898: Scheduler API - register/unregister a discipline:
linux/net/sched/sch_api.c

```

190 /* We know handle. Find qdisc among all qdisc's attached to device
191    (root qdisc, all its children, children of children etc.)
192    */
193
194 struct Qdisc *qdisc_lookup(struct net_device *dev, u32 handle)
195 {
196     struct Qdisc *q;
197
198     for (q = dev->qdisc_list; q; q = q->next) {
199         if (q->handle == handle)
200             return q;
201     }
202     return NULL;
203 }
204
205 struct Qdisc *qdisc_leaf(struct Qdisc *p, u32 classid)
206 {
207     unsigned long cl;
208     struct Qdisc *leaf;
209     struct Qdisc_class_ops *cops = p->ops->cl_ops;
210
211     if (cops == NULL)
212         return NULL;
213     cl = cops->get(p, classid);
214
215     if (cl == 0)
216         return NULL;
217     leaf = cops->leaf(p, cl);
218     cops->put(p, cl);
219     return leaf;
220 }

```

Figure 899: Scheduler API - find a discipline by handle:
linux/net/sched/sch_api.c

```

224 struct Qdisc_ops *qdisc_lookup_ops(struct rtattr *kind)
225 {
226     struct Qdisc_ops *q = NULL;
227
228     if (kind) {
229         read_lock(&qdisc_mod_lock);
230         for (q = qdisc_base; q; q = q->next) {
231             if (rtattr_strcmp(kind, q->id) == 0)
232                 break;
233         }
234         read_unlock(&qdisc_mod_lock);
235     }
236     return q;
237 }

```

Figure 900: Scheduler API - find a discipline by name:
linux/net/sched/sch_api.c

```

241 struct qdisc_rate_table *qdisc_get_rtab(struct tc_ratespec *r, struct rtattr *tab)
242 {
243     struct qdisc_rate_table *rtab;
244
245     for (rtab = qdisc_rtab_list; rtab; rtab = rtab->next) {
246         if (memcmp(&rtab->rate, r, sizeof(struct tc_ratespec)) == 0) {
247             rtab->refcnt++;
248             return rtab;
249         }
250     }
251
252     if (tab == NULL || r->rate == 0 || r->cell_log == 0 || RTA_PAYLOAD(tab)
253         != 1024)
254         return NULL;
255
256     rtab = kmalloc(sizeof(*rtab), GFP_KERNEL);
257     if (rtab) {
258         rtab->rate = *r;
259         rtab->refcnt = 1;
260         memcpy(rtab->data, RTA_DATA(tab), 1024);
261         rtab->next = qdisc_rtab_list;
262         qdisc_rtab_list = rtab;
263     }
264     return rtab;
265 }

```

Figure 901: Scheduler API - find a discipline by rate:
linux/net/sched/sch_api.c

```

266 void qdisc_put_rtab(struct qdisc_rate_table *tab)
267 {
268     struct qdisc_rate_table *rtab, **rtabp;
269
270     if (!tab || --tab->refcnt)
271         return;
272
273     for (rtabp = &qdisc_rtab_list; (rtab=*rtabp) != NULL; rtabp = &rtab->next) {
274         if (rtab == tab) {
275             *rtabp = rtab->next;
276             kfree(rtab);
277             return;
278         }
279     }
280 }

```

Figure 902: Scheduler API - free rate table entry: `linux/net/sched/sch_api.c`


```

285 u32 qdisc_alloc_handle(struct net_device *dev)
286 {
287     int i = 0x10000;
288     static u32 autohandle = TC_H_MAKE(0x80000000U, 0);
289
290     do {
291         autohandle += TC_H_MAKE(0x10000U, 0);
292         if (autohandle == TC_H_MAKE(TC_H_ROOT, 0))
293             autohandle = TC_H_MAKE(0x80000000U, 0);
294     } while (qdisc_lookup(dev, autohandle) && --i > 0);
295
296     return i>0 ? autohandle : 0;
297 }

```

Figure 903: Scheduler API - allocate discipline a unique handle:
[linux/net/sched/sch_api.c](#)

```

299 /* Attach toplevel qdisc to device dev */
300
301 static struct Qdisc *
302 dev_graft_qdisc(struct net_device *dev, struct Qdisc *qdisc)
303 {
304     struct Qdisc *oqdisc;
305
306     if (dev->flags & IFF_UP)
307         dev_deactivate(dev);
308
309     write_lock(&qdisc_tree_lock);
310     spin_lock_bh(&dev->queue_lock);
311     if (qdisc && qdisc->flags & TCQ_F_INGRES) {
312         oqdisc = dev->qdisc_ingress;
313         /* Prune old scheduler */
314         if (oqdisc && atomic_read(&oqdisc->refcnt) <= 1) {
315             /* delete */
316             qdisc_reset(oqdisc);
317             dev->qdisc_ingress = NULL;
318         } else { /* new */
319             dev->qdisc_ingress = qdisc;
320         }
321     } else {
322
323         oqdisc = dev->qdisc_sleeping;
324
325         /* Prune old scheduler */
326         if (oqdisc && atomic_read(&oqdisc->refcnt) <= 1)
327             qdisc_reset(oqdisc);
328
329         /* ... and graft new one */
330         if (qdisc == NULL)
331             qdisc = &noop_qdisc;
332         dev->qdisc_sleeping = qdisc;
333         dev->qdisc = &noop_qdisc;
334     }
335
336     spin_unlock_bh(&dev->queue_lock);
337     write_unlock(&qdisc_tree_lock);
338
339     if (dev->flags & IFF_UP)
340         dev_activate(dev);
341
342     return oqdisc;
343 }
344

```

Figure 904: Scheduler API - graft queue discipline to a device:
linux/net/sched/sch_api.c

```

347 /* Graft qdisc "new" to class "classid" of qdisc "parent" or
348    to device "dev".
349
350    Old qdisc is not destroyed but returned in *old.
351 */
352
353 int qdisc_graft (struct net_device *dev, struct Qdisc *parent, u32 classid,
354                struct Qdisc *new, struct Qdisc **old)
355 {
356     int err = 0;
357     struct Qdisc *q = *old;
358
359
360     if (parent == NULL) {
361         if (q && q->flags&TCQ_F_INGRES) {
362             *old = dev_graft_qdisc(dev, q);
363         } else {
364             *old = dev_graft_qdisc(dev, new);
365         }
366     } else {
367         struct Qdisc_class_ops *cops = parent->ops->cl_ops;
368
369         err = -EINVAL;
370
371         if (cops) {
372             unsigned long cl = cops->get(parent, classid);
373             if (cl) {
374                 err = cops->graft(parent, cl, new, old);
375                 cops->put(parent, cl);
376             }
377         }
378     }
379     return err;
380 }

```

Figure 905: Scheduler API - graft queue discipline child to parent class:
linux/net/sched/sch_api.c

```

384  /*
385   Allocate and initialize new qdisc.
386
387   Parameters are passed via opt.
388  */
389
390  static struct Qdisc *
391  qdisc_create(struct net_device *dev, u32 handle, struct rtattr **tca, int *errp)
392  {
393      int err;
394      struct rtattr *kind = tca[TCA_KIND-1];
395      struct Qdisc *sch = NULL;
396      struct Qdisc_ops *ops;
397      int size;
398
399      ops = qdisc_lookup_ops(kind);
400  #ifdef CONFIG_KMOD
401      if (ops==NULL && tca[TCA_KIND-1] != NULL) {
402          char module_name[4 + IFNAMSIZ + 1];
403
404          if (RTA_PAYLOAD(kind) <= IFNAMSIZ) {
405              sprintf(module_name, "sch_%s", (char*)RTA_DATA(kind));
406              request_module(module_name);
407              ops = qdisc_lookup_ops(kind);
408          }
409      }
410  #endif

```

Figure 906: Scheduler API - allocate and initialise new queue discipline:
linux/net/sched/sch_api.c

```
412     err = -EINVAL;
413     if (ops == NULL)
414         goto err_out;
415
416     size = sizeof(*sch) + ops->priv_size;
417
418     sch = kmalloc(size, GFP_KERNEL);
419     err = -ENOMEM;
420     if (!sch)
421         goto err_out;
422
423     /* Grrr... Resolve race condition with module unload */
424
425     err = -EINVAL;
426     if (ops != qdisc_lookup_ops(kind))
427         goto err_out;
428
429     memset(sch, 0, size);
```

Figure 907: Scheduler API - allocate and initialise new queue discipline:
`linux/net/sched/sch_api.c`

```

431     skb_queue_head_init(&sch->q);
432
433     if (handle == TC_H_INGRESS)
434         sch->flags |= TCQ_F_INGRES;
435
436     sch->ops = ops;
437     sch->enqueue = ops->enqueue;
438     sch->dequeue = ops->dequeue;
439     sch->dev = dev;
440     atomic_set(&sch->refcnt, 1);
441     sch->stats.lock = &dev->queue.lock;
442     if (handle == 0) {
443         handle = qdisc_alloc_handle(dev);
444         err = -ENOMEM;
445         if (handle == 0)
446             goto err_out;
447     }
448
449     if (handle == TC_H_INGRESS)
450         sch->handle = TC_H_MAKE(TC_H_INGRESS, 0);
451     else
452         sch->handle = handle;
453
454     if (!ops->init || (err = ops->init(sch, tca[TCA_OPTIONS-1])) == 0) {
455         write_lock(&qdisc_tree_lock);
456         sch->next = dev->qdisc_list;
457         dev->qdisc_list = sch;
458         write_unlock(&qdisc_tree_lock);
459 #ifdef CONFIG_NET_ESTIMATOR
460         if (tca[TCA_RATE-1])
461             qdisc_new_estimator(&sch->stats, tca[TCA_RATE-1]);
462 #endif
463         return sch;
464     }
465
466 err_out:
467     *errp = err;
468     if (sch)
469         kfree(sch);
470     return NULL;
471 }

```

Figure 908: Scheduler API - allocate and initialise new queue discipline:
linux/net/sched/sch_api.c

```

473 static int qdisc_change(struct Qdisc *sch, struct rtattr **tca)
474 {
475     if (tca[TCA_OPTIONS-1]) {
476         int err;
477
478         if (sch->ops->change == NULL)
479             return -EINVAL;
480         err = sch->ops->change(sch, tca[TCA_OPTIONS-1]);
481         if (err)
482             return err;
483     }
484 #ifdef CONFIG_NET_ESTIMATOR
485     if (tca[TCA_RATE-1]) {
486         qdisc_kill_estimator (&sch->stats);
487         qdisc_new_estimator(&sch->stats, tca[TCA_RATE-1]);
488     }
489 #endif
490     return 0;
491 }

```

Figure 909: Scheduler API - modify new queue discipline:
linux/net/sched/sch_api.c

```

493 struct check_loop_arg
494 {
495     struct qdisc_walker    w;
496     struct Qdisc          *p;
497     int                    depth;
498 };
499
500 static int check_loop_fn(struct Qdisc *q, unsigned long cl, struct qdisc_walker *w);
501
502 static int check_loop(struct Qdisc *q, struct Qdisc *p, int depth)
503 {
504     struct check_loop_arg  arg;
505
506     if (q->ops->cl_ops == NULL)
507         return 0;
508
509     arg.w.stop = arg.w.skip = arg.w.count = 0;
510     arg.w.fn = check_loop_fn;
511     arg.depth = depth;
512     arg.p = p;
513     q->ops->cl_ops->walk(q, &arg.w);
514     return arg.w.stop ? -ELOOP : 0;
515 }
516
517 static int
518 check_loop_fn(struct Qdisc *q, unsigned long cl, struct qdisc_walker *w)
519 {
520     struct Qdisc *leaf;
521     struct Qdisc_class_ops *cops = q->ops->cl_ops;
522     struct check_loop_arg *arg = (struct check_loop_arg *)w;
523
524     leaf = cops->leaf(q, cl);
525     if (leaf) {
526         if (leaf == arg->p || arg->depth > 7)
527             return -ELOOP;
528         return check_loop(leaf, arg->p, arg->depth + 1);
529     }
530     return 0;
531 }

```

Figure 910: Scheduler API - detect loops in queue operations graph:
linux/net/sched/sch_api.c


```

537 static int tc_get_qdisc(struct sk_buff *skb, struct nlmsg_hdr *n, void *arg)
538 {
539     struct tcm_msg *tcm = NLMSG_DATA(n);
540     struct rtattr **tca = arg;
541     struct net_device *dev;
542     u32 clid = tcm->tcm_parent;
543     struct Qdisc *q = NULL;
544     struct Qdisc *p = NULL;
545     int err;
546
547     if ((dev = _dev_get_by_index(tcm->tcm_ifindex)) == NULL)
548         return -ENODEV;
549
550     if (clid) {
551         if (clid != TC_H_ROOT) {
552             if (TC_H_MAJ(clid) != TC_H_MAJ(TC_H_INGRESS)) {
553                 if ((p = qdisc_lookup(dev, TC_H_MAJ(clid))) == NULL)
554                     return -ENOENT;
555                 q = qdisc_leaf(p, clid);
556             } else { /* ingress */
557                 q = dev->qdisc_ingress;
558             }
559         } else {
560             q = dev->qdisc_sleeping;
561         }
562         if (!q)
563             return -ENOENT;
564
565         if (tcm->tcm_handle && q->handle != tcm->tcm_handle)
566             return -EINVAL;
567     } else {
568         if ((q = qdisc_lookup(dev, tcm->tcm_handle)) == NULL)
569             return -ENOENT;
570     }

```

Figure 911: Scheduler API - destroy Queue discipline entry:
linux/net/sched/sch_api.c

```

572     if (tca[TCA_KIND-1] && rtattr_strcmp(tca[TCA_KIND-1], q->ops->id))
573         return -EINVAL;
574
575     if (n->nmsg_type == RTM_DELQDISC) {
576         if (!clid)
577             return -EINVAL;
578         if (q->handle == 0)
579             return -ENOENT;
580         if ((err = qdisc_graft(dev, p, clid, NULL, &q)) != 0)
581             return err;
582         if (q) {
583             qdisc_notify(skb, n, clid, q, NULL);
584             spin_lock_bh(&dev->queue_lock);
585             qdisc_destroy(q);
586             spin_unlock_bh(&dev->queue_lock);
587         }
588     } else {
589         qdisc_notify(skb, n, clid, NULL, q);
590     }
591     return 0;
592 }

```

Figure 912: Scheduler API - destroy Queue discipline entry:
linux/net/sched/sch_api.c

```

598 static int tc_modify_qdisc(struct sk_buff *skb, struct nlmsg_hdr *n, void *arg)
599 {
600     struct tcmsg *tcm = NLMSG_DATA(n);
601     struct rtattr **tca = arg;
602     struct net_device *dev;
603     u32 clid = tcm->tcm_parent;
604     struct Qdisc *q = NULL;
605     struct Qdisc *p = NULL;
606     int err;
607
608     if ((dev = _dev_get_by_index(tcm->tcm_ifindex)) == NULL)
609         return -ENODEV;
610
611     if (clid) {
612         if (clid != TC_H_ROOT) {
613             if (clid != TC_H_INGRESS) {
614                 if ((p = qdisc_lookup(dev, TC_H_MAJ(clid))) == NULL)
615                     return -ENOENT;
616                 q = qdisc_leaf(p, clid);
617             } else { /*ingress */
618                 q = dev->qdisc_ingress;
619             }
620         } else {
621             q = dev->qdisc_sleeping;
622         }
623
624         /* It may be default qdisc, ignore it */
625         if (q && q->handle == 0)
626             q = NULL;
627
628         if (!q || !tcm->tcm_handle || q->handle != tcm->tcm_handle) {
629             if (tcm->tcm_handle) {
630                 if (q && !(n->nlmsg_flags & NLM_F_REPLACE))
631                     return -EEXIST;
632                 if (TC_H_MIN(tcm->tcm_handle))
633                     return -EINVAL;
634                 if ((q = qdisc_lookup(dev, tcm->tcm_handle)) == NULL)
635                     goto create_n_graft;
636                 if (n->nlmsg_flags & NLM_F_EXCL)
637                     return -EEXIST;
638                 if (tca[TC_A_KIND-1] && rtattr_strcmp(tca[TC_A_KIND-1], q->ops->id))
639                     return -EINVAL;
640                 if (q == p ||
641                     (p && check_loop(q, p, 0)))
642                     return -ELOOP;
643                 atomic_inc(&q->refcnt);
644                 goto graft;
645             } else {

```

Figure 913: Scheduler API - change Queue discipline entry:
linux/net/sched/sch_api.c

```

646         if (q == NULL)
647             goto create_n_graft;
648
649         /* This magic test requires explanation.
650         *
651         * We know, that some child q is already
652         * attached to this parent and have choice:
653         * either to change it or to create/graft new one.
654         *
655         * 1. We are allowed to create/graft only
656         * if CREATE and REPLACE flags are set.
657         *
658         * 2. If EXCL is set, requestor wanted to say,
659         * that qdisc tcm_handle is not expected
660         * to exist, so that we choose create/graft too.
661         *
662         * 3. The last case is when no flags are set.
663         * Alas, it is sort of hole in API, we
664         * cannot decide what to do unambiguously.
665         * For now we select create/graft, if
666         * user gave KIND, which does not match existing.
667         */
668         if ((n->nmsg_flags&NLM_F_CREATE) &&
669             (n->nmsg_flags&NLM_F_REPLACE) &&
670             ((n->nmsg_flags&NLM_F_EXCL) ||
671              (tca[TCA_KIND-1] &&
672               rtattr_strcmp(tca[TCA_KIND-1], q->ops->id))))
673             goto create_n_graft;
674     }
675 } else {
676     if (!tcm->tcm_handle)
677         return -EINVAL;
678     q = qdisc_lookup(dev, tcm->tcm_handle);
679 }
680

```

Figure 914: Scheduler API - change Queue discipline entry:
linux/net/sched/sch_api.c

```

682     /* Change qdisc parameters */
683     if (q == NULL)
684         return -ENOENT;
685     if (n->nmsg_flags & NLM_F_EXCL)
686         return -EEXIST;
687     if (tca[TCA_KIND-1] && rtattr_strcmp(tca[TCA_KIND-1], q->ops->id))
688         return -EINVAL;
689     err = qdisc_change(q, tca);
690     if (err == 0)
691         qdisc_notify(skb, n, clid, NULL, q);
692     return err;
693
694 create_n_graft :
695     if (!(n->nmsg_flags & NLM_F_CREATE))
696         return -ENOENT;
697     if (clid == TC_H_INGRESS)
698         q = qdisc_create(dev, tcm->tcm_parent, tca, &err);
699     else
700         q = qdisc_create(dev, tcm->tcm_handle, tca, &err);
701     if (q == NULL)
702         return err;
703
704 graft :
705     if (1) {
706         struct Qdisc *old_q = NULL;
707         err = qdisc_graft(dev, p, clid, q, &old_q);
708         if (err) {
709             if (q) {
710                 spin_lock_bh(&dev->queue_lock);
711                 qdisc_destroy(q);
712                 spin_unlock_bh(&dev->queue_lock);
713             }
714             return err;
715         }
716         qdisc_notify(skb, n, clid, old_q, q);
717         if (old_q) {
718             spin_lock_bh(&dev->queue_lock);
719             qdisc_destroy(old_q);
720             spin_unlock_bh(&dev->queue_lock);
721         }
722     }
723     return 0;
724 }

```

Figure 915: Scheduler API - change Queue discipline entry:
linux/net/sched/sch_api.c

```
726 int qdisc_copy_stats(struct sk_buff *skb, struct tc_stats *st)
727 {
728     spin_lock_bh(st->lock);
729     RTA_PUT(skb, TCA_STATS, (char*)&st->lock - (char*)st, st);
730     spin_unlock_bh(st->lock);
731     return 0;
732
733 rtattr_failure :
734     spin_unlock_bh(st->lock);
735     return -1;
736 }
```

Figure 916: Scheduler API - report stats via RT: `linux/net/sched/sch_api.c`

```

738
739 static int tc_fill_qdisc (struct sk_buff *skb, struct Qdisc *q, u32 clid,
740                          u32 pid, u32 seq, unsigned flags, int event)
741 {
742     struct tcmmsg *tcm;
743     struct nlmsg_hdr *nlh;
744     unsigned char *b = skb->tail;
745
746     nlh = NLMSG_PUT(skb, pid, seq, event, sizeof(*tcm));
747     nlh->nlmsg_flags = flags;
748     tcm = NLMSG_DATA(nlh);
749     tcm->tcm_family = AF_UNSPEC;
750     tcm->tcm_ifindex = q->dev ? q->dev->ifindex : 0;
751     tcm->tcm_parent = clid;
752     tcm->tcm_handle = q->handle;
753     tcm->tcm_info = atomic_read(&q->refcnt);
754     RTA_PUT(skb, TCA_KIND, IFNAMSIZ, q->ops->id);
755     if (q->ops->dump && q->ops->dump(q, skb) < 0)
756         goto rtattr_failure ;
757     q->stats.qlen = q->q.qlen;
758     if (qdisc_copy_stats(skb, &q->stats))
759         goto rtattr_failure ;
760     nlh->nlmsg_len = skb->tail - b;
761     return skb->len;
762
763 nlmsg_failure:
764 rtattr_failure :
765     skb_trim(skb, b - skb->data);
766     return -1;
767 }

```

Figure 917: Scheduler API - NLM Message: `linux/net/sched/sch_api.c`

```

769 static int qdisc_notify(struct sk_buff *oskb, struct nlmsg_hdr *n,
770                        u32 clid, struct Qdisc *old, struct Qdisc *new)
771 {
772     struct sk_buff *skb;
773     u32 pid = oskb ? NETLINK_CB(oskb).pid : 0;
774
775     skb = alloc_skb(NLMSG_GOODSIZE, GFP_KERNEL);
776     if (!skb)
777         return -ENOBUFS;
778
779     if (old && old->handle) {
780         if (tc_fill_qdisc(skb, old, clid, pid, n->nlmsg_seq, 0,
781                        RTM_DELQDISC) < 0)
782             goto err_out;
783     }
784     if (new) {
785         if (tc_fill_qdisc(skb, new, clid, pid, n->nlmsg_seq, old ?
786                        NLM_F_REPLACE : 0, RTM_NEWQDISC) < 0)
787             goto err_out;
788     }
789     if (skb->len)
790         return rtnetlink_send(skb, pid, RTMGRP_TC, n->nlmsg_flags &
791                             NLM_F_ECHO);
792
793 err_out:
794     kfree_skb(skb);
795     return -EINVAL;
796 }

```

Figure 918: Scheduler API - notify RTNETLINK: `linux/net/sched/sch_api.c`


```

796 static int tc_dump_qdisc(struct sk_buff *skb, struct netlink_callback *cb)
797 {
798     int idx, q_idx;
799     int s_idx, s_q_idx;
800     struct net_device *dev;
801     struct Qdisc *q;
802
803     s_idx = cb->args[0];
804     s_q_idx = q_idx = cb->args[1];
805     read_lock(&dev_base_lock);
806     for (dev=dev_base, idx=0; dev; dev = dev->next, idx++) {
807         if (idx < s_idx)
808             continue;
809         if (idx > s_idx)
810             s_q_idx = 0;
811         read_lock(&qdisc_tree_lock);
812         for (q = dev->qdisc_list, q_idx = 0; q;
813             q = q->next, q_idx++) {
814             if (q_idx < s_q_idx)
815                 continue;
816             if ( tc_fill_qdisc (skb, q, 0, NETLINK_CB(cb->skb).pid,
817                             cb->nlh->nmsg_seq, NLM_F_MULTI,
818                             RTM_NEWQDISC) <= 0) {
819                 read_unlock(&qdisc_tree_lock);
820                 goto done;
821             }
822             read_unlock(&qdisc_tree_lock);
823         }
824     }
825 done:
826     read_unlock(&dev_base_lock);
827
828     cb->args[0] = idx;
829     cb->args[1] = q_idx;
830
831     return skb->len;
832 }

```

Figure 919: Scheduler API - dump statistics to RT:
linux/net/sched/sch_api.c

```

842 static int tc_ctl_tclass (struct sk_buff *skb, struct nlmsg_hdr *n, void *arg)
843 {
844     struct tcmsg *tcm = NLMSG_DATA(n);
845     struct rtattr **tca = arg;
846     struct net_device *dev;
847     struct Qdisc *q = NULL;
848     struct Qdisc_class_ops *cops;
849     unsigned long cl = 0;
850     unsigned long new_cl;
851     u32 pid = tcm->tcm_parent;
852     u32 clid = tcm->tcm_handle;
853     u32 qid = TC_H_MAJ(clid);
854     int err;
855
856     if ((dev = _dev_get_by_index(tcm->tcm_lifindex)) == NULL)
857         return -ENODEV;
858
859     /*
860     parent == TC_H_UNSPEC - unspecified parent.
861     parent == TC_H_ROOT - class is root, which has no parent.
862     parent == X:0 - parent is root class.
863     parent == X:Y - parent is a node in hierarchy.
864     parent == 0:Y - parent is X:Y, where X:0 is qdisc.
865
866     handle == 0:0 - generate handle from kernel pool.
867     handle == 0:Y - class is X:Y, where X:0 is qdisc.
868     handle == X:Y - clear.
869     handle == X:0 - root class.
870     */

```

Figure 920: Scheduler API - traffic class control: linux/net/sched/sch_api.c

```

872     /* Step 1. Determine qdisc handle X:0 */
873
874     if (pid != TC_H_ROOT) {
875         u32 qid1 = TC_H_MAJ(pid);
876
877         if (qid && qid1) {
878             /* If both majors are known, they must be identical. */
879             if (qid != qid1)
880                 return -EINVAL;
881         } else if (qid1) {
882             qid = qid1;
883         } else if (qid == 0)
884             qid = dev->qdisc_sleeping->handle;
885
886         /* Now qid is genuine qdisc handle consistent
887            both with parent and child.
888
889            TC_H_MAJ(pid) still may be unspecified, complete it now.
890            */
891         if (pid)
892             pid = TC_H_MAKE(qid, pid);
893     } else {
894         if (qid == 0)
895             qid = dev->qdisc_sleeping->handle;
896     }
897
898     /* OK. Locate qdisc */
899     if ((q = qdisc_lookup(dev, qid)) == NULL)
900         return -ENOENT;
901
902     /* An check that it supports classes */
903     cops = q->ops->cl_ops;
904     if (cops == NULL)
905         return -EINVAL;

```

Figure 921: Scheduler API - traffic class control: `linux/net/sched/sch_api.c`

```

907  /* Now try to get class */
908  if (clid == 0) {
909      if (pid == TC_H_ROOT)
910          clid = qid;
911  } else
912      clid = TC_H_MAKE(qid, clid);
913
914  if (clid)
915      cl = cops->get(q, clid);
916
917  if (cl == 0) {
918      err = -ENOENT;
919      if (n->nmsg_type != RTM_NEWTCCLASS || !(n->nmsg_flags&
920          NLM_F_CREATE))
921          goto out;
922  } else {
923      switch (n->nmsg_type) {
924      case RTM_NEWTCCLASS:
925          err = -EEXIST;
926          if (n->nmsg_flags&NLM_F_EXCL)
927              goto out;
928          break;
929      case RTM_DELTCLASS:
930          err = cops->delete(q, cl);
931          if (err == 0)
932              tclass_notify(skb, n, q, cl, RTM_DELTCLASS);
933          goto out;
934      case RTM_GETTCCLASS:
935          err = tclass_notify(skb, n, q, cl, RTM_NEWTCCLASS);
936          goto out;
937      default:
938          err = -EINVAL;
939          goto out;
940      }
941  }

```

Figure 922: Scheduler API - traffic class control: `linux/net/sched/sch_api.c`

```
942     new_cl = cl;
943     err = cops->change(q, clid, pid, tca, &new_cl);
944     if (err == 0)
945         tclass_notify(skb, n, q, new_cl, RTM_NEWTCLASS);
946
947 out:
948     if (cl)
949         cops->put(q, cl);
950
951     return err;
952 }
```

Figure 923: Scheduler API - traffic class control: `linux/net/sched/sch_api.c`

```

1016 static int tc_dump_tclass(struct sk_buff *skb, struct netlink_callback *cb)
1017 {
1018     int t;
1019     int s_t;
1020     struct net_device *dev;
1021     struct Qdisc *q;
1022     struct tcmsg *tcm = (struct tcmsg*)NLMSG_DATA(cb->nlh);
1023     struct qdisc_dump_args arg;
1024
1025     if (cb->nlh->nmlmsg_len < NLMSG_LENGTH(sizeof(*tcm)))
1026         return 0;
1027     if ((dev = dev_get_by_index(tcm->tcm_ifindex)) == NULL)
1028         return 0;
1029
1030     s_t = cb->args[0];
1031
1032     read_lock(&qdisc_tree_lock);
1033     for (q=dev->qdisc_list, t=0; q; q = q->next, t++) {
1034         if (t < s_t) continue;
1035         if (!q->ops->cl_ops) continue;
1036         if (tcm->tcm_parent && TC_H_MAJ(tcm->tcm_parent) != q->handle)
1037             continue;
1038         if (t > s_t)
1039             memset(&cb->args[1], 0, sizeof(cb->args)-sizeof(cb->args[0]));
1040         arg.w.fn = qdisc_class_dump;
1041         arg.skbf = skb;
1042         arg.cb = cb;
1043         arg.w.stop = 0;
1044         arg.w.skip = cb->args[1];
1045         arg.w.count = 0;
1046         q->ops->cl_ops->walk(q, &arg.w);
1047         cb->args[1] = arg.w.count;
1048         if (arg.w.stop)
1049             break;
1050     }
1051     read_unlock(&qdisc_tree_lock);
1052
1053     cb->args[0] = t;
1054
1055     dev_put(dev);
1056     return skb->len;
1057 }

```

Figure 924: Scheduler API - traffic class report to RTNETLINK:
linux/net/sched/sch_api.c

```

1060 int psched_us_per_tick = 1;
1061 int psched_tick_per_us = 1;
1062
1063 #ifdef CONFIG_PROC_FS
1064 static int psched_read_proc(char *buffer, char **start, off_t offset,
1065                             int length, int *eof, void *data)
1066 {
1067     int len;
1068
1069     len = sprintf(buffer, "%08x,%08x,%08x,%08x\n",
1070                  psched_tick_per_us, psched_us_per_tick,
1071                  1000000, HZ);
1072
1073     len -= offset;
1074
1075     if (len > length)
1076         len = length;
1077     if (len < 0)
1078         len = 0;
1079
1080     *start = buffer + offset;
1081     *eof = 1;
1082
1083     return len;
1084 }
1085 #endif
1086
1087 #if PSCHED_CLOCK_SOURCE == PSCHED_GETTIMEOFDAY
1088 int psched_tod_diff(int delta_sec, int bound)
1089 {
1090     int delta;
1091
1092     if (bound <= 1000000 || delta_sec > (0x7FFFFFFF/1000000)-1)
1093         return bound;
1094     delta = delta_sec * 1000000;
1095     if (delta > bound)
1096         delta = bound;
1097     return delta;
1098 }
1099 #endif
1100
1101 psched_time_t psched_time_base;
1102
1103 #if PSCHED_CLOCK_SOURCE == PSCHED_CPU
1104 psched_tdiff_t psched_clock_per_hz;
1105 int psched_clock_scale;
1106 #endif
1107
1108 #ifdef PSCHED_WATCHER
1109 PSCHED_WATCHER psched_time_mark;
1110
1111 static void psched_tick(unsigned long);
1112
1113 static struct timer_list psched_timer =
1114     { function: psched_tick };

```

Figure 925: Scheduler API - Packet Scheduling: `linux/net/sched/sch_api.c`

```

1116 static void psched_tick(unsigned long dummy)
1117 {
1118 #if PSCHED_CLOCK_SOURCE == PSCHED_CPU
1119     psched_time_t dummy_stamp;
1120     PSCHED_GET_TIME(dummy_stamp);
1121     /* It is OK up to 4GHz cpu */
1122     psched_timer.expires = jiffies + 1*HZ;
1123 #else
1124     unsigned long now = jiffies;
1125     psched_time_base = ((u64)now)<<PSCHED_JSCALE;
1126     psched_time_mark = now;
1127     psched_timer.expires = now + 60*60*HZ;
1128 #endif
1129     add_timer(&psched_timer);
1130 }

```

Figure 926: Scheduler API - Packet Scheduling Tick:
linux/net/sched/sch_api.c


```

1133 #if PSCHED_CLOCK_SOURCE == PSCHED_CPU
1134 int _init psched_calibrate_clock(void)
1135 {
1136     psched_time_t stamp, stamp1;
1137     struct timeval tv, tv1;
1138     psched_tdiff_t delay;
1139     long rdelay;
1140     unsigned long stop;
1141
1142 #ifdef PSCHED_WATCHER
1143     psched_tick(0);
1144 #endif
1145     stop = jiffies + HZ/10;
1146     PSCHED_GET_TIME(stamp);
1147     do_gettimeofday(&tv);
1148     while (time_before(jiffies, stop))
1149         barrier();
1150     PSCHED_GET_TIME(stamp1);
1151     do_gettimeofday(&tv1);
1152
1153     delay = PSCHED_TDIFFF(stamp1, stamp);
1154     rdelay = tv1.tv_usec - tv.tv_usec;
1155     rdelay += (tv1.tv_sec - tv.tv_sec)*1000000;
1156     if (rdelay > delay)
1157         return -1;
1158     delay /= rdelay;
1159     psched_tick_per_us = delay;
1160     while ((delay >>= 1) != 0)
1161         psched_clock_scale++;
1162     psched_us_per_tick = 1 << psched_clock_scale;
1163     psched_clock_per_hz = (psched_tick_per_us*(1000000/HZ)) >> psched_clock_scale;
1164     return 0;
1165 }
1166 #endif

```

Figure 927: Scheduler API - Packet Scheduling Tick:
linux/net/sched/sch_api.c

```

1168 int _init pktsched_init(void)
1169 {
1170 #ifdef CONFIG_RTNETLINK
1171     struct rtnetlink_link *link_p;
1172 #endif
1173
1174 #if PSCHED_CLOCK_SOURCE == PSCHED_CPU
1175     if ( psched_calibrate_clock () < 0)
1176         return -1;
1177 #elif PSCHED_CLOCK_SOURCE == PSCHED_JIFFIES
1178     psched_tick_per_us = HZ<<PSCHED_JSCALE;
1179     psched_us_per_tick = 1000000;
1180 #ifdef PSCHED_WATCHER
1181     psched_tick(0);
1182 #endif
1183 #endif
1184
1185 #ifdef CONFIG_RTNETLINK
1186     link_p = rtnetlink_links [PF_UNSPEC];
1187
1188     /* Setup rtnetlink links. It is made here to avoid
1189        exporting large number of public symbols.
1190        */
1191
1192     if (link_p) {
1193         link_p [RTM_NEWQDISC - RTM_BASE].doit = tc_modify_qdisc;
1194         link_p [RTM_DELQDISC - RTM_BASE].doit = tc_get_qdisc;
1195         link_p [RTM_GETQDISC - RTM_BASE].doit = tc_get_qdisc;
1196         link_p [RTM_GETQDISC - RTM_BASE].dumpit = tc_dump_qdisc;
1197         link_p [RTM_NEWTCLASS - RTM_BASE].doit = tc_ctl_tclass;
1198         link_p [RTM_DELTCLASS - RTM_BASE].doit = tc_ctl_tclass;
1199         link_p [RTM_GETTCLASS - RTM_BASE].doit = tc_ctl_tclass;
1200         link_p [RTM_GETTCLASS - RTM_BASE].dumpit = tc_dump_tclass;
1201     }

```

Figure 928: Scheduler API - Packet Scheduling Initialisation:
linux/net/sched/sch_api.c

```
1203
1204 #define INIT_QDISC(name) { \
1205     extern struct Qdisc_ops name##_qdisc_ops; \
1206     register_qdisc (& name##_qdisc_ops); \
1207 }
1208
1209     INIT_QDISC(pfifo);
1210     INIT_QDISC(bfifo);
```

Figure 929: Scheduler API - Packet Scheduling Initialisation:
linux/net/sched/sch_api.c

```

1212 #ifdef CONFIG_NET_SCH_CBQ
1213     INIT_QDISC(cbq);
1214 #endif
1215 #ifdef CONFIG_NET_SCH_CSZ
1216     INIT_QDISC(csz);
1217 #endif
1218 #ifdef CONFIG_NET_SCH_HPFQ
1219     INIT_QDISC(hpfq);
1220 #endif
1221 #ifdef CONFIG_NET_SCH_HFSC
1222     INIT_QDISC(hfsc);
1223 #endif
1224 #ifdef CONFIG_NET_SCH_RED
1225     INIT_QDISC(red);
1226 #endif
1227 #ifdef CONFIG_NET_SCH_GRED
1228     INIT_QDISC(gred);
1229 #endif
1230 #ifdef CONFIG_NET_SCH_INGRESS
1231     INIT_QDISC(ingress);
1232 #endif
1233 #ifdef CONFIG_NET_SCH_DSMARK
1234     INIT_QDISC(dsmark);
1235 #endif
1236 #ifdef CONFIG_NET_SCH_SFQ
1237     INIT_QDISC(sfq);
1238 #endif
1239 #ifdef CONFIG_NET_SCH_TBF
1240     INIT_QDISC(tbf);
1241 #endif
1242 #ifdef CONFIG_NET_SCH_TEQL
1243     teql_init ();
1244 #endif
1245 #ifdef CONFIG_NET_SCH_PRIO
1246     INIT_QDISC(prio);
1247 #endif
1248 #ifdef CONFIG_NET_SCH_ATM
1249     INIT_QDISC(atm);
1250 #endif
1251 #ifdef CONFIG_NET_CLS
1252     tc_filter_init ();
1253 #endif
1254
1255 #ifdef CONFIG_PROC_FS
1256     create_proc_read_entry("net/psched", 0, 0, psched_read_proc, NULL);
1257 #endif

```

Figure 930: Scheduler API - Packet Scheduling Initialisation:
linux/net/sched/sch_api.c

```
20
21 struct qdisc_walker
22 {
23     int    stop;
24     int    skip;
25     int    count;
26     int    (*fn)(struct Qdisc *, unsigned long cl, struct qdisc_walker *);
27 };
```

Figure 931: Packet Scheduler Interface - QDisc Walker Definition:
linux/include/net/pkt_sched.h

```

29 struct Qdisc_class_ops
30 {
31     /* Child qdisc manipulation */
32     int (*graft)(struct Qdisc *, unsigned long cl, struct Qdisc
33         *, struct Qdisc **);
34     struct Qdisc * (*leaf)(struct Qdisc *, unsigned long cl);
35
36     /* Class manipulation routines */
37     unsigned long (*get)(struct Qdisc *, u32 classid);
38     void (*put)(struct Qdisc *, unsigned long);
39     int (*change)(struct Qdisc *, u32, u32, struct rtattr **,
40         unsigned long *);
41     int (*delete)(struct Qdisc *, unsigned long);
42     void (*walk)(struct Qdisc *, struct qdisc_walker * arg);
43
44     /* Filter manipulation */
45     struct tcf_proto ** (*tcf_chain)(struct Qdisc *, unsigned long);
46     unsigned long (*bind_tcf)(struct Qdisc *, unsigned long, u32 classid);
47     void (*unbind_tcf)(struct Qdisc *, unsigned long);
48
49     /* rtnetlink specific */
50     int (*dump)(struct Qdisc *, unsigned long, struct sk_buff *
51         skb, struct tcmsg*);
52 };

```

Figure 932: Packet Scheduler Interface QDisc Class Operations:
[linux/include/net/pkt_sched.h](#)

```

51 struct Qdisc_ops
52 {
53     struct Qdisc_ops    *next;
54     struct Qdisc_class_ops *cl_ops;
55     char                id[IFNAMSIZ];
56     int                 priv_size;
57
58     int                 (*enqueue)(struct sk_buff *, struct Qdisc *);
59     struct sk_buff *    (*dequeue)(struct Qdisc *);
60     int                 (*requeue)(struct sk_buff *, struct Qdisc *);
61     int                 (*drop)(struct Qdisc *);
62
63     int                 (*init)(struct Qdisc *, struct rtattr *arg);
64     void                (*reset)(struct Qdisc *);
65     void                (*destroy)(struct Qdisc *);
66     int                 (*change)(struct Qdisc *, struct rtattr *arg);
67
68     int                 (*dump)(struct Qdisc *, struct sk_buff *);
69 };

```

Figure 933: Packet Scheduler Interface QDisc Operations:
linux/include/net/pkt_sched.h

```

73 struct Qdisc
74 {
75     int                (*enqueue)(struct sk_buff *skb, struct Qdisc *dev);
76     struct sk_buff *   (*dequeue)(struct Qdisc *dev);
77     unsigned           flags;
78     #define TCQ_F_BUILTIN 1
79     #define TCQ_F_THROTTLED 2
80     #define TCQ_F_INGRES 4
81     struct Qdisc_ops   *ops;
82     struct Qdisc       *next;
83     u32                handle;
84     atomic_t           refcnt;
85     struct sk_buff_head q;
86     struct net_device  *dev;
87
88     struct tc_stats    stats;
89     int                (*reshape_fail)(struct sk_buff *skb, struct Qdisc *q);
90
91     /* This field is deprecated, but it is still used by CBQ
92      * and it will live until better solution will be invented.
93      */
94     struct Qdisc       *_parent;
95
96     char               data[0];
97 };

```

Figure 934: Packet Scheduler Interface QDisc Structure:
linux/include/net/pkt_sched.h


```

99 struct qdisc_rate_table
100 {
101     struct tc_ratespec rate;
102     u32      data[256];
103     struct qdisc_rate_table *next;
104     int      refcnt;
105 };
106
107 static inline void sch_tree_lock(struct Qdisc *q)
108 {
109     write_lock(&qdisc_tree_lock);
110     spin_lock_bh(&q->dev->queue_lock);
111 }
112
113 static inline void sch_tree_unlock(struct Qdisc *q)
114 {
115     spin_unlock_bh(&q->dev->queue_lock);
116     write_unlock(&qdisc_tree_lock);
117 }
118
119 static inline void tcf_tree_lock(struct tcf_proto *tp)
120 {
121     write_lock(&qdisc_tree_lock);
122     spin_lock_bh(&tp->q->dev->queue_lock);
123 }
124
125 static inline void tcf_tree_unlock(struct tcf_proto *tp)
126 {
127     spin_unlock_bh(&tp->q->dev->queue_lock);
128     write_unlock(&qdisc_tree_lock);
129 }
130
131
132 static inline unsigned long
133 cls_set_class(struct tcf_proto *tp, unsigned long *clp, unsigned long cl)
134 {
135     unsigned long old_cl;
136
137     tcf_tree_lock(tp);
138     old_cl = *clp;
139     *clp = cl;
140     tcf_tree_unlock(tp);
141     return old_cl;
142 }
143
144 static inline unsigned long
145 _cls_set_class(unsigned long *clp, unsigned long cl)
146 {
147     unsigned long old_cl;
148
149     old_cl = *clp;
150     *clp = cl;
151     return old_cl;
152 }

```

Figure 935: Packet Scheduler Interface QDisc Operations:
linux/include/net/pkt_sched.h

```

155  /*
156     Timer resolution MUST BE < 10% of min_schedulable_packet_size/bandwidth
157
158     Normal IP packet size ~ 512byte, hence:
159
160     0.5Kbyte/1Mbyte/sec = 0.5msec, so that we need 50usec timer for
161     10Mbit ethernet.
162
163     10msec resolution -> <50Kbit/sec.
164
165     The result: [34]86 is not good choice for QoS router :-(
166
167     The things are not so bad, because we may use artificial
168     clock evaluated by integration of network data flow
169     in the most critical places.
170
171     Note: we do not use fastgettimeofday.
172     The reason is that, when it is not the same thing as
173     gettimeofday, it returns invalid timestamp, which is
174     not updated, when net_bh is active.
175
176     So, use PSCHED_CLOCK_SOURCE = PSCHED_CPU on alpha and pentiums
177     with rtdsc. And PSCHED_JIFFIES on all other architectures, including [34]86
178     and pentiums without rtdsc.
179     You can use PSCHED_GETTIMEOFDAY on another architectures,
180     which have fast and precise clock source, but it is too expensive.
181  */
182
183  /* General note about internal clock.
184
185     Any clock source returns time intervals, measured in units
186     close to 1usec. With source PSCHED_GETTIMEOFDAY it is precisely
187     microseconds, otherwise something close but different chosen to minimize
188     arithmetic cost. Ratio usec/internal untis in form nominator/denominator
189     may be read from /proc/net/psched.
190  */

```

Figure 936: Packet Scheduler Interface Clock Comment:
linux/include/net/pkt_sched.h

```
44 #ifdef CONFIG_RTNETLINK
45 static int qdisc_notify(struct sk_buff *oskb, struct nlmsg_hdr *n, u32 clid,
46                       struct Qdisc *old, struct Qdisc *new);
47 static int tclass_notify(struct sk_buff *oskb, struct nlmsg_hdr *n,
48                         struct Qdisc *q, unsigned long cl, int event);
49 #endif
```

Figure 937: Scheduler API ii: `linux/net/sched/sch_api.c`

```
51  /*
52
53     Short review.
54     -----
55
56     This file consists of two interrelated parts:
57
58     1. queueing disciplines manager frontend.
59     2. traffic classes manager frontend.
60
61     Generally, queueing discipline ("qdisc") is a black box,
62     which is able to enqueue packets and to dequeue them (when
63     device is ready to send something) in order and at times
64     determined by algorithm hidden in it.
65
66     qdisc's are divided to two categories:
67     - "queues", which have no internal structure visible from outside.
68     - "schedulers", which split all the packets to "traffic classes",
69       using "packet classifiers" (look at cls_api.c)
70
71     In turn, classes may have child qdiscs (as rule, queues)
72     attached to them etc. etc. etc.
```

Figure 938: Scheduler API discussion: linux/net/sched/sch_api.c

```
74 The goal of the routines in this file is to translate
75 information supplied by user in the form of handles
76 to more intelligible for kernel form, to make some sanity
77 checks and part of work, which is common to all qdiscs
78 and to provide rtnetlink notifications .
79
80 All real intelligent work is done inside qdisc modules.
```

Figure 939: Scheduler API more discussion: `linux/net/sched/sch_api.c`

```

84 Every discipline has two major routines: enqueue and dequeue.
85
86 ---dequeue
87
88 dequeue usually returns a skb to send. It is allowed to return NULL,
89 but it does not mean that queue is empty, it just means that
90 discipline does not want to send anything this time.
91 Queue is really empty if q->q.qlen == 0.
92 For complicated disciplines with multiple queues q->q is not
93 real packet queue, but however q->q.qlen must be valid.
94
95 ---enqueue
96
97 enqueue returns 0, if packet was enqueued successfully.
98 If packet (this one or another one) was dropped, it returns
99 not zero error code.
100 NET_XMIT_DROP - this packet dropped
101     Expected action: do not backoff, but wait until queue will clear.
102 NET_XMIT_CN   - probably this packet enqueued, but another one dropped.
103     Expected action: backoff or ignore
104 NET_XMIT_POLICED - dropped by police.
105     Expected action: backoff or error to real-time apps.

```

Figure 940: Scheduler API more discussion: `linux/net/sched/sch_api.c`

```
107 Auxiliary routines:
108
109 ---requeue
110
111 requeues once dequeued packet. It is used for non-standard or
112 just buggy devices, which can defer output even if dev->tbusy=0.
113
114 ---reset
115
116 returns qdisc to initial state: purge all buffers, clear all
117 timers, counters (except for statistics) etc.
118
119 ---init
120
121 initializes newly created qdisc.
122
123 ---destroy
124
125 destroys resources allocated by init and during lifetime of qdisc.
126
127 ---change
128
129 changes qdisc parameters.
130 */
```

Figure 941: Scheduler API more: `linux/net/sched/sch_api.c`

```

7  *           2 of the License, or (at your option) any later version.
8  *
9  * Authors:   Alexey Kuznetsov, <kuznet@ms2.inr.ac.ru>
10 *           Jamal Hadi Salim, <hadi@nortelnetworks.com> 990601
11 *           - Ingress support
12 */
13
14 #include <asm/uaccess.h>
15 #include <asm/system.h>
16 #include <asm/bitops.h>
17 #include <linux/config.h>
18 #include <linux/types.h>
19 #include <linux/kernel.h>
20 #include <linux/sched.h>
21 #include <linux/string.h>
22 #include <linux/mm.h>
23 #include <linux/socket.h>
24 #include <linux/sockios.h>
25 #include <linux/in.h>
26 #include <linux/errno.h>
27 #include <linux/interrupt.h>
28 #include <linux/netdevice.h>
29 #include <linux/skbuff.h>
30 #include <linux/rtnetlink.h>
31 #include <linux/init.h>
32 #include <net/sock.h>
33 #include <net/pkt_sched.h>
34
35 /* Main transmission queue. */
36
37 /* Main qdisc structure lock.
38
39    However, modifications
40    to data, participating in scheduling must be additionally
41    protected with dev->queue_lock spinlock.
42
43    The idea is the following:
44    - enqueue, dequeue are serialized via top level device
45      spinlock dev->queue_lock.
46    - tree walking is protected by read_lock(qdisc_tree_lock)
47      and this lock is used only in process context.
48    - updates to tree are made only under rtnl semaphore,
49      hence this lock may be made without local bh disabling.
50
51    qdisc_tree_lock must be grabbed BEFORE dev->queue_lock!
52 */
53 rwlock_t qdisc_tree_lock = RW_LOCK_UNLOCKED;
54
55 /*
56    dev->queue_lock serializes queue accesses for this device
57    AND dev->qdisc pointer itself.
58
59    dev->xmit_lock serializes accesses to device driver.
60
61    dev->queue_lock and dev->xmit_lock are mutually exclusive,
62    if one is grabbed, another must be free.
63 */

```

Figure 942: A Generic Scheduler - Queue Disciple Lock Comment:
linux/net/sched/sch_generic.c


```
66  /* Kick device.
67     Note, that this procedure can be called by a watchdog timer, so that
68     we do not check dev->tbusy flag here.
69
70     Returns: 0 - queue is empty.
71             >0 - queue is not empty, but throttled.
72             <0 - queue is not empty. Device is throttled, if dev->tbusy != 0.
73
74     NOTE: Called under dev->queue_lock with locally disabled BH.
75  */
```

Figure 943: A Generic Scheduler i: linux/net/sched/sch_generic.c

```

77 int qdisc_restart (struct net_device *dev)
78 {
79     struct Qdisc *q = dev->qdisc;
80     struct sk_buff *skb;
81
82     /* Dequeue packet */
83     if ((skb = q->dequeue(q)) != NULL) {
84         if (spin_trylock(&dev->xmit_lock)) {
85             /* Remember that the driver is grabbed by us. */
86             dev->xmit_lock_owner = smp_processor_id();
87
88             /* And release queue */
89             spin_unlock(&dev->queue_lock);
90
91             if (!netif_queue_stopped(dev)) {
92                 if (netdev_nit)
93                     dev_queue_xmit_nit(skb, dev);
94
95                 if (dev->hard_start_xmit(skb, dev) == 0) {
96                     dev->xmit_lock_owner = -1;
97                     spin_unlock(&dev->xmit_lock);
98
99                     spin_lock(&dev->queue_lock);
100                    return -1;
101                }
102            }
103
104            /* Release the driver */
105            dev->xmit_lock_owner = -1;
106            spin_unlock(&dev->xmit_lock);
107            spin_lock(&dev->queue_lock);
108            q = dev->qdisc;

```

Figure 944: A Generic Scheduler i: `linux/net/sched/sch_generic.c`

```

109     } else {
110         /* So, someone grabbed the driver. */
111
112         /* It may be transient configuration error,
113            when hard_start_xmit() recurses. We detect
114            it by checking xmit owner and drop the
115            packet when deadlock is detected.
116            */
117         if (dev->xmit_lock_owner == smp_processor_id()) {
118             kfree_skb(skb);
119             if (net_ratelimit())
120                 printk(KERN_DEBUG "Dead loop on net device
121                    _%s, fix it urgently!\n", dev->name);
122             return -1;
123         }
124         netdev_rx_stat[smp_processor_id()].cpu_collision++;
125     }
126
127     /* Device kicked us out :(
128        This is possible in three cases:
129
130        0. driver is locked
131        1. fastroute is enabled
132        2. device cannot determine busy state
133           before start of transmission (f.e. dialout)
134        3. device is buggy (ppp)
135        */
136     q->ops->requeue(skb, q);
137     netif_schedule(dev);
138     return 1;
139 }
140 return q->q.len;
141 }

```

Figure 945: A Generic Scheduler ii: linux/net/sched/sch_generic.c

```

143 static void dev_watchdog(unsigned long arg)
144 {
145     struct net_device *dev = (struct net_device *)arg;
146
147     spin_lock(&dev->xmit_lock);
148     if (dev->qdisc != &noop_qdisc) {
149         if (netif_device_present(dev) &&
150             netif_running(dev) &&
151             netif_carrier_ok(dev)) {
152             if (netif_queue_stopped(dev) &&
153                 (jiffies - dev->trans_start) > dev->watchdog_timeo) {
154                 printk(KERN_INFO "NETDEV WATCHDOG: %s:
155                     transmit timed out\n", dev->name);
156                 dev->tx_timeout(dev);
157             }
158             if (!mod_timer(&dev->watchdog_timer, jiffies + dev->
159                 watchdog_timeo))
160                 dev_hold(dev);
161         }
162     }
163     spin_unlock(&dev->xmit_lock);
164     dev_put(dev);
165 }
166
167 static void dev_watchdog_init(struct net_device *dev)
168 {
169     init_timer(&dev->watchdog_timer);
170     dev->watchdog_timer.data = (unsigned long)dev;
171     dev->watchdog_timer.function = dev_watchdog;
172 }
173
174 void __netdev_watchdog_up(struct net_device *dev)
175 {
176     if (dev->tx_timeout) {
177         if (dev->watchdog_timeo <= 0)
178             dev->watchdog_timeo = 5*HZ;
179         if (!mod_timer(&dev->watchdog_timer, jiffies + dev->watchdog_timeo))
180             dev_hold(dev);
181     }
182 }
183
184 static void dev_watchdog_up(struct net_device *dev)
185 {
186     spin_lock_bh(&dev->xmit_lock);
187     __netdev_watchdog_up(dev);
188     spin_unlock_bh(&dev->xmit_lock);
189 }
190
191 static void dev_watchdog_down(struct net_device *dev)
192 {
193     spin_lock_bh(&dev->xmit_lock);
194     if (del_timer(&dev->watchdog_timer))
195         __dev_put(dev);
196     spin_unlock_bh(&dev->xmit_lock);
197 }

```

Figure 946: A Generic Scheduler - Watchdog Timers: linux/net/sched/sch_generic.c

```

198 /* "NOOP" scheduler: the best scheduler, recommended for all interfaces
199    under all circumstances. It is difficult to invent anything faster or
200    cheaper.
201 */
202
203 static int
204 noop_enqueue(struct sk_buff *skb, struct Qdisc *qdisc)
205 {
206     kfree_skb(skb);
207     return NET_XMIT_CN;
208 }
209
210 static struct sk_buff *
211 noop_dequeue(struct Qdisc *qdisc)
212 {
213     return NULL;
214 }
215
216 static int
217 noop_requeue(struct sk_buff *skb, struct Qdisc *qdisc)
218 {
219     if (net_ratelimit())
220         printk(KERN_DEBUG "%s: deferred output. It is buggy.\n", skb->dev
221                ->name);
222     kfree_skb(skb);
223     return NET_XMIT_CN;
224 }

```

Figure 947: A Generic Scheduler - No-op Scheduler:
linux/net/sched/sch_generic.c

```

225 struct Qdisc_ops noop_qdisc_ops =
226 {
227     NULL,
228     NULL,
229     "noop",
230     0,
231
232     noop_enqueue,
233     noop_dequeue,
234     noop_requeue,
235 };
236
237 struct Qdisc noop_qdisc =
238 {
239     noop_enqueue,
240     noop_dequeue,
241     TCQ_F_BUILTIN,
242     &noop_qdisc_ops,
243 };
244
245 struct Qdisc_ops noqueue_qdisc_ops =
246 {
247     NULL,
248     NULL,
249     "noqueue",
250     0,
251
252     noop_enqueue,
253     noop_dequeue,
254     noop_requeue,
255 };
256
257 };
258
259 struct Qdisc noqueue_qdisc =
260 {
261     NULL,
262     noop_dequeue,
263     TCQ_F_BUILTIN,
264     &noqueue_qdisc_ops,
265 };

```

Figure 948: A Generic Scheduler - Operations:
linux/net/sched/sch_generic.c

```

268 static const u8 prio2band[TC_PRIO_MAX+1] =
269     { 1, 2, 2, 2, 1, 2, 0, 0, 1, 1, 1, 1, 1, 1, 1 };
270
271 /* 3-band FIFO queue: old style, but should be a bit faster than
272    generic prio+fifo combination.
273    */
274
275 static int
276 pfifo_fast_enqueue(struct sk_buff *skb, struct Qdisc* qdisc)
277 {
278     struct sk_buff_head *list ;
279
280     list = ((struct sk_buff_head*)qdisc->data) +
281           prio2band[skb->priority&TC_PRIO_MAX];
282
283     if ( list->qlen <= skb->dev->tx_queue_len) {
284         __skb_queue_tail( list , skb);
285         qdisc->q.qlen++;
286         return 0;
287     }
288     qdisc->stats.drops++;
289     kfree_skb(skb);
290     return NET_XMIT_DROP;
291 }

```

Figure 949: A Generic Scheduler - PFIFO Operations:
linux/net/sched/sch_generic.c

```

293 static struct sk_buff *
294 pfifo_fast_dequeue(struct Qdisc* qdisc)
295 {
296     int prio;
297     struct sk_buff_head * list = ((struct sk_buff_head*)qdisc->data);
298     struct sk_buff *skb;
299
300     for (prio = 0; prio < 3; prio++, list++) {
301         skb = _skb_dequeue(list);
302         if (skb) {
303             qdisc->q.qlen--;
304             return skb;
305         }
306     }
307     return NULL;
308 }
309
310 static int
311 pfifo_fast_requeue(struct sk_buff *skb, struct Qdisc* qdisc)
312 {
313     struct sk_buff_head * list ;
314
315     list = ((struct sk_buff_head*)qdisc->data) +
316           prio2band[skb->priority&TC_PRIO_MAX];
317
318     _skb_queue_head(list , skb);
319     qdisc->q.qlen++;
320     return 0;
321 }
322
323 static void
324 pfifo_fast_reset (struct Qdisc* qdisc)
325 {
326     int prio;
327     struct sk_buff_head * list = ((struct sk_buff_head*)qdisc->data);
328
329     for (prio=0; prio < 3; prio++)
330         skb_queue_purge(list+prio);
331     qdisc->q.qlen = 0;
332 }
333
334 static int pfifo_fast_init (struct Qdisc *qdisc, struct rtattr *opt)
335 {
336     int i;
337     struct sk_buff_head * list ;
338
339     list = ((struct sk_buff_head*)qdisc->data);
340
341     for (i=0; i<3; i++)
342         skb_queue_head_init(list+i);
343
344     return 0;
345 }

```

Figure 950: A Generic Scheduler - PFIFO Operations:
linux/net/sched/sch_generic.c


```
347 static struct Qdisc_ops pfifo_fast_ops =
348 {
349     NULL,
350     NULL,
351     " pfifo_fast ",
352     3 * sizeof(struct sk_buff_head),
353
354     pfifo_fast_enqueue ,
355     pfifo_fast_dequeue ,
356     pfifo_fast_requeue ,
357     NULL,
358
359     pfifo_fast_init ,
360     pfifo_fast_reset ,
361 };
```

Figure 951: A Generic Scheduler - PFIFO declare:
linux/net/sched/sch_generic.c

```

363 struct Qdisc * qdisc_create_dflt (struct net_device *dev, struct Qdisc_ops *ops)
364 {
365     struct Qdisc *sch;
366     int size = sizeof(*sch) + ops->priv_size;
367
368     sch = kmalloc(size, GFP_KERNEL);
369     if (!sch)
370         return NULL;
371     memset(sch, 0, size);
372
373     skb_queue_head_init(&sch->q);
374     sch->ops = ops;
375     sch->enqueue = ops->enqueue;
376     sch->dequeue = ops->dequeue;
377     sch->dev = dev;
378     sch->stats.lock = &dev->queue_lock;
379     atomic_set(&sch->refcnt, 1);
380     if (!ops->init || ops->init(sch, NULL) == 0)
381         return sch;
382
383     kfree(sch);
384     return NULL;
385 }

```

Figure 952: A Generic Scheduler - create default entry:
linux/net/sched/sch_generic.c

```

40  /*      Class-Based Queueing (CBQ) algorithm.
41  =====
42
43  Sources: [1] Sally Floyd and Van Jacobson, "Link-sharing and Resource
44            Management Models for Packet Networks",
45            IEEE/ACM Transactions on Networking, Vol.3, No.4, 1995
46
47            [2] Sally Floyd, "Notes on CBQ and Guaranteed Service", 1995
48
49            [3] Sally Floyd, "Notes on Class-Based Queueing: Setting
50            Parameters", 1996
51
52            [4] Sally Floyd and Michael Speer, "Experimental Results
53            for Class-Based Queueing", 1998, not published.
54
55  -----
56
57  Algorithm skeleton was taken from NS simulator cbq.cc.
58  If someone wants to check this code against the LBL version,
59  he should take into account that ONLY the skeleton was borrowed,
60  the implementation is different. Particularly:
61
62  --- The WRR algorithm is different. Our version looks more
63  reasonable (I hope) and works when quanta are allowed to be
64  less than MTU, which is always the case when real time classes
65  have small rates. Note, that the statement of [3] is
66  incomplete, delay may actually be estimated even if class
67  per-round allotment is less than MTU. Namely, if per-round
68  allotment is  $W*r_i$ , and  $r_1+...+r_k = r < 1$ 
69
70   $delay_i \leq ([MTU/(W*r_i)]*W*r + W*r + k*MTU)/B$ 
71
72  In the worst case we have IntServ estimate with  $D = W*r+k*MTU$ 
73  and  $C = MTU*r$ . The proof (if correct at all) is trivial.

```

Figure 953: CBQ - Initial Comment i: linux/net/sched/sch_cbq.c

```
76      --- It seems that cbq-2.0 is not very accurate. At least, I cannot
77      interpret some places, which look like wrong translations
78      from NS. Anyone is advised to find these differences
79      and explain to me, why I am wrong 8).
80
81      --- Linux has no EOI event, so that we cannot estimate true class
82      idle time. Workaround is to consider the next dequeue event
83      as sign that previous packet is finished . This is wrong because of
84      internal device queueing, but on a permanently loaded link it is true.
85      Moreover, combined with clock integrator, this scheme looks
86      very close to an ideal solution. */
```

Figure 954: CBQ - Initial Comment ii: `linux/net/sched/sch_cbq.c`

```
88 struct cbq_sched_data;
89
90
91 struct cbq_class
92 {
93     struct cbq_class    *next;        /* hash table link */
94     struct cbq_class    *next_alive;  /* next class with backlog in this
95     priority band */
96
97     /* Parameters */
98     u32                  classid;
99     unsigned char        priority;    /* class priority */
```

Figure 955: CBQ - Class Structure Definition i: `linux/net/sched/sch_cbq.c`

```
99     unsigned char    priority2;    /* priority to be used after
      overlimit */
100     unsigned char    ewma_log;    /* time constant for idle time
      calculation */
101     unsigned char    ovl_strategy;
102 #ifdef CONFIG_NET_CLS_POLICE
103     unsigned char    police;
104 #endif
105
106     u32                defmap;
```

Figure 956: CBQ - Class Structure Definition i: `linux/net/sched/sch_cbq.c`

```

108  /* Link-sharing scheduler parameters */
109  long          maxidle;      /* Class paramters: see below. */
110  long          offtime;
111  long          minidle;
112  u32           avpkt;
113  struct qdisc_rate_table *R_tab;
114
115  /* Overlimit strategy parameters */
116  void          (*overlimit)(struct cbq_class *cl);
117  long          penalty;
118
119  /* General scheduler (WRR) parameters */
120  long          allot;
121  long          quantum;     /* Allotment per WRR round */
122  long          weight;     /* Relative allotment: see below */
123
124  struct Qdisc  *qdisc;     /* Ptr to CBQ discipline */
125  struct cbq_class *split;  /* Ptr to split node */
126  struct cbq_class *share;  /* Ptr to LS parent in the class tree
127  */
128  struct cbq_class *tparent; /* Ptr to tree parent in the class
129  tree */
130  struct cbq_class *borrow; /* NULL if class is bandwidth limited
131  ;
132  parent otherwise */
133  struct cbq_class *sibling; /* Sibling chain */
134  struct cbq_class *children; /* Pointer to children chain */
135
136  struct Qdisc  *q;        /* Elementary queueing discipline */

```

Figure 957: CBQ - Class Structure Definition ii: linux/net/sched/sch_cbq.c

```
136 /* Variables */
137     unsigned char    cpriority ;    /* Effective priority */
138     unsigned char    delayed;
139     unsigned char    level;        /* level of the class in hierarchy:
140                                     0 for leaf classes, and maximal
141                                     level of children + 1 for nodes.
142                                     */
143
144     psched_time_t    last;        /* Last end of service */
```

Figure 958: CBQ - Class Structure Definition iii: linux/net/sched/sch_cbq.c


```

145     psched_time_t      undertime;
146     long               avgidle;
147     long               deficit ;           /* Saved deficit for WRR */
148     unsigned long     penalized;
149     struct tc_stats   stats;
150     struct tc_cbq_xstats xstats;
151
152     struct tcf_proto   * filter_list ;
153
154     int                refcnt ;
155     int                filters ;
156
157     struct cbq_class   *defaults[TC_PRIO_MAX+1];
158 };

```

Figure 959: CBQ - Class Structure Definition iii: `linux/net/sched/sch_cbq.c`

```

160 struct cbq_sched_data
161 {
162     struct cbq_class    *classes [16];          /* Hash table of all classes
163         */
164     int                 nclasses [TC_CBQ_MAXPRIO+1];
165     unsigned            quanta [TC_CBQ_MAXPRIO+1];
166     struct cbq_class    link;
167
168     unsigned            activemask;
169     struct cbq_class    *active [TC_CBQ_MAXPRIO+1]; /* List of all classes
170         with backlog */
171
172 #ifdef CONFIG_NET_CLS_POLICE
173     struct cbq_class    *rx_class;
174 #endif
175     struct cbq_class    *tx_class;
176     struct cbq_class    *tx_borrowed;
177     int                 tx_len;
178     psched_time_t       now;                    /* Cached timestamp */
179     psched_time_t       now_rt;                /* Cached real time */
180     unsigned            pmask;
181
182     struct timer_list    delay_timer;
183     struct timer_list    wd_timer;             /* Watchdog timer,
184         started when CBQ has
185         backlog, but cannot
186         transmit just now */
187     long                wd_expires;
188     int                 toplevel;
189     u32                 hgenerator;
190 };

```

Figure 960: CBQ - Scheduling Data Forwarding: linux/net/sched/sch_cbq.c

```

230 /* Classify packet. The procedure is pretty complicated, but
231    it allows us to combine link sharing and priority scheduling
232    transparently.
233
234    Namely, you can put link sharing rules (f.e. route based) at root of CBQ,
235    so that it resolves to split nodes. Then packets are classified
236    by logical priority, or a more specific classifier may be attached
237    to the split node.
238 */
239
240 static struct cbq_class *
241 cbq_classify (struct sk_buff *skb, struct Qdisc *sch)
242 {
243     struct cbq_sched_data *q = (struct cbq_sched_data*)sch->data;
244     struct cbq_class *head = &q->link;
245     struct cbq_class **defmap;
246     struct cbq_class *cl = NULL;
247     u32 prio = skb->priority;
248     struct tcf_result res;
249
250     /*
251      * Step 1. If skb->priority points to one of our classes, use it.
252      */
253     if (TC_H_MAJ(prio^sch->handle) == 0 &&
254         (cl = cbq_class_lookup(q, prio)) != NULL)
255         return cl;
256
257     for (;;) {
258         int result = 0;
259
260         defmap = head->defaults;
261
262         /*
263          * Step 2+n. Apply classifier.
264          */
265         if (!head->filter_list || (result = tc_classify (skb, head->filter_list, &
266             res)) < 0)
267             goto fallback;
268
269         if ((cl = (void*)res.class) == NULL) {
270             if (TC_H_MAJ(res.classid))
271                 cl = cbq_class_lookup(q, res.classid);
272             else if ((cl = defmap[res.classid&TC_PRIO_MAX]) == NULL)
273                 cl = defmap[TC_PRIO_BESTEFFORT];
274
275             if (cl == NULL || cl->level >= head->level)
276                 goto fallback;
277         }
278     }

```

Figure 961: CBQ - Classifier - i: linux/net/sched/sch_cbq.c

```

278 #ifdef CONFIG_NET_CLS_POLICE
279     switch (result) {
280     case TC_POLICE_RECLASSIFY:
281         return cbq_reclassify(skb, cl);
282     case TC_POLICE_SHOT:
283         return NULL;
284     default:
285     }
286 #endif
287     if (cl->level == 0)
288         return cl;
289
290     /*
291     * Step 3+n. If classifier selected a link sharing class,
292     *     apply agency specific classifier.
293     *     Repeat this procedure until we hit a leaf node.
294     */
295     head = cl;
296 }
297
298 fallback:
299     cl = head;
300
301     /*
302     * Step 4. No success...
303     */
304     if (TC_H_MAJ(prio) == 0 &&
305         !(cl = head->defaults[prio&TC_PRIO_MAX]) &&
306         !(cl = head->defaults[TC_PRIO_BESTEFFORT]))
307         return head;
308
309     return cl;
310 }

```

Figure 962: CBQ - Classifier - ii: linux/net/sched/sch_cbq.c

```
216 static struct cbq_class *
217 cbq_reclassify(struct sk_buff *skb, struct cbq_class *this)
218 {
219     struct cbq_class *cl, *new;
220
221     for (cl = this->tparent; cl; cl = cl->tparent)
222         if ((new = cl->defaults[TC_PRIO_BESTEFFORT]) != NULL && new
223             != this)
224             return new;
225     return NULL;
226 }
```

Figure 963: CBQ - Re-Classifer: `linux/net/sched/sch_cbq.c`

```

312  /*
313     A packet has just been enqueued on the empty class.
314     cbq_activate_class adds it to the tail of active class list
315     of its priority band.
316  */
317
318  static __inline__ void cbq_activate_class(struct cbq_class *cl)
319  {
320     struct cbq_sched_data *q = (struct cbq_sched_data*)cl->qdisc->data;
321     int prio = cl->cpriority;
322     struct cbq_class *cl_tail;
323
324     cl_tail = q->active[prio];
325     q->active[prio] = cl;
326
327     if ( cl_tail != NULL) {
328         cl->next_alive = cl_tail->next_alive;
329         cl_tail->next_alive = cl;
330     } else {
331         cl->next_alive = cl;
332         q->activemask |= (1<<prio);
333     }
334 }

```

Figure 964: CBQ - Activate or Deactivate a Class:
linux/net/sched/sch_cbq.c

```

336  /*
337   Unlink class from active chain.
338   Note that this same procedure is done directly in cbq_dequeue*
339   during round-robin procedure.
340  */
341
342  static void cbq_deactivate_class(struct cbq_class *this)
343  {
344      struct cbq_sched_data *q = (struct cbq_sched_data*)this->qdisc->data;
345      int prio = this->cpriority;
346      struct cbq_class *cl;
347      struct cbq_class *cl_prev = q->active[prio];
348
349      do {
350          cl = cl_prev->next_alive;
351          if (cl == this) {
352              cl_prev->next_alive = cl->next_alive;
353              cl->next_alive = NULL;
354
355              if (cl == q->active[prio]) {
356                  q->active[prio] = cl_prev;
357                  if (cl == q->active[prio]) {
358                      q->active[prio] = NULL;
359                      q->activemask &= ~(1<<prio);
360                      return;
361                  }
362              }
363
364              cl = cl_prev->next_alive;
365              return;
366          }
367      } while ((cl_prev = cl) != q->active[prio]);
368  }

```

Figure 965: CBQ - Activate or Deactivate a Class:
linux/net/sched/sch_cbq.c

```

370 static void
371 cbq_mark_toplevel(struct cbq_sched_data *q, struct cbq_class *cl)
372 {
373     int toplevel = q->toplevel;
374
375     if (toplevel > cl->level && !(cl->q->flags & TCQ_F_THROTTLED)) {
376         psched_time_t now;
377         psched_tdiff_t incr;
378
379         PSCHED_GET_TIME(now);
380         incr = PSCHED_TDIF2(now, q->now_rt);
381         PSCHED_TADD2(q->now, incr, now);
382
383         do {
384             if (PSCHED_TLESS(cl->undertime, now)) {
385                 q->toplevel = cl->level;
386                 return;
387             }
388         } while ((cl = cl->borrow) != NULL && toplevel > cl->level);
389     }
390 }

```

Figure 966: CBQ - Mark Top Level: `linux/net/sched/sch_cbq.c`


```

392 static int
393 cbq_enqueue(struct sk_buff *skb, struct Qdisc *sch)
394 {
395     struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
396     struct cbq_class *cl = cbq_classify(skb, sch);
397     int len = skb->len;
398     int ret = NET_XMIT_POLICED;
399
400 #ifdef CONFIG_NET_CLS_POLICE
401     q->rx_class = cl;
402 #endif
403     if (cl) {
404 #ifdef CONFIG_NET_CLS_POLICE
405         cl->q->_parent = sch;
406 #endif
407         if ((ret = cl->q->enqueue(skb, cl->q)) == 0) {
408             sch->q.qlen++;
409             sch->stats.packets++;
410             sch->stats.bytes += len;
411             cbq_mark_toplevel(q, cl);
412             if (!cl->next_alive)
413                 cbq_activate_class(cl);
414             return 0;
415         }
416     }
417
418     sch->stats.drops++;
419     if (cl == NULL)
420         kfree_skb(skb);
421     else {
422         cbq_mark_toplevel(q, cl);
423         cl->stats.drops++;
424     }
425     return ret;
426 }

```

Figure 967: CBQ - Enqueue: `linux/net/sched/sch_cbq.c`

```

428 static int
429 cbq_requeue(struct sk_buff *skb, struct Qdisc *sch)
430 {
431     struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
432     struct cbq_class *cl;
433     int ret;
434
435     if ((cl = q->tx_class) == NULL) {
436         kfree_skb(skb);
437         sch->stats.drops++;
438         return NET_XMIT_CN;
439     }
440     q->tx_class = NULL;
441
442     cbq_mark_toplevel(q, cl);
443
444 #ifdef CONFIG_NET_CLS_POLICE
445     q->rx_class = cl;
446     cl->q->_parent = sch;
447 #endif
448     if ((ret = cl->q->ops->requeue(skb, cl->q)) == 0) {
449         sch->q.len++;
450         if (!cl->next_alive)
451             cbq_activate_class(cl);
452         return 0;
453     }
454     sch->stats.drops++;
455     cl->stats.drops++;
456     return ret;
457 }

```

Figure 968: CBQ - Requeue: `linux/net/sched/sch_cbq.c`

```

459 /* Overlimit actions */
460
461 /* TC_CBQ_OVL_CLASSIC: (default) penalize leaf class by adding offtime */
462
463 static void cbq_ovl_classic (struct cbq_class *cl)
464 {
465     struct cbq_sched_data *q = (struct cbq_sched_data *)cl->qdisc->data;
466     psched_tdiff_t delay = PSCHED_TDIFF(cl->undertime, q->now);
467
468     if (!cl->delayed) {
469         delay += cl->offtime;
470
471         /*
472          * Class goes to sleep, so that it will have no
473          * chance to work avgidle. Let's forgive it 8)
474
475          * BTW cbq-2.0 has a crap in this
476          * place, apparently they forgot to shift it by cl->ewma_log.
477          */
478         if (cl->avgidle < 0)
479             delay -= (-cl->avgidle) - ((-cl->avgidle) >> cl->ewma_log);
480         if (cl->avgidle < cl->minidle)
481             cl->avgidle = cl->minidle;
482         if (delay <= 0)
483             delay = 1;
484         PSCHED_TADD2(q->now, delay, cl->undertime);
485
486         cl->xstats.overactions++;
487         cl->delayed = 1;
488     }
489     if (q->wd_expires == 0 || q->wd_expires > delay)
490         q->wd_expires = delay;
491
492     /* Dirty work! We must schedule wakeups based on
493      * real available rate, rather than leaf rate,
494      * which may be tiny (even zero).
495      */
496     if (q->toplevel == TC_CBQ_MAXLEVEL) {
497         struct cbq_class *b;
498         psched_tdiff_t base_delay = q->wd_expires;
499
500         for (b = cl->borrow; b; b = b->borrow) {
501             delay = PSCHED_TDIFF(b->undertime, q->now);
502             if (delay < base_delay) {
503                 if (delay <= 0)
504                     delay = 1;
505                 base_delay = delay;
506             }
507         }
508         q->wd_expires = base_delay;
509     }
510 }
511

```

Figure 969: CBQ - overlimit: linux/net/sched/sch_cbq.c

```

513 /* TC_CBQ_OVL_RCLASSIC: penalize by offtime classes in hierarchy, when
514    they go overlimit
515    */
516
517 static void cbq_ovl_rclassic (struct cbq_class *cl)
518 {
519     struct cbq_sched_data *q = (struct cbq_sched_data *)cl->qdisc->data;
520     struct cbq_class *this = cl;
521
522     do {
523         if (cl->level > q->toplevel) {
524             cl = NULL;
525             break;
526         }
527     } while ((cl = cl->borrow) != NULL);
528
529     if (cl == NULL)
530         cl = this;
531     cbq_ovl_classic (cl);
532 }

```

Figure 970: CBQ - overlimit: reclassify: `linux/net/sched/sch_cbq.c`

```

534 /* TC_CBQ_OVL_DELAY: delay until it will go to underlimit */
535
536 static void cbq_ovl_delay(struct cbq_class *cl)
537 {
538     struct cbq_sched_data *q = (struct cbq_sched_data *)cl->qdisc->data;
539     psched_tdiff_t delay = PSCHED_TDIFFF(cl->undertime, q->now);
540
541     if (!cl->delayed) {
542         unsigned long sched = jiffies ;
543
544         delay += cl->offtime;
545         if (cl->avgidle < 0)
546             delay -= (-cl->avgidle) - ((-cl->avgidle) >> cl->ewma_log);
547         if (cl->avgidle < cl->minidle)
548             cl->avgidle = cl->minidle;
549         PSCHED_TADD2(q->now, delay, cl->undertime);
550
551         if (delay > 0) {
552             sched += PSCHED_US2JIFFIE(delay) + cl->penalty;
553             cl->penalized = sched;
554             cl->cpriority = TC_CBQ_MAXPRIO;
555             q->pmask |= (1 << TC_CBQ_MAXPRIO);
556             if (del_timer(&q->delay_timer) &&
557                 (long)(q->delay_timer.expires - sched) > 0)
558                 q->delay_timer.expires = sched;
559             add_timer(&q->delay_timer);
560             cl->delayed = 1;
561             cl->xstats.overactions++;
562             return;
563         }
564         delay = 1;
565     }
566     if (q->wd_expires == 0 || q->wd_expires > delay)
567         q->wd_expires = delay;
568 }

```

Figure 971: CBQ - overlimit: delay: linux/net/sched/sch_cbq.c

```

570 /* TC_CBQ_OVL_LOWPRIO: penalize class by lowering its priority band */
571
572 static void cbq_ovl_lowprio(struct cbq_class *cl)
573 {
574     struct cbq_sched_data *q = (struct cbq_sched_data*)cl->qdisc->data;
575
576     cl->penalized = jiffies + cl->penalty;
577
578     if (cl->cpriority != cl->priority2) {
579         cl->cpriority = cl->priority2;
580         q->pmask |= (1<<cl->cpriority);
581         cl->xstats.overactions++;
582     }
583     cbq_ovl_classic (cl);
584 }
585
586 /* TC_CBQ_OVL_DROP: penalize class by dropping */
587
588 static void cbq_ovl_drop(struct cbq_class *cl)
589 {
590     if (cl->q->ops->drop)
591         if (cl->q->ops->drop(cl->q))
592             cl->qdisc->q.qlen--;
593     cl->xstats.overactions++;
594     cbq_ovl_classic (cl);
595 }

```

Figure 972: CBQ - overlimit: lower priority or even drop:
linux/net/sched/sch_cbq.c

```
597 static void cbq_watchdog(unsigned long arg)
598 {
599     struct Qdisc *sch = (struct Qdisc*)arg;
600
601     sch->flags &= ~TCQ_F_THROTTLED;
602     netif_schedule(sch->dev);
603 }
```

Figure 973: CBQ - watchdog timer: `linux/net/sched/sch_cbq.c`

```

605 static unsigned long cbq_undelay_prio(struct cbq_sched_data *q, int prio)
606 {
607     struct cbq_class *cl;
608     struct cbq_class *cl_prev = q->active[prio];
609     unsigned long now = jiffies;
610     unsigned long sched = now;
611
612     if (cl_prev == NULL)
613         return now;
614
615     do {
616         cl = cl_prev->next_alive;
617         if ((long)(now - cl->penalized) > 0) {
618             cl_prev->next_alive = cl->next_alive;
619             cl->next_alive = NULL;
620             cl->cpriority = cl->priority;
621             cl->delayed = 0;
622             cbq_activate_class(cl);
623
624             if (cl == q->active[prio]) {
625                 q->active[prio] = cl_prev;
626                 if (cl == q->active[prio]) {
627                     q->active[prio] = NULL;
628                     return 0;
629                 }
630             }
631
632             cl = cl_prev->next_alive;
633         } else if ((long)(sched - cl->penalized) > 0)
634             sched = cl->penalized;
635     } while ((cl_prev = cl) != q->active[prio]);
636
637     return (long)(sched - now);
638 }

```

Figure 974: CBQ - remove delay priority penalty: `linux/net/sched/sch_cbq.c`


```

640 static void cbq_undelay(unsigned long arg)
641 {
642     struct Qdisc *sch = (struct Qdisc*)arg;
643     struct cbq_sched_data *q = (struct cbq_sched_data*)sch->data;
644     long delay = 0;
645     unsigned pmask;
646
647     pmask = q->pmask;
648     q->pmask = 0;
649
650     while (pmask) {
651         int prio = ffz(~pmask);
652         long tmp;
653
654         pmask &= ~(1<<prio);
655
656         tmp = cbq_undelay_prio(q, prio);
657         if (tmp > 0) {
658             q->pmask |= 1<<prio;
659             if (tmp < delay || delay == 0)
660                 delay = tmp;
661         }
662     }
663
664     if (delay) {
665         q->delay_timer.expires = jiffies + delay;
666         add_timer(&q->delay_timer);
667     }
668
669     sch->flags &= ~TCQ_F_THROTTLED;
670     netif_schedule(sch->dev);
671 }

```

Figure 975: CBQ - remove delay penalty: `linux/net/sched/sch_cbq.c`

```

675 static int cbq_reshape_fail(struct sk_buff *skb, struct Qdisc *child)
676 {
677     int len = skb->len;
678     struct Qdisc *sch = child->__parent;
679     struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
680     struct cbq_class *cl = q->rx_class;
681
682     q->rx_class = NULL;
683
684     if (cl && (cl = cbq_reclassify(skb, cl)) != NULL) {
685         cbq_mark_toplevel(q, cl);
686
687         q->rx_class = cl;
688         cl->q->__parent = sch;
689
690         if (cl->q->enqueue(skb, cl->q) == 0) {
691             sch->q.qlen++;
692             sch->stats.packets++;
693             sch->stats.bytes+=len;
694             if (!cl->next_alive)
695                 cbq_activate_class(cl);
696             return 0;
697         }
698         sch->stats.drops++;
699         return 0;
700     }
701     sch->stats.drops++;
702     return -1;
703 }
704
705 }
706

```

Figure 976: CBQ - reshape: linux/net/sched/sch_cbq.c

```

709  /*
710     It is mission critical procedure.
711
712     We "regenerate" toplevel cutoff, if transmitting class
713     has backlog and it is not regulated. It is not part of
714     original CBQ description, but looks more reasonable.
715     Probably, it is wrong. This question needs further investigation.
716  */
717
718  static __inline__ void
719  cbq_update_toplevel(struct cbq_sched_data *q, struct cbq_class *cl,
720                    struct cbq_class *borrowed)
721  {
722      if (cl && q->toplevel >= borrowed->level) {
723          if (cl->q->q.len > 1) {
724              do {
725                  if (PSCHED_IS_PASTPERFECT(borrowed->undertime)
726                      ) {
727                      q->toplevel = borrowed->level;
728                      return;
729                  } while ((borrowed=borrowed->borrow) != NULL);
730              }
731  #if 0
732      /* It is not necessary now. Uncommenting it
733         will save CPU cycles, but decrease fairness.
734     */
735         q->toplevel = TC_CBQ_MAXLEVEL;
736  #endif
737      }
738  }

```

Figure 977: CBQ - update toplevel occupancy: linux/net/sched/sch_cbq.c

```

740 static void
741 cbq_update(struct cbq_sched_data *q)
742 {
743     struct cbq_class *this = q->tx_class;
744     struct cbq_class *cl = this;
745     int len = q->tx_len;
746
747     q->tx_class = NULL;
748
749     for (; cl; cl = cl->share) {
750         long avgidle = cl->avgidle;
751         long idle;
752
753         cl->stats.packets++;
754         cl->stats.bytes += len;
755
756         /*
757          * (now - last) is total time between packet right edges.
758          * (last_pktlen/rate) is "virtual" busy time, so that
759
760             idle = (now - last) - last_pktlen/rate
761          */
762
763         idle = PSCHED_TDIFF(q->now, cl->last);
764         if ((unsigned long)idle > 128*1024*1024) {
765             avgidle = cl->maxidle;
766         } else {
767             idle -= L2T(cl, len);
768
769             /* true_avgidle := (1-W)*true_avgidle + W*idle,
770              * where W=2^{-ewma_log}. But cl->avgidle is scaled:
771              * cl->avgidle == true_avgidle/W,
772              * hence:
773              */
774             avgidle += idle - (avgidle >> cl->ewma_log);
775         }
776     }

```

Figure 978: CBQ - update each level's occupancy:
linux/net/sched/sch_cbq.c

```

779         if (avgidle < cl->minidle)
780             avgidle = cl->minidle;
781
782
783         cl->avgidle = avgidle;
784
785         /* Calculate expected time, when this class
786            will be allowed to send.
787            It will occur, when:
788            (1-W)*true_avgidle + W*delay = 0, i.e.
789            idle = (1/W - 1)*(-true_avgidle)
790            or
791            idle = (1 - W)*(-cl->avgidle);
792         */
793         idle = (-avgidle) - ((-avgidle) >> cl->ewma_log);
794
795         /*
796            That is not all.
797            To maintain the rate allocated to the class,
798            we add to undertime virtual clock,
799            necessary to complete transmitted packet.
800            (len/phys_bandwidth has been already passed
801            to the moment of cbq_update)
802         */
803
804         idle -= L2T(&q->link, len);
805         idle += L2T(cl, len);
806
807         PSCHED_AUDIT_TDIF(idle);
808
809         PSCHED_TADD2(q->now, idle, cl->undertime);

```

Figure 979: CBQ - update each level's occupancy ii:
linux/net/sched/sch_cbq.c

```

810         } else {
811             /* Underlimit */
812
813             PSCHED_SET_PASTPERFECT(cl->undertime);
814             if (avgidle > cl->maxidle)
815                 cl->avgidle = cl->maxidle;
816             else
817                 cl->avgidle = avgidle;
818         }
819         cl->last = q->now;
820     }
821     cbq_update_toplevel(q, this, q->tx_borrowed);
822 }
823

```

Figure 980: CBQ - update each level's occupancy iii:
linux/net/sched/sch_cbq.c

```

825 static __inline__ struct cbq_class *
826 cbq_under_limit(struct cbq_class *cl)
827 {
828     struct cbq_sched_data *q = (struct cbq_sched_data*)cl->qdisc->data;
829     struct cbq_class *this_cl = cl;
830
831     if (cl->tparent == NULL)
832         return cl;
833
834     if (PSCHED_IS_PASTPERFECT(cl->undertime) ||
835         !PSCHED_TLESS(q->now, cl->undertime)) {
836         cl->delayed = 0;
837         return cl;
838     }
839
840     do {
841         /* It is very suspicious place. Now overlimit
842            action is generated for not bounded classes
843            only if link is completely congested.
844            Though it is in agree with ancestor-only paradigm,
845            it looks very stupid. Particularly,
846            it means that this chunk of code will either
847            never be called or result in strong amplification
848            of burstiness. Dangerous, silly, and, however,
849            no another solution exists.
850            */
851         if ((cl = cl->borrow) == NULL) {
852             this_cl->stats.overlimits++;
853             this_cl->overlimit(this_cl);
854             return NULL;
855         }
856         if (cl->level > q->toplevel)
857             return NULL;
858     } while (!PSCHED_IS_PASTPERFECT(cl->undertime) &&
859             PSCHED_TLESS(q->now, cl->undertime));
860
861     cl->delayed = 0;
862     return cl;
863 }

```

Figure 981: CBQ - underlimit actions: linux/net/sched/sch_cbq.c

```

865 static __inline__ struct sk_buff *
866 cbq_dequeue_prio(struct Qdisc *sch, int prio)
867 {
868     struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
869     struct cbq_class *cl_tail, *cl_prev, *cl;
870     struct sk_buff *skb;
871     int deficit;
872
873     cl_tail = cl_prev = q->active[prio];
874     cl = cl_prev->next_alive;
875
876     do {
877         deficit = 0;
878
879         /* Start round */
880         do {
881             struct cbq_class *borrow = cl;
882
883             if (cl->q->q.qlen &&
884                 (borrow = cbq_under_limit(cl)) == NULL)
885                 goto skip_class;
886
887             if (cl->deficit <= 0) {
888                 /* Class exhausted its allotment per
889                    this round. Switch to the next one.
890                    */
891                 deficit = 1;
892                 cl->deficit += cl->quantum;
893                 goto next_class;
894             }
895
896             skb = cl->q->dequeue(cl->q);

```

Figure 982: CBQ - dequeue by priorities i: `linux/net/sched/sch_cbq.c`


```

898      /* Class did not give us any skb :-(  

899      It could occur even if cl->q->q.qlen != 0  

900      f.e. if cl->q == "tbf"  

901      */  

902      if (skb == NULL)  

903          goto skip_class ;  

904  

905      cl->deficit -= skb->len;  

906      q->tx_class = cl;  

907      q->tx_borrowed = borrow;  

908      if (borrow != cl) {  

909  #ifndef CBQ_XSTATS_BORROWS_BYTES  

910          borrow->xstats.borrowed++;  

911          cl->xstats.borrowed++;  

912  #else  

913          borrow->xstats.borrowed += skb->len;  

914          cl->xstats.borrowed += skb->len;  

915  #endif  

916      }  

917      q->tx_len = skb->len;  

918  

919      if (cl->deficit <= 0) {  

920          q->active[prio] = cl;  

921          cl = cl->next_alive;  

922          cl->deficit += cl->quantum;  

923      }  

924      return skb;

```

Figure 983: CBQ - dequeue by priorities ii: `linux/net/sched/sch_cbq.c`

```

926 skip_class :
927     if (cl->q->q.qlen == 0 || prio != cl->cpriority) {
928         /* Class is empty or penalized.
929          * Unlink it from active chain.
930          */
931         cl_prev->next_alive = cl->next_alive;
932         cl->next_alive = NULL;
933
934         /* Did cl_tail point to it? */
935         if (cl == cl_tail) {
936             /* Repair it! */
937             cl_tail = cl_prev;
938
939             /* Was it the last class in this band? */
940             if (cl == cl_tail) {
941                 /* Kill the band! */
942                 q->active[prio] = NULL;
943                 q->activemask &= ~(1<<prio);
944                 if (cl->q->q.qlen)
945                     cbq_activate_class (cl);
946                 return NULL;
947             }

```

Figure 984: CBQ - dequeue by priorities iii: linux/net/sched/sch_cbq.c

```

949         q->active[prio] = cl_tail;
950     }
951     if (cl->q->q.len)
952         cbq_activate_class (cl);
953
954     cl = cl_prev;
955 }
956
957 next_class:
958     cl_prev = cl;
959     cl = cl->next_alive;
960     } while (cl_prev != cl_tail);
961 } while ( deficit );
962
963 q->active[prio] = cl_prev;
964
965 return NULL;
966 }

```

Figure 985: CBQ - dequeue by priorities iv: `linux/net/sched/sch_cbq.c`

```

968 static __inline__ struct sk_buff *
969 cbq_dequeue_1(struct Qdisc *sch)
970 {
971     struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
972     struct sk_buff *skb;
973     unsigned activemask;
974
975     activemask = q->activemask&0xFF;
976     while (activemask) {
977         int prio = ffz(~activemask);
978         activemask &= ~(1<<prio);
979         skb = cbq_dequeue_prio(sch, prio);
980         if (skb)
981             return skb;
982     }
983     return NULL;
984 }

```

Figure 986: CBQ - dequeue 1: `linux/net/sched/sch_cbq.c`

```

986 static struct sk_buff *
987 cbq_dequeue(struct Qdisc *sch)
988 {
989     struct sk_buff *skb;
990     struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
991     psched_time_t now;
992     psched_tdiff_t incr;
993
994     PSCHED_GET_TIME(now);
995     incr = PSCHED_TDIFF(now, q->now_rt);
996
997     if (q->tx_class) {
998         psched_tdiff_t incr2;
999         /* Time integrator. We calculate EOS time
1000          * by adding expected packet transmission time.
1001          * If real time is greater, we warp artificial clock,
1002          * so that:
1003
1004          * cbq_time = max(real_time, work);
1005          */
1006         incr2 = L2T(&q->link, q->tx_len);
1007         PSCHED_TADD(q->now, incr2);
1008         cbq_update(q);
1009         if ((incr -= incr2) < 0)
1010             incr = 0;
1011     }
1012     PSCHED_TADD(q->now, incr);
1013     q->now_rt = now;

```

Figure 987: CBQ - main dequeue function: `linux/net/sched/sch_cbq.c`

```

1015     for (;;) {
1016         q->wd_expires = 0;
1017
1018         skb = cbq_dequeue_1(sch);
1019         if (skb) {
1020             sch->q_len--;
1021             sch->flags &= ~TCQ_F_THROTTLED;
1022             return skb;
1023         }
1024
1025         /* All the classes are overlimit.
1026
1027            It is possible, if:
1028
1029            1. Scheduler is empty.
1030            2. Toplevel cutoff inhibited borrowing.
1031            3. Root class is overlimit.
1032
1033            Reset 2d and 3d conditions and retry.
1034
1035            Note, that NS and cbq-2.0 are buggy, peeking
1036            an arbitrary class is appropriate for ancestor-only
1037            sharing, but not for toplevel algorithm.
1038
1039            Our version is better, but slower, because it requires
1040            two passes, but it is unavoidable with top-level sharing.
1041         */
1042
1043         if (q->toplevel == TC_CBQ_MAXLEVEL &&
1044             PSCHED_IS_PASTPERFECT(q->link.undertime))
1045             break;
1046
1047         q->toplevel = TC_CBQ_MAXLEVEL;
1048         PSCHED_SET_PASTPERFECT(q->link.undertime);
1049     }

```

Figure 988: CBQ - main dequeue function: `linux/net/sched/sch_cbq.c`

```

1051     /* No packets in scheduler or nobody wants to give them to us :-(  

1052     Sigh... start watchdog timer in the last case. */  

1053  

1054     if (sch->q.qlen) {  

1055         sch->stats.overlimits++;  

1056         if (q->wd_expires && !netif_queue_stopped(sch->dev)) {  

1057             long delay = PSCHED_US2JIFFIE(q->wd_expires);  

1058             del_timer(&q->wd_timer);  

1059             if (delay <= 0)  

1060                 delay = 1;  

1061             q->wd_timer.expires = jiffies + delay;  

1062             add_timer(&q->wd_timer);  

1063             sch->flags |= TCQ_F_THROTTLED;  

1064         }  

1065     }  

1066     return NULL;  

1067 }

```

Figure 989: CBQ - main dequeue function: `linux/net/sched/sch_cbq.c`

```

1069 /* CBQ class maintenance routines */
1070
1071 static void cbq_adjust_levels(struct cbq_class *this)
1072 {
1073     if (this == NULL)
1074         return;
1075
1076     do {
1077         int level = 0;
1078         struct cbq_class *cl;
1079
1080         if ((cl = this->children) != NULL) {
1081             do {
1082                 if (cl->level > level)
1083                     level = cl->level;
1084             } while ((cl = cl->sibling) != this->children);
1085         }
1086         this->level = level+1;
1087     } while ((this = this->tparent) != NULL);

```

Figure 990: CBQ - Maintenance Functions: `linux/net/sched/sch_cbq.c`


```

1394 static int cbq_init(struct Qdisc *sch, struct rtattr *opt)
1395 {
1396     struct cbq_sched_data *q = (struct cbq_sched_data*)sch->data;
1397     struct rtattr *tb[TCA_CBQ_MAX];
1398     struct tc_ratespec *r;
1399
1400     if (rtattr_parse(tb, TCA_CBQ_MAX, RTA_DATA(opt), RTA_PAYLOAD(opt))
1401         < 0 ||
1402         tb[TCA_CBQ_RTAB-1] == NULL || tb[TCA_CBQ_RATE-1] == NULL ||
1403         RTA_PAYLOAD(tb[TCA_CBQ_RATE-1]) < sizeof(struct tc_ratespec))
1404         return -EINVAL;
1405
1406     if (tb[TCA_CBQ_LSSOPT-1] &&
1407         RTA_PAYLOAD(tb[TCA_CBQ_LSSOPT-1]) < sizeof(struct tc_cbq_lssopt))
1408         return -EINVAL;
1409
1410     r = RTA_DATA(tb[TCA_CBQ_RATE-1]);
1411
1412     MOD_INC_USE_COUNT;
1413     if ((q->link.R_tab = qdisc_get_rtab(r, tb[TCA_CBQ_RTAB-1])) == NULL) {
1414         MOD_DEC_USE_COUNT;
1415         return -EINVAL;
1416     }

```

Figure 991: CBQ - Initialise CBQ: `linux/net/sched/sch_cbq.c`

```

1417     q->link.refcnt = 1;
1418     q->link.sibling = &q->link;
1419     q->link.classid = sch->handle;
1420     q->link.qdisc = sch;
1421     if (!(q->link.q = qdisc_create_dflt(sch->dev, &pfifo_qdisc_ops)))
1422         q->link.q = &noop_qdisc;
1423
1424     q->link.priority = TC_CBQ_MAXPRIO-1;
1425     q->link.priority2 = TC_CBQ_MAXPRIO-1;
1426     q->link.cpriority = TC_CBQ_MAXPRIO-1;
1427     q->link.ovl_strategy = TC_CBQ_OVL_CLASSIC;
1428     q->link.overlimit = cbq_ovl_classic;
1429     q->link.allot = psched_mtu(sch->dev);
1430     q->link.quantum = q->link.allot;
1431     q->link.weight = q->link.Rtab->rate.rate;
1432
1433     q->link.ewma_log = TC_CBQ_DEF_EWMA;
1434     q->link.avpkt = q->link.allot/2;
1435     q->link.minidle = -0x7FFFFFFF;
1436     q->link.stats.lock = &sch->dev->queue_lock;

```

Figure 992: CBQ - Initialise CBQ ii: linux/net/sched/sch_cbq.c

```

1438     init_timer (&q->wd_timer);
1439     q->wd_timer.data = (unsigned long)sch;
1440     q->wd_timer.function = cbq_watchdog;
1441     init_timer (&q->delay_timer);
1442     q->delay_timer.data = (unsigned long)sch;
1443     q->delay_timer.function = cbq_undelay;
1444     q->toplevel = TC_CBQ_MAXLEVEL;
1445     PSCHED_GET_TIME(q->now);
1446     q->now_rt = q->now;
1447
1448     cbq_link_class (&q->link);
1449
1450     if (tb[TCA_CBQ_LSSOPT-1])
1451         cbq_set_lass (&q->link, RTA_DATA(tb[TCA_CBQ_LSSOPT-1]));
1452
1453     cbq_addprio(q, &q->link);
1454     return 0;
1455 }

```

Figure 993: CBQ - Initialise CBQ iii: `linux/net/sched/sch_cbq.c`

```

1660 static int cbq_graft(struct Qdisc *sch, unsigned long arg, struct Qdisc *new,
1661                   struct Qdisc **old)
1662 {
1663     struct cbq_class *cl = (struct cbq_class*)arg;
1664
1665     if (cl) {
1666         if (new == NULL) {
1667             if ((new = qdisc_create_dflt(sch->dev, &pfifo_qdisc_ops) ==
1668                 NULL)
1669                 return -ENOBUFS;
1670             } else {
1671 #ifdef CONFIG_NET_CLS_POLICE
1672                 if (cl->police == TC_POLICE_RECLASSIFY)
1673                     new->reshape_fail = cbq_reshape_fail;
1674 #endif
1675             }
1676         sch_tree_lock(sch);
1677         *old = cl->q;
1678         cl->q = new;
1679         qdisc_reset(*old);
1680         sch_tree_unlock(sch);
1681
1682         return 0;
1683     }
1684     return -ENOENT;
1685 }

```

Figure 994: CBQ - Grafting and pruning classes: `linux/net/sched/sch_cbq.c`

```

1686 static struct Qdisc *
1687 cbq_leaf(struct Qdisc *sch, unsigned long arg)
1688 {
1689     struct cbq_class *cl = (struct cbq_class*)arg;
1690
1691     return cl ? cl->q : NULL;
1692 }
1693
1694 static unsigned long cbq_get(struct Qdisc *sch, u32 classid)
1695 {
1696     struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
1697     struct cbq_class *cl = cbq_class_lookup(q, classid);
1698
1699     if (cl) {
1700         cl->refcnt++;
1701         return (unsigned long)cl;
1702     }
1703     return 0;
1704 }

```

Figure 995: CBQ - Grafting and pruning classes ii:
linux/net/sched/sch_cbq.c

```

1706 static void cbq_destroy_filters (struct cbq_class *cl)
1707 {
1708     struct tcf_proto *tp;
1709
1710     while ((tp = cl->filter_list) != NULL) {
1711         cl->filter_list = tp->next;
1712         tp->ops->destroy(tp);
1713     }
1714 }
1715
1716 static void cbq_destroy_class(struct cbq_class *cl)
1717 {
1718     cbq_destroy_filters (cl);
1719     qdisc_destroy (cl->q);
1720     qdisc_put_rtab (cl->R_rtab);
1721 #ifdef CONFIG_NET_ESTIMATOR
1722     qdisc_kill_estimator (&cl->stats);
1723 #endif
1724     kfree (cl);
1725 }

```

Figure 996: CBQ - Grafting and pruning classes iii:
linux/net/sched/sch_cbq.c

```

1770         cbq_destroy_class (cl);
1771     }
1772 }
1773
1774 static int
1775 cbq_change_class(struct Qdisc *sch, u32 classid, u32 parentid, struct rtattr **tca,
1776                 unsigned long *arg)
1777 {
1778     int err;
1779     struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
1780     struct cbq_class *cl = (struct cbq_class*)arg;
1781     struct rtattr *opt = tca[TCA_OPTIONS-1];
1782     struct rtattr *tb[TCA_CBQ_MAX];
1783     struct cbq_class *parent;
1784     struct qdisc_rate_table *rtab = NULL;
1785
1786     if (opt==NULL ||
1787         rtattr_parse(tb, TCA_CBQ_MAX, RTA_DATA(opt), RTA_PAYLOAD(opt)))
1788         return -EINVAL;
1789
1790     if (tb[TCA_CBQ_OVL_STRATEGY-1] &&
1791         RTA_PAYLOAD(tb[TCA_CBQ_OVL_STRATEGY-1]) < sizeof(struct
1792             tc_cbq_ovl))
1793         return -EINVAL;
1794
1795     if (tb[TCA_CBQ_FOPT-1] &&
1796         RTA_PAYLOAD(tb[TCA_CBQ_FOPT-1]) < sizeof(struct tc_cbq_fopt))
1797         return -EINVAL;
1798
1799     if (tb[TCA_CBQ_RATE-1] &&
1800         RTA_PAYLOAD(tb[TCA_CBQ_RATE-1]) < sizeof(struct tc_ratespec))
1801         return -EINVAL;
1802
1803     if (tb[TCA_CBQ_LSSOPT-1] &&
1804         RTA_PAYLOAD(tb[TCA_CBQ_LSSOPT-1]) < sizeof(struct tc_cbq_lassopt))
1805         return -EINVAL;
1806
1807     if (tb[TCA_CBQ_WRROPT-1] &&
1808         RTA_PAYLOAD(tb[TCA_CBQ_WRROPT-1]) < sizeof(struct tc_cbq_wrropt))
1809         return -EINVAL;
1810
1811     #ifdef CONFIG_NET_CLS_POLICE
1812     if (tb[TCA_CBQ_POLICE-1] &&
1813         RTA_PAYLOAD(tb[TCA_CBQ_POLICE-1]) < sizeof(struct tc_cbq_police))
1814         return -EINVAL;
1815     #endif
1816
1817     if (cl) {
1818         /* Check parent */
1819         if (parentid) {
1820             if (cl->tparent && cl->tparent->classid != parentid)
1821                 return -EINVAL;
1822             if (!cl->tparent && parentid != TC_H_ROOT)
1823                 return -EINVAL;
1824         }
1825
1826         if (tb[TCA_CBQ_RATE-1]) {

```

Figure 997: CBQ: linux/net/sched/sch_cbq.c

```

1827         if (rtab == NULL)
1828             return -EINVAL;
1829     }
1830
1831     /* Change class parameters */
1832     sch_tree_lock(sch);
1833
1834     if (cl->next_alive != NULL)
1835         cbq_deactivate_class(cl);
1836
1837     if (rtab) {
1838         rtab = xchg(&cl->Rtab, rtab);
1839         qdisc_put_rtab(rtab);
1840     }
1841
1842     if (tb[TCA_CBQ_LSSOPT-1])
1843         cbq_set_lass(cl, RTA_DATA(tb[TCA_CBQ_LSSOPT-1]));
1844
1845     if (tb[TCA_CBQ_WRROPT-1]) {
1846         cbq_rmprio(q, cl);
1847         cbq_set_wrrr(cl, RTA_DATA(tb[TCA_CBQ_WRROPT-1]));
1848     }
1849
1850     if (tb[TCA_CBQ_OVL_STRATEGY-1])
1851         cbq_set_overlimit(cl, RTA_DATA(tb[
1852             TCA_CBQ_OVL_STRATEGY-1]));
1853
1854     #ifdef CONFIG_NET_CLS_POLICE
1855     if (tb[TCA_CBQ_POLICE-1])
1856         cbq_set_police(cl, RTA_DATA(tb[TCA_CBQ_POLICE-1]));
1857     #endif
1858
1859     if (tb[TCA_CBQ_FOPT-1])
1860         cbq_set_fopt(cl, RTA_DATA(tb[TCA_CBQ_FOPT-1]));

```

Figure 998: CBQ: linux/net/sched/sch_cbq.c


```

1862         cbq_activate_class (cl);
1863
1864         sch_tree_unlock (sch);
1865
1866 #ifdef CONFIG_NET_ESTIMATOR
1867         if (tca[TCA_RATE-1]) {
1868             qdisc_kill_estimator (&cl->stats);
1869             qdisc_new_estimator(&cl->stats, tca[TCA_RATE-1]);
1870         }
1871 #endif
1872         return 0;
1873     }
1874
1875     if (parentid == TC_H_ROOT)
1876         return -EINVAL;
1877
1878     if (tb[TCA_CBQ_WRROPT-1] == NULL || tb[TCA_CBQ_RATE-1] == NULL
1879         ||
1880         tb[TCA_CBQ_LSSOPT-1] == NULL)
1881         return -EINVAL;
1882
1883     rtab = qdisc_get_rtab(RTA_DATA(tb[TCA_CBQ_RATE-1]), tb[TCA_CBQ_RTAB
1884         -1]);
1885     if (rtab == NULL)
1886         return -EINVAL;
1887
1888     if (classid) {
1889         err = -EINVAL;
1890         if (TC_H_MAJ(classid^sch->handle) || cbq_class_lookup(q, classid))
1891             goto failure;
1892     } else {
1893         int i;
1894         classid = TC_H_MAKE(sch->handle, 0x8000);
1895
1896         for (i=0; i<0x8000; i++) {
1897             if (++q->hgenerator >= 0x8000)
1898                 q->hgenerator = 1;
1899             if (cbq_class_lookup(q, classid | q->hgenerator) == NULL)
1900                 break;
1901         }
1902         err = -ENOSR;

```

Figure 999: CBQ: linux/net/sched/sch_cbq.c

```

1902         goto failure ;
1903         classid = classid |q->hgenerator;
1904     }
1905
1906     parent = &q->link;
1907     if (parentid) {
1908         parent = cbq_class_lookup(q, parentid);
1909         err = -EINVAL;
1910         if (parent == NULL)
1911             goto failure ;
1912     }
1913
1914     err = -ENOBUFS;
1915     cl = kmalloc(sizeof(*cl), GFP_KERNEL);
1916     if (cl == NULL)
1917         goto failure ;
1918     memset(cl, 0, sizeof(*cl));
1919     cl->R_ttab = rtab;
1920     rtab = NULL;
1921     cl->refcnt = 1;
1922     if (!(cl->q = qdisc_create_dflt(sch->dev, &pfifo_qdisc_ops)))
1923         cl->q = &noop_qdisc;
1924     cl->classid = classid;
1925     cl->tparent = parent;
1926     cl->qdisc = sch;
1927     cl->allot = parent->allot;
1928     cl->quantum = cl->allot;
1929     cl->weight = cl->R_ttab->rate.rate;
1930     cl->stats.lock = &sch->dev->queue_lock;
1931
1932     sch_tree_lock(sch);
1933     cbq_link_class(cl);
1934     cl->borrow = cl->tparent;
1935     if (cl->tparent != &q->link)
1936         cl->share = cl->tparent;
1937     cbq_adjust_levels(parent);
1938     cl->minidle = -0x7FFFFFFF;
1939     cbq_set_lss(cl, RTA_DATA(tb[TCA_CBQ_LSSOPT-1]));
1940     cbq_set_wrr(cl, RTA_DATA(tb[TCA_CBQ_WRROPT-1]));
1941     if (cl->ewma_log==0)
1942         cl->ewma_log = q->link.ewma_log;
1943     if (cl->maxidle==0)
1944         cl->maxidle = q->link.maxidle;
1945     if (cl->avpkt==0)

```

Figure 1000: CBQ: `linux/net/sched/sch_cbq.c`

```

1946         cl->avpkt = q->link.avpkt;
1947         cl->overlimit = cbq_ovl_classic;
1948         if (tb[TCA_CBQ_OVL_STRATEGY-1])
1949             cbq_set_overlimit (cl , RTA_DATA(tb[TCA_CBQ_OVL_STRATEGY-1]));
1950 #ifdef CONFIG_NET_CLS_POLICE
1951         if (tb[TCA_CBQ_POLICE-1])
1952             cbq_set_police (cl , RTA_DATA(tb[TCA_CBQ_POLICE-1]));
1953 #endif
1954         if (tb[TCA_CBQ_FOPT-1])
1955             cbq_set_fopt (cl , RTA_DATA(tb[TCA_CBQ_FOPT-1]));
1956         sch_tree_unlock (sch);
1957
1958 #ifdef CONFIG_NET_ESTIMATOR
1959         if (tca[TCA_RATE-1])
1960             qdisc_new_estimator(&cl->stats, tca[TCA_RATE-1]);
1961 #endif
1962
1963         *arg = (unsigned long)cl;
1964         return 0;

```

Figure 1001: CBQ: linux/net/sched/sch_cbq.c

```

1967     qdisc_put_rtab(rtab);
1968     return err;
1969 }
1970
1971 static int cbq_delete(struct Qdisc *sch, unsigned long arg)
1972 {
1973     struct cbq_sched_data *q = (struct cbq_sched_data *)sch->data;
1974     struct cbq_class *cl = (struct cbq_class*)arg;
1975
1976     if (cl->filters || cl->children || cl == &q->link)
1977         return -EBUSY;
1978
1979     sch_tree_lock(sch);
1980
1981     if (cl->next_alive)
1982         cbq_deactivate_class(cl);
1983
1984     if (q->tx_borrowed == cl)
1985         q->tx_borrowed = q->tx_class;
1986     if (q->tx_class == cl) {
1987         q->tx_class = NULL;
1988         q->tx_borrowed = NULL;
1989     }
1990 #ifdef CONFIG_NET_CLS_POLICE
1991     if (q->rx_class == cl)
1992         q->rx_class = NULL;
1993 #endif
1994
1995     cbq_unlink_class(cl);
1996     cbq_adjust_levels(cl->tparent);
1997     cl->defmap = 0;
1998     cbq_sync_defmap(cl);
1999
2000     cbq_rmprio(q, cl);
2001     sch_tree_unlock(sch);
2002
2003     if (--cl->refcnt == 0)

```

Figure 1002: CBQ - delete CBQ from Qdisc : `linux/net/sched/sch_cbq.c`

```

40  /*      Clark-Shenker-Zhang algorithm.
41  =====
42
43  SOURCE.
44
45  David D. Clark, Scott Shenker and Lixia Zhang
46  "Supporting Real-Time Applications in an Integrated Services Packet
47  Network: Architecture and Mechanism".
48
49  CBQ presents a flexible universal algorithm for packet scheduling,
50  but it has pretty poor delay characteristics.
51  Round-robin scheduling and link-sharing goals
52  apparently contradict minimization of network delay and jitter.
53  Moreover, correct handling of predictive flows seems to be
54  impossible in CBQ.
55
56  CSZ presents a more precise but less flexible and less efficient
57  approach. As I understand it, the main idea is to create
58  WFQ flows for each guaranteed service and to allocate
59  the rest of bandwidth to dummy flow-0. Flow-0 comprises
60  the predictive services and the best effort traffic;
61  it is handled by a priority scheduler with the highest
62  priority band allocated for predictive services, and the rest ---
63  to the best effort packets.
64
65  Note that in CSZ flows are NOT limited to their bandwidth. It
66  is supposed that the flow passed admission control at the edge
67  of the QoS network and it doesn't need further shaping. Any
68  attempt to improve the flow or to shape it to a token bucket
69  at intermediate hops will introduce undesired delays and raise
70  jitter.
71
72  At the moment CSZ is the only scheduler that provides
73  true guaranteed service. Another schemes (including CBQ)
74  do not provide guaranteed delay and randomize jitter.
75  There is a proof (Sally Floyd), that delay
76  can be estimated by a IntServ compliant formula.
77  This result is true formally, but it is wrong in principle.
78  It takes into account only round-robin delays,
79  ignoring delays introduced by link sharing i.e. overlimiting.
80  Note that temporary overlimits are inevitable because
81  real links are not ideal, and the real algorithm must take this
82  into account.

```

Figure 1003: Clark/Shenker/Zhang Scheduling - Initial Comment:
linux/net/sched/sch_csz.c

```

86      --- Notations.
87
88      $B$ is link bandwidth (bits/sec).
89
90      $I$ is set of all flows, including flow $0$.
91      Every flow $a \in I$ has associated bandwidth slice $r_a < 1$ and
92      $\sum_{a \in I} r_a = 1$.
93
94      --- Flow model.
95
96      Let $m_a$ is the number of backlogged bits in flow $a$.
97      The flow is {\em active}, if $m_a > 0$.
98      This number is a discontinuous function of time;
99      when a packet $i$ arrives:
100     \[
101     m_a(t_i+0) - m_a(t_i-0) = L^i,
102     \]
103     where $L^i$ is the length of the arrived packet.
104     The flow queue is drained continuously until $m_a == 0$:
105     \[
106     \{d m_a \over dt\} = - \{ B r_a \over \sum_{b \in A} r_b \}.
107     \]
108     I.e. flow rates are their allocated rates proportionally
109     scaled to take all available link bandwidth. Apparently,
110     it is not the only possible policy. F.e. CBQ classes
111     without borrowing would be modelled by:
112     \[
113     \{d m_a \over dt\} = - B r_a .
114     \]
115     More complicated hierarchical bandwidth allocation
116     policies are possible, but unfortunately, the basic
117     flow equations have a simple solution only for proportional
118     scaling.

```

Figure 1004: CSZ Notation: `linux/net/sched/sch_csz.c`

```

120     --- Departure times.
121
122     We calculate the time until the last bit of packet is sent :
123     \[
124     E_a^i(t) = { m_a(t_i) - \delta_a(t) \over r_a },
125     \]
126     where  $\delta_a(t)$  is number of bits drained since  $t_i$  .
127     We have to evaluate  $E_a^i$  for all queued packets,
128     then find the packet with minimal  $E_a^i$  and send it.
129
130     This sounds good, but direct implementation of the algorithm
131     is absolutely infeasible . Luckily, if flow rates
132     are scaled proportionally , the equations have a simple solution .
133
134     The differential equation for  $E_a^i$  is
135     \[
136     {d E_a^i (t) \over dt} = - { d \delta_a(t) \over dt} { 1 \over r_a} =
137     { B \over \sum_{b \in A} r_b}
138     \]
139     with initial condition
140     \[
141     E_a^i ( t_i) = { m_a(t_i) \over r_a } .
142     \]
143
144     Let's introduce an auxiliary function  $R(t)$ :

```

Figure 1005: CSZ Departure Times: `linux/net/sched/sch_csz.c`

```

146      --- Round number.
147
148      Consider the following model: we rotate over active flows,
149      sending  $r_a B$  bits from every flow, so that we send
150       $B \sum_{a \in A} r_a$  bits per round, that takes
151       $\sum_{a \in A} r_a$  seconds.
152
153      Hence,  $R(t)$  (round number) is a monotonically increasing
154      linear function of time when  $A$  is not changed
155      \[
156      \frac{d R(t)}{dt} = \frac{1}{\sum_{a \in A} r_a}
157      \]
158      and it is continuous when  $A$  changes.
159
160      The central observation is that the quantity
161       $F_a^i = R(t) + E_a^i(t)/B$  does not depend on time at all!
162       $R(t)$  does not depend on flow, so that  $F_a^i$  can be
163      calculated only once on packet arrival, and we need not
164      recalculate  $E$  numbers and resorting queues.
165      The number  $F_a^i$  is called finish number of the packet.
166      It is just the value of  $R(t)$  when the last bit of packet
167      is sent out.
168
169      Maximal finish number on flow is called finish number of flow
170      and minimal one is "start_number_of_flow".
171      Apparently, flow is active if and only if  $F_a \leq R$ .

```

Figure 1006: CSZ - The Round: linux/net/sched/sch_cs.c


```
173     When a packet of length  $L_i$  bit arrives to flow  $s_a$  at time  $t_i$ ,
174     we calculate  $F_a^i$  as:
175
176     If flow was inactive ( $F_a < R$ ):
177      $F_a^i = R(t) + \{L_i \over B r_a\}$ 
178     otherwise
179      $F_a^i = F_a + \{L_i \over B r_a\}$ 
180
181     These equations complete the algorithm specification.
```

Figure 1007: CSZ - The Round: `linux/net/sched/sch_csz.c`

```
183     It looks pretty hairy, but there is a simple
184     procedure for solving these equations.
185     See procedure csz_update(), that is a generalization of
186     the algorithm from S. Keshav's thesis Chapter 3
187     "Efficient Implementation of Fair Queueing".
188
189     NOTES.
190
191     * We implement only the simplest variant of CSZ,
192     when flow-0 is a explicit 4band priority fifo .
193     This is bad, but we need a "peek" operation in addition
194     to "dequeue" to implement complete CSZ.
195     I do not want to do that, unless it is absolutely
196     necessary.
```

Figure 1008: CSZ Discussion: `linux/net/sched/sch_csz.c`

```
198 * A primitive support for token bucket filtering
199 presents itself too. It directly contradicts CSZ, but
200 even though the Internet is on the globe ... :-)
201 "the_edges_of_the_network" really exist .
202
203 BUGS.
204
205 * Fixed point arithmetic is overcomplicated, suboptimal and even
206 wrong. Check it later. */
```

Figure 1009: CSZ Discussion: `linux/net/sched/sch_csz.c`

```
37 /* 1 band FIFO pseudo-scheduler */
38
39 struct fifo_sched_data
40 {
41     unsigned limit;
42 };
```

Figure 1010: FIFO Queueing Scheduler - data: `linux/net/sched/sch_fifo.c`

```

44 static int
45 bfifo_enqueue(struct sk_buff *skb, struct Qdisc* sch)
46 {
47     struct fifo_sched_data *q = (struct fifo_sched_data *)sch->data;
48
49     if (sch->stats.backlog <= q->limit) {
50         __skb_queue_tail(&sch->q, skb);
51         sch->stats.backlog += skb->len;
52         sch->stats.bytes += skb->len;
53         sch->stats.packets++;
54         return 0;
55     }
56     sch->stats.drops++;
57 #ifdef CONFIG_NET_CLS_POLICE
58     if (sch->reshape_fail==NULL || sch->reshape_fail(skb, sch))
59 #endif
60         kfree_skb(skb);
61     return NET_XMIT_DROP;
62 }

```

Figure 1011: FIFO Queueing Scheduler - enqueue:
linux/net/sched/sch_fifo.c

```
64 static int
65 bfifo_requeue(struct sk_buff *skb, struct Qdisc* sch)
66 {
67     __skb_queue_head(&sch->q, skb);
68     sch->stats.backlog += skb->len;
69     return 0;
70 }
```

Figure 1012: FIFO Queueing Scheduler - requeue:
linux/net/sched/sch_fifo.c

```
72 static struct sk_buff *
73 bfifo_dequeue(struct Qdisc* sch)
74 {
75     struct sk_buff *skb;
76
77     skb = _skb_dequeue(&sch->q);
78     if (skb)
79         sch->stats.backlog -= skb->len;
80     return skb;
81 }
```

Figure 1013: FIFO Queuing Scheduler - dequeue:
linux/net/sched/sch_fifo.c

```
83 static int
84 ffo_drop(struct Qdisc* sch)
85 {
86     struct sk_buff *skb;
87
88     skb = _skb_dequeue_tail(&sch->q);
89     if (skb) {
90         sch->stats.backlog -= skb->len;
91         kfree_skb(skb);
92         return 1;
93     }
94     return 0;
95 }
```

Figure 1014: FIFO Queuing Scheduler - drop: `linux/net/sched/sch_fifo.c`


```
97 static void
98 fifo_reset (struct Qdisc* sch)
99 {
100     skb_queue_purge(&sch->q);
101     sch->stats.backlog = 0;
102 }
```

Figure 1015: FIFO Queueing Scheduler - reset: `linux/net/sched/sch_fifo.c`

```

105 pfifo_enqueue(struct sk_buff *skb, struct Qdisc* sch)
106 {
107     struct fifo_sched_data *q = (struct fifo_sched_data *)sch->data;
108
109     if (sch->q.len <= q->limit) {
110         __skb_queue_tail(&sch->q, skb);
111         sch->stats.bytes += skb->len;
112         sch->stats.packets++;
113         return 0;
114     }
115     sch->stats.drops++;
116 #ifdef CONFIG_NET_CLS_POLICE
117     if (sch->reshape_fail==NULL || sch->reshape_fail(skb, sch))
118 #endif
119         kfree_skb(skb);
120     return NET_XMIT_DROP;
121 }

```

Figure 1016: FIFO Queueing Scheduler - enqueue:
linux/net/sched/sch_fifo.c

```
123 static int
124 pfifo_requeue(struct sk_buff *skb, struct Qdisc* sch)
125 {
126     _skb_queue_head(&sch->q, skb);
127     return 0;
128 }
```

Figure 1017: FIFO Queueing Scheduler - requeue:
linux/net/sched/sch_fifo.c

```
131 static struct sk_buff *
132 pfifo_dequeue(struct Qdisc* sch)
133 {
134     return __skb_dequeue(&sch->q);
135 }
```

Figure 1018: FIFO Queueing Scheduler - dequeue:
linux/net/sched/sch_fifo.c

```

138 {
139     struct fifo_sched_data *q = (void*)sch->data;
140
141     if (opt == NULL) {
142         if (sch->ops == &bfifo_qdisc_ops)
143             q->limit = sch->dev->tx_queue_len*sch->dev->mtu;
144         else
145             q->limit = sch->dev->tx_queue_len;
146     } else {
147         struct tc_fifo_qopt *ctl = RTA_DATA(opt);
148         if (opt->rta_len < RTA_LENGTH(sizeof(*ctl)))
149             return -EINVAL;
150         q->limit = ctl->limit;
151     }
152     return 0;
153 }

```

Figure 1019: FIFO Queueing Scheduler - init: `linux/net/sched/sch_fifo.c`

```

33  /*
34  This code is NOT intended to be used for statistics collection,
35  its purpose is to provide a base for statistical multiplexing
36  for controlled load service.
37  If you need only statistics, run a user level daemon which
38  periodically reads byte counters.
39
40  Unfortunately, rate estimation is not a very easy task.
41  F.e. I did not find a simple way to estimate the current peak rate
42  and even failed to formulate the problem 8)8)
43
44  So I preferred not to built an estimator into the scheduler,
45  but run this task separately.
46  Ideally, it should be kernel thread(s), but for now it runs
47  from timers, which puts apparent top bounds on the number of rated
48  flows, has minimal overhead on small, but is enough
49  to handle controlled load service, sets of aggregates.
50
51  We measure rate over A=(1<<interval) seconds and evaluate EWMA:
52
53  avrate = avrate*(1-W) + rate*W
54
55  where W is chosen as negative power of 2: W = 2-(ewma_log)
56
57  The resulting time constant is:
58
59  T = A/(-ln(1-W))
60
61
62  NOTES.
63
64  * The stored value for avbps is scaled by 25, so that maximal
65  rate is ~1Gbit, avpps is scaled by 210.
66
67  * Minimal interval is HZ/4=250msec (it is the greatest common divisor
68  for HZ=100 and HZ=1024 8)), maximal interval
69  is (HZ/4)*2EST_MAX_INTERVAL = 8sec. Shorter intervals
70  are too expensive, longer ones can be implemented
71  at user level painlessly.
72  */

```

Figure 1020: Rate Estimator - general purpose block comment:
linux/net/sched/estimator.c

```

74 #if (HZ%4) != 0
75 #error Bad HZ value.
76 #endif
77
78 #define EST_MAX_INTERVAL 5
79
80 struct qdisc_estimator
81 {
82     struct qdisc_estimator *next;
83     struct tc_stats        *stats;
84     unsigned               interval;
85     int                    ewma_log;
86     u64                    last_bytes;
87     u32                    last_packets;
88     u32                    avpps;
89     u32                    avbps;
90 };
91
92 struct qdisc_estimator_head
93 {
94     struct timer_list      timer;
95     struct qdisc_estimator *list;
96 };
97
98 static struct qdisc_estimator_head elist [EST_MAX_INTERVAL+1];
99
100 /* Estimator array lock */
101 static rwlock_t est_lock = RW_LOCK_UNLOCKED;

```

Figure 1021: Rate Estimator - Data: `linux/net/sched/estimator.c`

```

103 static void est_timer(unsigned long arg)
104 {
105     int idx = (int)arg;
106     struct qdisc_estimator *e;
107
108     read_lock(&est_lock);
109     for (e = elist[idx].list; e; e = e->next) {
110         struct tc_stats *st = e->stats;
111         u64 nbytes;
112         u32 npackets;
113         u32 rate;
114
115         spin_lock(st->lock);
116         nbytes = st->bytes;
117         npackets = st->packets;
118         rate = (nbytes - e->last_bytes) << (7 - idx);
119         e->last_bytes = nbytes;
120         e->avbps += ((long)rate - (long)e->avbps) >> e->ewma_log;
121         st->bps = (e->avbps + 0xF) >> 5;
122
123         rate = (npackets - e->last_packets) << (12 - idx);
124         e->last_packets = npackets;
125         e->avpps += ((long)rate - (long)e->avpps) >> e->ewma_log;
126         e->stats->pps = (e->avpps + 0x1FF) >> 10;
127         spin_unlock(st->lock);
128     }
129
130     mod_timer(&elist[idx].timer, jiffies + ((HZ/4) << idx));
131     read_unlock(&est_lock);
132 }

```

Figure 1022: Rate Estimator - timer routine for estimation:
linux/net/sched/estimator.c


```

134 int qdisc_new_estimator(struct tc_stats *stats, struct rtattr *opt)
135 {
136     struct qdisc_estimator *est;
137     struct tc_estimator *parm = RTA_DATA(opt);
138
139     if (RTA_PAYLOAD(opt) < sizeof(*parm))
140         return -EINVAL;
141
142     if (parm->interval < -2 || parm->interval > 3)
143         return -EINVAL;
144
145     est = kmalloc(sizeof(*est), GFP_KERNEL);
146     if (est == NULL)
147         return -ENOMEM;
148
149     memset(est, 0, sizeof(*est));
150     est->interval = parm->interval + 2;
151     est->stats = stats;
152     est->ewma_log = parm->ewma_log;
153     est->last_bytes = stats->bytes;
154     est->avbps = stats->bps<<5;
155     est->last_packets = stats->packets;
156     est->avpps = stats->pps<<10;
157
158     est->next = elist[est->interval].list;
159     if (est->next == NULL) {
160         init_timer(&elist[est->interval].timer);
161         elist[est->interval].timer.data = est->interval;
162         elist[est->interval].timer.expires = jiffies + ((HZ/4)<<est->interval);
163         elist[est->interval].timer.function = est_timer;
164         add_timer(&elist[est->interval].timer);
165     }
166     write_lock_bh(&est_lock);
167     elist[est->interval].list = est;
168     write_unlock_bh(&est_lock);
169     return 0;
170 }

```

Figure 1023: Rate Estimator - add a timer routine for estimation:
[linux/net/sched/estimator.c](#)

```

172 void qdisc_kill_estimator (struct tc_stats *stats)
173 {
174     int idx;
175     struct qdisc_estimator *est, **pest;
176
177     for (idx=0; idx <= EST_MAX_INTERVAL; idx++) {
178         int killed = 0;
179         pest = &elist[idx].list;
180         while ((est=*pest) != NULL) {
181             if (est->stats != stats) {
182                 pest = &est->next;
183                 continue;
184             }
185
186             write_lock_bh(&est_lock);
187             *pest = est->next;
188             write_unlock_bh(&est_lock);
189
190             kfree(est);
191             killed++;
192         }
193         if (killed && elist[idx].list == NULL)
194             del_timer(&elist[idx].timer);
195     }
196 }

```

Figure 1024: Rate Estimator - stop estimation timer routine for estimation:
linux/net/sched/estimator.c

```

34 #define L2T(p,L) ((p)->R_tab->data[(L)>>(p)->R_tab->rate.cell_log])
35 #define L2T_P(p,L) ((p)->P_tab->data[(L)>>(p)->P_tab->rate.cell_log])
36
37 static u32 idx_gen;
38 static struct tcf_police * tcf_police_ht [16];
39 /* Policer hash table lock */
40 static rwlock_t police_lock = RW_LOCK_UNLOCKED;
41
42 /* Each policer is serialized by its individual spinlock */
43
44 static __inline__ unsigned tcf_police_hash(u32 index)
45 {
46     return index&0xF;
47 }
48
49 static __inline__ struct tcf_police * tcf_police_lookup(u32 index)
50 {
51     struct tcf_police *p;
52
53     read_lock(&police_lock);
54     for (p = tcf_police_ht [ tcf_police_hash (index)]; p; p = p->next) {
55         if (p->index == index)
56             break;
57     }
58     read_unlock(&police_lock);
59     return p;
60 }
61
62 static __inline__ u32 tcf_police_new_index(void)
63 {
64     do {
65         if (++idx_gen == 0)
66             idx_gen = 1;
67     } while ( tcf_police_lookup (idx_gen));
68
69     return idx_gen;
70 }

```

Figure 1025: Policer i: `linux/net/sched/police.c`

```

73 void tcf_police_destroy (struct tcf_police *p)
74 {
75     unsigned h = tcf_police_hash(p->index);
76     struct tcf_police **p1p;
77
78     for (p1p = &tcf_police_ht[h]; *p1p; p1p = &(*p1p)->next) {
79         if (*p1p == p) {
80             write_lock_bh(&police_lock);
81             *p1p = p->next;
82             write_unlock_bh(&police_lock);
83 #ifdef CONFIG_NET_ESTIMATOR
84             qdisc_kill_estimator (&p->stats);
85 #endif
86             if (p->R_tab)
87                 qdisc_put_rtab(p->R_tab);
88             if (p->P_tab)
89                 qdisc_put_rtab(p->P_tab);
90             kfree(p);
91             return;
92         }
93     }
94     BUG_TRAP(0);
95 }

```

Figure 1026: Policer ii - destroy policer: `linux/net/sched/police.c`

```

97 struct tcf_police * tcf_police_locate (struct rtattr *rta, struct rtattr *est)
98 {
99     unsigned h;
100     struct tcf_police *p;
101     struct rtattr *tb[TCA_POLICE_MAX];
102     struct tc_police *parm;
103
104     if (rtattr_parse(tb, TCA_POLICE_MAX, RTA_DATA(rta), RTA_PAYLOAD(rta))
105         < 0)
106         return NULL;
107
108     if (tb[TCA_POLICE_TBF-1] == NULL)
109         return NULL;
110
111     parm = RTA_DATA(tb[TCA_POLICE_TBF-1]);
112
113     if (parm->index && (p = tcf_police_lookup(parm->index)) != NULL) {
114         p->refcnt++;
115         return p;
116     }
117
118     p = kmalloc(sizeof(*p), GFP_KERNEL);
119     if (p == NULL)
120         return NULL;

```

Figure 1027: Policer iii - locate: `linux/net/sched/police.c`

```

120     memset(p, 0, sizeof(*p));
121     p->refcnt = 1;
122     spin_lock_init (&p->lock);
123     p->stats.lock = &p->lock;
124     if (parm->rate.rate) {
125         if ((p->R_tab = qdisc_get_rtab(&parm->rate, tb[TCA_POLICE_RATE
126             -1])) == NULL)
127             goto failure;
128         if (parm->peakrate.rate &&
129             (p->P_tab = qdisc_get_rtab(&parm->peakrate, tb[
130                 TCA_POLICE_PEAKRATE-1])) == NULL)
131             goto failure;
132     }
133     if (tb[TCA_POLICE_RESULT-1])
134         p->result = *(int*)RTA_DATA(tb[TCA_POLICE_RESULT-1]);
135 #ifdef CONFIG_NET_ESTIMATOR
136     if (tb[TCA_POLICE_AVRATE-1])
137         p->ewma_rate = *(u32*)RTA_DATA(tb[TCA_POLICE_AVRATE-1]);
138 #endif
139     p->toks = p->burst = parm->burst;
140     p->mtu = parm->mtu;
141     if (p->mtu == 0) {
142         p->mtu = ~0;
143         if (p->R_tab)
144             p->mtu = 255 << (p->R_tab->rate.cell_log);
145     }
146     if (p->P_tab)
147         p->ptoks = L2T_P(p, p->mtu);
148     PSCHED_GET_TIME(p->t_c);
149     p->index = parm->index ? : tcf_police_new_index();
150     p->action = parm->action;

```

Figure 1028: Policer iv - locate: `linux/net/sched/police.c`

```

149     p->action = parm->action;
150 #ifdef CONFIG_NET_ESTIMATOR
151     if (est)
152         qdisc_new_estimator(&p->stats, est);
153 #endif
154     h = tcf_police_hash(p->index);
155     write_lock_bh(&police_lock);
156     p->next = tcf_police_ht[h];
157     tcf_police_ht[h] = p;
158     write_unlock_bh(&police_lock);
159     return p;
160
161 failure :
162     if (p->R_tab)
163         qdisc_put_rtab(p->R_tab);
164     kfree(p);
165     return NULL;
166 }

```

Figure 1029: Policer iv - the Police!!!: `linux/net/sched/police.c`

```

168 int tcf_police (struct sk_buff *skb, struct tcf_police *p)
169 {
170     psched_time_t now;
171     long toks;
172     long ptoks = 0;
173
174     spin_lock(&p->lock);
175
176     p->stats.bytes += skb->len;
177     p->stats.packets++;
178
179     #ifdef CONFIG_NET_ESTIMATOR
180     if (p->ewma_rate && p->stats.bps >= p->ewma_rate) {
181         p->stats.overlimits++;
182         spin_unlock(&p->lock);
183         return p->action;
184     }
185     #endif
186
187     if (skb->len <= p->mtu) {
188         if (p->Rtab == NULL) {
189             spin_unlock(&p->lock);
190             return p->result;
191         }
192
193         PSCHED_GET_TIME(now);
194
195         toks = PSCHED_TDIFF_SAFE(now, p->t_c, p->burst, 0);

```

Figure 1030: Policer v - the Police!!!: `linux/net/sched/police.c`


```

197         if (p->P_tab) {
198             ptoks = toks + p->ptoks;
199             if (ptoks > (long)L2T_P(p, p->mtu))
200                 ptoks = (long)L2T_P(p, p->mtu);
201             ptoks -= L2T_P(p, skb->len);
202         }
203         toks += p->toks;
204         if (toks > (long)p->burst)
205             toks = p->burst;
206         toks -= L2T(p, skb->len);
207
208         if ((toks|ptoks) >= 0) {
209             p->t_c = now;
210             p->toks = toks;
211             p->ptoks = ptoks;
212             spin_unlock(&p->lock);
213             return p->result;
214         }
215     }
216
217     p->stats.overlimits++;
218     spin_unlock(&p->lock);
219     return p->action;
220 }

```

Figure 1031: Policer v - the Police!!!: `linux/net/sched/police.c`

```
47 struct ingress_qdisc_data {
48     struct Qdisc      *q;
49     struct tcf_proto  * filter_list ;
50 };
```

Figure 1032: Ingress Queuing Scheduler - Data:
linux/net/sched/sch_ingress.c

```

56 static int ingress_graft (struct Qdisc *sch, unsigned long arg,
57                          struct Qdisc *new, struct Qdisc **old)
58 {
59 #ifdef DEBUG_INGRESS
60     struct ingress_qdisc_data *p = PRIV(sch);
61 #endif
62
63     DPRINTK("ingress_graft(sch_%p,[qdisc_%p],new_%p,old_%p)\n",
64            sch, p, new, old);
65     DPRINTK("\ningress_graft: You cannot add qdiscs to classes");
66     return 1;
67 }
68
69
70 static struct Qdisc * ingress_leaf (struct Qdisc *sch, unsigned long arg)
71 {
72     return NULL;
73 }
74
75
76 static unsigned long ingress_get (struct Qdisc *sch, u32 classid)
77 {
78 #ifdef DEBUG_INGRESS
79     struct ingress_qdisc_data *p = PRIV(sch);
80 #endif
81     DPRINTK("ingress_get(sch_%p,[qdisc_%p],classid_%x)\n", sch, p, classid);
82     return TC_H_MIN(classid) + 1;
83 }
84
85
86 static unsigned long ingress_bind_filter (struct Qdisc *sch,
87                                          unsigned long parent, u32 classid)
88 {
89     return ingress_get (sch, classid);
90 }
91
92
93 static void ingress_put (struct Qdisc *sch, unsigned long cl)
94 {
95 }

```

Figure 1033: Ingress Queueing - Tree handling:
linux/net/sched/sch_ingress.c

```

98 static int ingress_change(struct Qdisc *sch, u32 classid, u32 parent,
99                          struct rtattr **tca, unsigned long *arg)
100 {
101 #ifdef DEBUG_INGRESS
102     struct ingress_qdisc_data *p = PRIV(sch);
103 #endif
104     DPRINTK("ingress_change(sch=%p,[qdisc=%p],classid=%x,parent=%x),"
105            "arg=0x%lx\n", sch, p, classid, parent, *arg);
106     DPRINTK("No effect. sch_ingress_doesnt_maintain_classes_at_the_moment");
107     return 0;
108 }

```

Figure 1034: Ingress Queueing - Change Parameters:
linux/net/sched/sch_ingress.c

```

112 static void ingress_walk(struct Qdisc *sch,struct qdisc_walker *walker)
113 {
114 #ifdef DEBUG_INGRESS
115     struct ingress_qdisc_data *p = PRIV(sch);
116 #endif
117     DPRINTK("ingress_walk(sch=%p,[qdisc=%p],walker=%p)\n", sch, p, walker);
118     DPRINTK("No effect. sch_ingress doesn't maintain classes at the moment");
119 }
120
121
122 static struct tcf_proto ** ingress_find_tcf (struct Qdisc *sch,unsigned long cl)
123 {
124     struct ingress_qdisc_data *p = PRIV(sch);
125
126     return &p->filter_list ;
127 }

```

Figure 1035: Ingress Queueing - Walk Tree: `linux/net/sched/sch_ingress.c`

```

133 static int ingress_enqueue(struct sk_buff *skb, struct Qdisc *sch)
134 {
135     struct ingress_qdisc_data *p = PRIV(sch);
136     struct tcf_result res;
137     int result;
138
139     D2PRINTK("ingress_enqueue(skb_%p, sch_%p, [qdisc_%p])\n", skb, sch, p);
140     result = tc_classify(skb, p->filter_list, &res);
141     D2PRINTK("result_%d, class_0x%04x\n", result, res.classid);
142     /*
143      * Unlike normal "enqueue" functions, ingress_enqueue returns a
144      * firewall FW_* code.
145      */
146     switch (result) {
147 #ifdef CONFIG_NET_CLS_POLICE
148         case TC_POLICE_SHOT:
149             result = NF_DROP;
150             break;
151         case TC_POLICE_RECLASSIFY: /* DSCP remarking here ? */
152         case TC_POLICE_OK:
153         case TC_POLICE_UNSPEC:
154             default:
155                 result = NF_ACCEPT;
156                 break;
157 #endif
158     };
159
160     skb->tc_index = TC_H_MIN(res.classid);
161     return result;
162 }

```

Figure 1036: Ingress Queueing - Enqueue: linux/net/sched/sch_ingress.c

```
165 static struct sk_buff *ingress_dequeue(struct Qdisc *sch)
166 {
167     /*
168         struct ingress_qdisc_data *p = PRIV(sch);
169         D2PRINTK("ingress_dequeue(sch %p,[qdisc %p])\n",sch,PRIV(p));
170     */
171     return NULL;
172 }
```

Figure 1037: Ingress Queueing - Dequeue: `linux/net/sched/sch_ingress.c`

```
175 static int ingress_requeue(struct sk_buff *skb, struct Qdisc *sch)
176 {
177     /*
178         struct ingress_qdisc_data *p = PRIV(sch);
179         D2PRINTK("ingress_requeue(skb %p, sch %p, [qdisc %p])\n", skb, sch, PRIV(p));
180     */
181     return 0;
182 }
```

Figure 1038: Ingress Queueing - Requeue: `linux/net/sched/sch_ingress.c`


```
184 static int ingress_drop(struct Qdisc *sch)
185 {
186 #ifdef DEBUG_INGRESS
187     struct ingress_qdisc_data *p = PRIV(sch);
188 #endif
189     DPRINTK("ingress_drop(sch=%p,[qdisc=%p])\n", sch, p);
190     return 0;
191 }
```

Figure 1039: Ingress Queueing - Drop: `linux/net/sched/sch_ingress.c`

```

30
31
32 #define PRIV(sch) ((struct dsmark_qdisc_data *) (sch)->data)
33
34
35 /*
36 * classid      class      marking
37 * -----      -
38 * n/a          0          n/a
39 * x:0          1          use entry [0]
40 * ...          ...        ...
41 * x:y y>0     y+1        use entry [y]
42 * ...          ...        ...
43 * x:indices-1 indices    use entry [indices-1]
44 * ...          ...        ...
45 * x:y          y+1        use entry [y & (indices-1)]
46 * ...          ...        ...
47 * 0xffff      0x10000    use entry [indices-1]
48 */
49
50
51 #define NO_DEFAULT_INDEX (1 << 16)
52
53 struct dsmark_qdisc_data {

```

Figure 1040: Differentiated Services Field Marker - Data Types:
linux/net/sched/sch_dsmark.c

```

56     _u8          *mask; /* "owns" the array */
57     _u8          *value;
58     _u16         indices;
59     _u32         default_index; /* index range is 0...0xffff */
60     int          set_tc_index;
61 };
62
63
64 /* ----- Class/flow operations ----- */
65
66
67 static int dsmark_graft(struct Qdisc *sch,unsigned long arg,
68     struct Qdisc *new,struct Qdisc **old)
69 {
70     struct dsmark_qdisc_data *p = PRIV(sch);
71
72     DPRINTK("dsmark_graft(sch_%p,[qdisc_%p],new_%p,old_%p)\n",sch,p,new,
73     old);
74     if (!new)
75         new = &noop_qdisc;
76     sch_tree_lock(sch);
77     *old = xchg(&p->q,new);
78     if (*old)
79         qdisc_reset(*old);
80     sch_tree_unlock(sch); /* @@@ move up ? */
81     return 0;
82 }
83
84
85 static struct Qdisc *dsmark_leaf(struct Qdisc *sch, unsigned long arg)
86 {
87     struct dsmark_qdisc_data *p = PRIV(sch);
88
89     return p->q;
90 }
91
92
93 static unsigned long dsmark_get(struct Qdisc *sch,u32 classid)
94 {
95     struct dsmark_qdisc_data *p __attribute__((unused)) = PRIV(sch);
96
97     DPRINTK("dsmark_get(sch_%p,[qdisc_%p],classid_%x)\n",sch,p,classid);
98     return TC_H_MIN(classid)+1;
99 }

```

Figure 1041: Differentiated Services Field Marker - tree functions:
linux/net/sched/sch_dsmark.c

```

103     unsigned long parent, u32 classid)
104 {
105     return dsmark_get(sch,classid);
106 }
107
108
109 static void dsmark_put(struct Qdisc *sch, unsigned long cl)
110 {
111 }
112
113
114 static int dsmark_change(struct Qdisc *sch, u32 classid, u32 parent,
115     struct rtattr **tca, unsigned long *arg)
116 {
117     struct dsmark_qdisc_data *p = PRIV(sch);
118     struct rtattr *opt = tca[TCA_OPTIONS-1];
119     struct rtattr *tb[TCA_DSMARK_MAX];
120
121     DPRINTK("dsmark_change(sch_%p,[qdisc_%p],classid_%x,parent_%x),"
122         "arg_0x%lx\n",sch,p,classid,parent,*arg);
123     if (*arg > p->indices)
124         return -ENOENT;
125     if (!opt || rtattr_parse(tb, TCA_DSMARK_MAX, RTA_DATA(opt),
126         RTA_PAYLOAD(opt)))
127         return -EINVAL;
128     if (tb[TCA_DSMARK_MASK-1]) {
129         if (!RTA_PAYLOAD(tb[TCA_DSMARK_MASK-1]))

```

Figure 1042: Differentiated Services Field Marker - change parameters:
linux/net/sched/sch_dsmark.c

```
132     }
133     if (tb[TCA_DSMARK_VALUE-1]) {
134         if (!RTA_PAYLOAD(tb[TCA_DSMARK_VALUE-1]))
135             return -EINVAL;
136         p->value[*arg-1] = *((_u8 *) RTA_DATA(tb[TCA_DSMARK_VALUE-1]))
137             ;
138     }
139     return 0;
140 }
```

Figure 1043: Marker - delete: `linux/net/sched/sch_dsmark.c`

```

144     struct dsmark_qdisc_data *p = PRIV(sch);
145
146     if (!arg || arg > p->indices)
147         return -EINVAL;
148     p->mask[arg-1] = 0xff;
149     p->value[arg-1] = 0;
150     return 0;
151 }
152
153
154 static void dsmark_walk(struct Qdisc *sch, struct qdisc_walker *walker)
155 {
156     struct dsmark_qdisc_data *p = PRIV(sch);
157     int i;
158
159     DPRINTK("dsmark_walk(sch_%p,[qdisc_%p],walker_%p)\n", sch, p, walker);
160     if (walker->stop)
161         return;
162     for (i = 0; i < p->indices; i++) {
163         if (p->mask[i] == 0xff && !p->value[i])

```

Figure 1044: Marker - walk: `linux/net/sched/sch_dsmark.c`

```
166         if (walker->fn(sch, i+1, walker) < 0) {
167             walker->stop = 1;
168             break;
169         }
170     }
171     walker->count++;
```

Figure 1045: Marker - find: `linux/net/sched/sch_dsmark.c`

```

177 {
178     struct dsmark_qdisc_data *p = PRIV(sch);
179
180     return &p->filter_list;
181 }
182
183
184 /* ----- Qdisc operations ----- */
185
186
187 static int dsmark_enqueue(struct sk_buff *skb, struct Qdisc *sch)
188 {
189     struct dsmark_qdisc_data *p = PRIV(sch);
190     struct tcf_result res;
191     int result;
192     int ret = NET_XMIT_POLICED;
193
194     D2PRINTK("dsmark_enqueue(skb_%p,sch_%p,[qdisc_%p])\n",skb,sch,p);
195     if (p->set_tc_index) {
196         switch (skb->protocol) {
197             case __constant_htons(ETH_P_IP):

```

Figure 1046: Marker - enqueue i: `linux/net/sched/sch_dsmark.c`


```

198         skb->tc_index = ipv4_get_dsfield(skb->nh.iph);
199         break;
200     case _constant_htons(ETH_P_IPV6):
201         skb->tc_index = ipv6_get_dsfield(skb->nh.ipv6h);
202         break;
203     default :
204         skb->tc_index = 0;
205         break;
206     };
207 }
208 result = TC_POLICE_OK; /* be nice to gcc */
209 if (TC_H_MAJ(skb->priority) == sch->handle) {
210     skb->tc_index = TC_H_MIN(skb->priority);
211 } else {
212     result = tc_classify(skb, p->filter_list, &res);
213     D2PRINTK("result_%d_class_0x%04x\n", result, res.classid);
214     switch (result) {
215 #ifdef CONFIG_NET_CLS_POLICE
216         case TC_POLICE_SHOT:
217             kfree_skb(skb);
218             break;
219 #if 0
220         case TC_POLICE_RECLASSIFY:
221             /* FIXME: what to do here ??? */
222 #endif
223 #endif
224         case TC_POLICE_OK:
225             skb->tc_index = TC_H_MIN(res.classid);
226             break;
227         case TC_POLICE_UNSPEC:
228             /* fall through */
229         default :
230             if (p->default_index != NO_DEFAULT_INDEX)
231                 skb->tc_index = p->default_index;
232             break;
233     };
234 }
235 if (
236 #ifdef CONFIG_NET_CLS_POLICE
237     result == TC_POLICE_SHOT ||
238 #endif

```

Figure 1047: Marker - enqueue ii: linux/net/sched/sch_dsmark.c

```

240         ((ret = p->q->enqueue(skb,p->q) != 0)) {
241             sch->stats.drops++;
242             return ret;
243         }
244     sch->stats.bytes += skb->len;
245     sch->stats.packets++;
246     sch->q.qlen++;
247     return ret;
248 }
249
250
251 static struct sk_buff *dsmark_dequeue(struct Qdisc *sch)
252 {
253     struct dsmark_qdisc_data *p = PRIV(sch);
254     struct sk_buff *skb;
255     int index;
256
257     D2PRINTK(" dsmark_dequeue(sch_%p,[qdisc_%p])\n",sch,p);
258     skb = p->q->ops->dequeue(p->q);
259     if (!skb)
260         return NULL;
261     sch->q.qlen--;
262     index = skb->tc_index & (p->indices-1);
263     D2PRINTK(" index_%d->%d\n",skb->tc_index,index);
264     switch (skb->protocol) {
265         case __constant_htons(ETH_P_IP):
266             ipv4_change_dsfield (skb->nh.iph,
267                                 p->mask[index],p->value[index]);
268             break;
269         case __constant_htons(ETH_P_IPV6):
270             ipv6_change_dsfield (skb->nh.ipv6h,
271                                 p->mask[index],p->value[index]);
272             break;
273         default:
274             /*
275              * Only complain if a change was actually attempted.
276              * This way, we can send non-IP traffic through dsmark

```

Figure 1048: Marker - dequeue: linux/net/sched/sch_dsmark.c

```

279         if (p->mask[index] != 0xff || p->value[index])
280             printk(KERN_WARNING "dsmark_dequeue: "
281                    "unsupported_protocol_%d\n",
282                    htons(skb->protocol));
283         break;
284     };
285     return skb;
286 }
287
288 static int dsmark_requeue(struct sk_buff *skb, struct Qdisc *sch)
289 {
290     int ret;
291

```

Figure 1049: Marker - requeue: `linux/net/sched/sch_dsmark.c`

```

294     D2PRINTK(" dsmark_requeue(skb,%p,sch,%p,[qdisc,%p])\n",skb,sch,p);
295     if ((ret = p->q->ops->requeue(skb, p->q)) == 0) {
296         sch->q.qlen++;
297         return 0;
298     }
299     sch->stats.drops++;
300     return ret;
301 }
302
303
304 static int dsmark_drop(struct Qdisc *sch)
305 {

```

Figure 1050: Marker - drop: `linux/net/sched/sch_dsmark.c`

```

308     DPRINTF("dsmark_reset(sch_%p,[qdisc_%p])\n",sch,p);
309     if (!p->q->ops->drop)
310         return 0;
311     if (!p->q->ops->drop(p->q))
312         return 0;
313     sch->q.qlen--;
314     return 1;
315 }
316
317
318 int dsmark_init(struct Qdisc *sch,struct rtattr *opt)
319 {
320     struct dsmark_qdisc_data *p = PRIV(sch);
321     struct rtattr *tb[TCA_DSMARK_MAX];
322     __u16 tmp;
323
324     DPRINTF("dsmark_init(sch_%p,[qdisc_%p],opt_%p)\n",sch,p,opt);
325     if (rtattr_parse(tb,TCA_DSMARK_MAX,RTA_DATA(opt),RTA_PAYLOAD(opt))
326         < 0 ||
327         !tb[TCA_DSMARK_INDICES-1] ||
328         RTA_PAYLOAD(tb[TCA_DSMARK_INDICES-1]) < sizeof(__u16))
329         return -EINVAL;
330     memset(p,0,sizeof(*p));
331     p->filter_list = NULL;
332     p->indices = *(__u16 *) RTA_DATA(tb[TCA_DSMARK_INDICES-1]);
333     if (!p->indices)
334         return -EINVAL;
335     for (tmp = p->indices; tmp != 1; tmp >>= 1) {
336         if (tmp & 1)
337             return -EINVAL;
338     }
339     p->default_index = NO_DEFAULT_INDEX;
340     if (tb[TCA_DSMARK_DEFAULT_INDEX-1]) {
341         if (RTA_PAYLOAD(tb[TCA_DSMARK_DEFAULT_INDEX-1]) < sizeof(
342             __u16))
343             return -EINVAL;
344         p->default_index =
345             *(__u16 *) RTA_DATA(tb[TCA_DSMARK_DEFAULT_INDEX-1]);
346     }
347     p->set_tc_index = !tb[TCA_DSMARK_SET_TC_INDEX-1];
348     p->mask = kmalloc(p->indices*2,GFP_KERNEL);
349     if (!p->mask)
350         return -ENOMEM;
351     p->value = p->mask+p->indices;

```

Figure 1051: Marker - init: linux/net/sched/sch_dsmark.c

```
352     if (!(p->q = qdisc_create_dflt(sch->dev, &pfifo_qdisc_ops))
353         p->q = &noop_qdisc;
354     DPRINTK("dsmark_init: %p\n", &p->q);
355     MOD_INC_USE_COUNT;
356     return 0;
357 }
```

Figure 1052: Marker - reset: `linux/net/sched/sch_dsmark.c`

```

362     struct dsmark_qdisc_data *p = PRIV(sch);
363
364     DPRINTK("dsmark_reset(sch_%p,[qdisc_%p])\n",sch,p);
365     qdisc_reset (p->q);
366     sch->q qlen = 0;
367 }
368
369
370 static void dsmark_destroy(struct Qdisc *sch)
371 {
372     struct dsmark_qdisc_data *p = PRIV(sch);
373     struct tcf_proto *tp;
374
375     DPRINTK("dsmark_destroy(sch_%p,[qdisc_%p])\n",sch,p);
376     while (p->filter_list ) {
377         tp = p->filter_list ;

```

Figure 1053: Marker - destroy: `linux/net/sched/sch_dsmark.c`

```

382     p->q = &noop_qdisc;
383     kfree(p->mask);
384     MOD_DEC_USE_COUNT;
385 }
386
387
388 #ifdef CONFIG_RTNETLINK
389
390 static int dsmark_dump_class(struct Qdisc *sch, unsigned long cl,
391     struct sk_buff *skb, struct tcmg *tcm)
392 {
393     struct dsmark_qdisc_data *p = PRIV(sch);
394     unsigned char *b = skb->tail;
395     struct rtattr *rta;
396
397     DPRINTK("dsmark_dump_class(sch=%p,[qdisc=%p],class=%d\n",sch,p,cl);
398     if (!cl || cl > p->indices)
399         return -EINVAL;
400     tcm->tcm_handle = TC_H_MAKE(TC_H_MAJ(sch->handle),cl-1);
401     rta = (struct rtattr *) b;
402     RTA_PUT(skb,TCA_OPTIONS,0,NULL);
403     RTA_PUT(skb,TCA_DSMARK_MASK,1,&p->mask[cl-1]);

```

Figure 1054: Marker - dump class: `linux/net/sched/sch_dsmark.c`


```

405     rta->rta_len = skb->tail-b;
406     return skb->len;
407
408 rtattr_failure :
409     skb_trim(skb,b-skb->data);
410     return -1;
411 }
412
413 static int dsmark_dump(struct Qdisc *sch, struct sk_buff *skb)
414 {
415     struct dsmark_qdisc_data *p = PRIV(sch);
416     unsigned char *b = skb->tail;
417     struct rtattr *rta;
418
419     rta = (struct rtattr *) b;
420     RTA_PUT(skb,TCA_OPTIONS,0,NULL);
421     RTA_PUT(skb,TCA_DSMARK_INDICES,sizeof(_u16),&p->indices);
422     if (p->default_index != NO_DEFAULT_INDEX) {
423         _u16 tmp = p->default_index;
424
425         RTA_PUT(skb,TCA_DSMARK_DEFAULT_INDEX, sizeof(_u16), &tmp);

```

Figure 1055: Marker - dump: `linux/net/sched/sch_dsmark.c`

```

430     return skb->len;
431
432 rtattr_failure :
433     skb_trim(skb,b-skb->data);
434     return -1;
435 }
436
437 #endif
438
439
440 static struct Qdisc_class_ops dsmark_class_ops =
441 {
442     dsmark_graft,           /* graft */
443     dsmark_leaf,          /* leaf */
444     dsmark_get,           /* get */
445     dsmark_put,           /* put */
446     dsmark_change,        /* change */
447     dsmark_delete,        /* delete */
448     dsmark_walk,          /* walk */
449
450     dsmark_find_tcf,       /* tcf_chain */
451     dsmark_bind_filter,    /* bind_tcf */
452     dsmark_put,           /* unbind_tcf */
453
454 #ifdef CONFIG_RTNETLINK
455     dsmark_dump_class,     /* dump */
456 #endif
457 };
458
459 struct Qdisc_ops dsmark_qdisc_ops =
460 {
461     NULL,                  /* next */
462     &dsmark_class_ops,     /* cl_ops */
463     "dsmark",
464     sizeof(struct dsmark_qdisc_data),
465
466     dsmark_enqueue,        /* enqueue */
467     dsmark_dequeue,        /* dequeue */
468     dsmark_requeue,        /* requeue */
469     dsmark_drop,           /* drop */
470
471     dsmark_init,           /* init */
472     dsmark_reset,          /* reset */
473     dsmark_destroy,        /* destroy */
474     NULL,                  /* change */
475
476 #ifdef CONFIG_RTNETLINK
477     dsmark_dump            /* dump */
478 #endif
479 };
480
481 #ifdef MODULE

```

Figure 1056: Marker - operations registration:
linux/net/sched/sch_dsmark.c

```

44 #define RED_ECN_ECT 0x02
45 #define RED_ECN_CE 0x01
46
47
48 /*      Random Early Detection (RED) algorithm.
49      =====
50
51      Source: Sally Floyd and Van Jacobson, "Random Early Detection Gateways
52      for Congestion Avoidance", 1993, IEEE/ACM Transactions on Networking.
53
54      This file codes a "divisionless" version of RED algorithm
55      as written down in Fig.17 of the paper.
56
57      Short description.
58      -----
59
60      When a new packet arrives we calculate the average queue length:
61
62      avg = (1-W)*avg + W*current_queue_len,
63
64      W is the filter time constant (chosen as 2(-Wlog)), it controls
65      the inertia of the algorithm. To allow larger bursts, W should be
66      decreased.

```

Figure 1057: Random Early Detection (RED) Queue - Introductory Remarks:
linux/net/sched/sch_red.c

```

68     if (avg > th_max) -> packet marked (dropped).
69     if (avg < th_min) -> packet passes.
70     if (th_min < avg < th_max) we calculate probability:
71
72     Pb = max_P * (avg - th_min)/(th_max-th_min)
73
74     and mark (drop) packet with this probability.
75     Pb changes from 0 (at avg==th_min) to max_P (avg==th_max).
76     max_P should be small (not 1), usually 0.01..0.02 is good value.
77
78     max_P is chosen as a number, so that max_P/(th_max-th_min)
79     is a negative power of two in order arithmetics to contain
80     only shifts.
81
82
83     Parameters, settable by user:
84     -----
85
86     limit          - bytes (must be > qth_max + burst)
87
88     Hard limit on queue length, should be chosen >qth_max
89     to allow packet bursts. This parameter does not
90     affect the algorithms behaviour and can be chosen
91     arbitrarily high (well, less than ram size)
92     Really, this limit will never be reached
93     if RED works correctly.

```

Figure 1058: RED Queue - Behaviour: linux/net/sched/sch_red.c

```

95     qth_min      - bytes (should be < qth_max/2)
96     qth_max     - bytes (should be at least 2*qth_min and less limit)
97     Wlog        - bits (<32) log(1/W).
98     Plog        - bits (<32)
99
100    Plog is related to max_P by formula:
101
102    max_P = (qth_max-qth_min)/2^Plog;
103
104    F.e. if qth_max=128K and qth_min=32K, then Plog=22
105    corresponds to max_P=0.02
106
107    Scell_log
108    Stab
109
110    Lookup table for log((1-W)^(t/t_ave)).

```

Figure 1059: RED Queue - Parameters: `linux/net/sched/sch_red.c`

```
113 NOTES:
114
115 Upper bound on W.
116 -----
117
118     If you want to allow bursts of L packets of size S,
119     you should choose W:
120
121      $L + 1 - th\_min/S < (1-(1-W)^L)/W$ 
122
123      $th\_min/S = 32$        $th\_min/S = 4$ 
124
125     log(W) L
126     -1      33
127     -2      35
128     -3      39
129     -4      46
130     -5      57
131     -6      75
132     -7     101
133     -8     135
134     -9     190
135     etc.
136 */
```

Figure 1060: RED Queue - Notes: linux/net/sched/sch_red.c

```

138 struct red_sched_data
139 {
140     /* Parameters */
141     u32      limit;          /* HARD maximal queue length */
142     u32      qth_min;       /* Min average length threshold: A scaled */
143     u32      qth_max;       /* Max average length threshold: A scaled */
144     u32      Rmask;
145     u32      Scell_max;
146     unsigned char flags;
147     char     Wlog;          /* log(W) */
148     char     Plog;          /* random number bits */
149     char     Scell_log;
150     u8       Stab[256];
151
152     /* Variables */
153     unsigned long qave;      /* Average queue length: A scaled */
154     int          qcount;     /* Packets since last random number
155         generation */
156     u32          qR;         /* Cached random number */
157     psched_time_t qidlestart; /* Start of idle period */
158     struct tc_red_xstats st;
159 };

```

Figure 1061: RED - Scheduler Data: linux/net/sched/sch_red.c

```

161 static int red_ecn_mark(struct sk_buff *skb)
162 {
163     if (skb->nh.raw + 20 > skb->tail)
164         return 0;
165
166     switch (skb->protocol) {
167     case __constant_htons(ETH_P_IP):
168     {
169         u8 tos = skb->nh.iph->tos;
170
171         if (!(tos & RED_ECN_ECT))
172             return 0;
173
174         if (!(tos & RED_ECN_CE))
175             IP_ECN_set_ce(skb->nh.iph);
176
177         return 1;
178     }
179
180     case __constant_htons(ETH_P_IPV6):
181     {
182         u32 label = *(u32*)skb->nh.raw;
183
184         if (!(label & __constant_htonl(RED_ECN_ECT<<20)))
185             return 0;
186         label |= __constant_htonl(RED_ECN_CE<<20);
187         return 1;
188     }
189
190     default :
191         return 0;
192     }
193 }

```

Figure 1062: RED - ECN bit handling: `linux/net/sched/sch_red.c`


```

195 static int
196 red_enqueue(struct sk_buff *skb, struct Qdisc* sch)
197 {
198     struct red_sched_data *q = (struct red_sched_data *)sch->data;
199
200     psched_time_t now;
201
202     if (!PSCHED_IS_PASTPERFECT(q->qidlestart)) {
203         long us_idle;
204         int shift;
205
206         PSCHED_GET_TIME(now);
207         us_idle = PSCHED_TDIFF_SAFE(now, q->qidlestart, q->Scell_max, 0);
208         PSCHED_SET_PASTPERFECT(q->qidlestart);
209
210     /*
211     The problem: ideally, average length queue recalculation should
212     be done over constant clock intervals. This is too expensive, so that
213     the calculation is driven by outgoing packets.
214     When the queue is idle we have to model this clock by hand.
215
216     SF+VJ proposed to "generate" m = idletime/(average_pkt_size/bandwidth)
217     dummy packets as a burst after idle time, i.e.
218
219         q->qave *= (1-W)^m
220
221     This is an apparently overcomplicated solution (f.e. we have to precompute
222     a table to make this calculation in reasonable time)
223     I believe that a simpler model may be used here,
224     but it is field for experiments.
225     */

```

Figure 1063: RED - Enqueue i: linux/net/sched/sch_red.c

```

226     shift = q->Stab[us_idle]>>q->Scell_log;
227
228     if ( shift ) {
229         q->qave >>= shift;
230     } else {
231         /* Approximate initial part of exponent
232            with linear function:
233            (1-W)^m ~ 1-mW + ...
234
235            Seems, it is the best solution to
236            problem of too coarse exponent tabulation.
237            */
238
239         us_idle = (q->qave * us_idle)>>q->Scell_log;
240         if ( us_idle < q->qave/2)
241             q->qave -= us_idle;
242         else
243             q->qave >>= 1;
244     }
245 } else {
246     q->qave += sch->stats.backlog - (q->qave >> q->Wlog);
247     /* NOTE:
248        q->qave is fixed point number with point at Wlog.
249        The formulae above is equivalent to floating point
250        version:
251
252        qave = qave*(1-W) + sch->stats.backlog*W;
253                                     --ANK (980924)
254     */
255 }

```

Figure 1064: RED - Enqueue ii: linux/net/sched/sch_red.c

```

257     if (q->qave < q->qth_min) {
258         q->qcount = -1;
259 enqueue:
260         if (sch->stats.backlog <= q->limit) {
261             __skb_queue_tail(&sch->q, skb);
262             sch->stats.backlog += skb->len;
263             sch->stats.bytes += skb->len;
264             sch->stats.packets++;
265             return NET_XMIT_SUCCESS;
266         } else {
267             q->st.pdrop++;
268         }
269         kfree_skb(skb);
270         sch->stats.drops++;
271         return NET_XMIT_DROP;
272     }
273     if (q->qave >= q->qth_max) {
274         q->qcount = -1;
275         sch->stats.overlimits++;
276 mark:
277         if (!(q->flags & TC_RED_ECN) || !red_ecn_mark(skb)) {
278             q->st.early++;
279             goto drop;
280         }
281         q->st.marked++;
282         goto enqueue;
283     }

```

Figure 1065: RED - Enqueue iii: `linux/net/sched/sch_red.c`

```

285     if (++q->qcount) {
286         /* The formula used below causes questions.
287
288         OK. qR is random number in the interval 0..Rmask
289         i.e. 0..(2^Plog). If we used floating point
290         arithmetics, it would be: (2^Plog)*rnd_num,
291         where rnd_num is less 1.
292
293         Taking into account, that qave have fixed
294         point at Wlog, and Plog is related to max_P by
295         max_P = (qth_max-qth_min)/2^Plog; two lines
296         below have the following floating point equivalent:
297
298         max_P*(qave - qth_min)/(qth_max-qth_min) < rnd/qcount
299
300         Any questions? --ANK (980924)
301         */
302         if (((q->qave - q->qth_min)>>q->Wlog)*q->qcount < q->qR)
303             goto enqueue;
304         q->qcount = 0;
305         q->qR = net_random()&q->Rmask;
306         sch->stats.overlimits++;
307         goto mark;
308     }
309     q->qR = net_random()&q->Rmask;
310     goto enqueue;
311
312 drop:
313     kfree_skb(skb);
314     sch->stats.drops++;
315     return NET_XMIT_CN;
316 }

```

Figure 1066: RED - Enqueue i: linux/net/sched/sch_red.c

```

318 static int
319 red_requeue(struct sk_buff *skb, struct Qdisc* sch)
320 {
321     struct red_sched_data *q = (struct red_sched_data *)sch->data;
322
323     PSCHED_SET_PASTPERFECT(q->qidlestart);
324
325     __skb_queue_head(&sch->q, skb);
326     sch->stats.backlog += skb->len;
327     return 0;
328 }
329
330 static struct sk_buff *
331 red_dequeue(struct Qdisc* sch)
332 {
333     struct sk_buff *skb;
334     struct red_sched_data *q = (struct red_sched_data *)sch->data;
335
336     skb = __skb_dequeue(&sch->q);
337     if (skb) {
338         sch->stats.backlog -= skb->len;
339         return skb;
340     }
341     PSCHED_GET_TIME(q->qidlestart);
342     return NULL;
343 }

```

Figure 1067: RED - Requeue and Dequeue: `linux/net/sched/sch_red.c`

```

345 static int
346 red_drop(struct Qdisc* sch)
347 {
348     struct sk_buff *skb;
349     struct red_sched_data *q = (struct red_sched_data *)sch->data;
350
351     skb = __skb_dequeue_tail(&sch->q);
352     if (skb) {
353         sch->stats.backlog -= skb->len;
354         sch->stats.drops++;
355         q->st.other++;
356         kfree_skb(skb);
357         return 1;
358     }
359     PSCHED_GET_TIME(q->qidlestart);
360     return 0;
361 }

```

Figure 1068: RED - Drop: `linux/net/sched/sch_red.c`

```
363 static void red_reset(struct Qdisc* sch)
364 {
365     struct red_sched_data *q = (struct red_sched_data *)sch->data;
366     struct sk_buff *skb;
367
368     while((skb=_skb_dequeue(&sch->q))!=NULL)
369         kfree_skb(skb);
370     sch->stats.backlog = 0;
371     PSCHED_SET_PASTPERFECT(q->qidlestart);
372     q->qave = 0;
373     q->qcount = -1;
374 }
```

Figure 1069: RED - Reset: `linux/net/sched/sch_red.c`

```

376 static int red_change(struct Qdisc *sch, struct rtattr *opt)
377 {
378     struct red_sched_data *q = (struct red_sched_data *)sch->data;
379     struct rtattr *tb[TCA_RED_STAB];
380     struct tc_red_qopt *ctl;
381
382     if (opt == NULL ||
383         rtattr_parse(tb, TCA_RED_STAB, RTA_DATA(opt), RTA_PAYLOAD(opt)) ||
384         tb[TCA_RED_PARMS-1] == 0 || tb[TCA_RED_STAB-1] == 0 ||
385         RTA_PAYLOAD(tb[TCA_RED_PARMS-1]) < sizeof(*ctl) ||
386         RTA_PAYLOAD(tb[TCA_RED_STAB-1]) < 256)
387         return -EINVAL;
388
389     ctl = RTA_DATA(tb[TCA_RED_PARMS-1]);
390
391     sch_tree_lock(sch);
392     q->flags = ctl->flags;
393     q->Wlog = ctl->Wlog;
394     q->Plog = ctl->Plog;
395     q->Rmask = ctl->Plog < 32 ? ((1<<ctl->Plog) - 1) : ~0UL;
396     q->Scell_log = ctl->Scell_log;
397     q->Scell_max = (255<<ctl->Scell_log);
398     q->qth_min = ctl->qth_min<<ctl->Wlog;
399     q->qth_max = ctl->qth_max<<ctl->Wlog;
400     q->limit = ctl->limit;
401     memcpy(q->Stab, RTA_DATA(tb[TCA_RED_STAB-1]), 256);
402
403     q->qcount = -1;
404     if (skb_queue_len(&sch->q) == 0)
405         PSCHED_SET_PASTPERFECT(q->qidlestart);
406     sch_tree_unlock(sch);
407     return 0;
408 }

```

Figure 1070: RED - Change: `linux/net/sched/sch_red.c`


```
410 static int red_init(struct Qdisc* sch, struct rtattr *opt)
411 {
412     int err;
413
414     MOD_INC_USE_COUNT;
415
416     if ((err = red_change(sch, opt)) != 0) {
417         MOD_DEC_USE_COUNT;
418     }
419     return err;
420 }
```

Figure 1071: RED - Init: `linux/net/sched/sch_red.c`

```

40  /*      Simple Token Bucket Filter.
41  =====
42
43  SOURCE.
44  -----
45
46  None.
47
48  Description.
49  -----
50
51  A data flow obeys TBF with rate R and depth B, if for any
52  time interval  $t_i \dots t_f$  the number of transmitted bits
53  does not exceed  $B + R*(t_f - t_i)$ .
54
55  Packetized version of this definition:
56  The sequence of packets of sizes  $s_i$  served at moments  $t_i$ 
57  obeys TBF, if for any  $i \leq k$ :
58
59   $s_i + \dots + s_k \leq B + R*(t_k - t_i)$ 
60
61  Algorithm.
62  -----
63
64  Let  $N(t_i)$  be  $B/R$  initially and  $N(t)$  grow continuously with time as:
65
66   $N(t + \delta) = \min\{B/R, N(t) + \delta\}$ 
67
68  If the first packet in queue has length  $S$ , it may be
69  transmitted only at the time  $t_*$  when  $S/R \leq N(t_*)$ ,
70  and in this case  $N(t)$  jumps:
71
72   $N(t_* + 0) = N(t_* - 0) - S/R$ .

```

Figure 1072: Token Bucket Filter Queue - Initial Remarks:
linux/net/sched/sch_tbf.c

```

75
76     Actually, QoS requires two TBF to be applied to a data stream.
77     One of them controls steady state burst size , another
78     one with rate P (peak rate) and depth M (equal to link MTU)
79     limits bursts at a smaller time scale.
80
81     It is easy to see that  $P > R$ , and  $B > M$ . If P is infinity, this double
82     TBF is equivalent to a single one.
83
84     When TBF works in reshaping mode, latency is estimated as:
85
86      $lat = \max ((L-B)/R, (L-M)/P)$ 
87
88
89     NOTES.
90     -----
91
92     If TBF throttles, it starts a watchdog timer, which will wake it up
93     when it is ready to transmit.
94     Note that the minimal timer resolution is  $1/HZ$ .
95     If no new packets arrive during this period,
96     or if the device is not awoken by EOI for some previous packet,
97     TBF can stop its activity for  $1/HZ$ .
98
99
100    This means, that with depth B, the maximal rate is
101
102     $R_{crit} = B * HZ$ 
103
104    F.e. for 10Mbit ethernet and  $HZ=100$  the minimal allowed B is ~10Kbytes.
105
106    Note that the peak rate TBF is much more tough: with MTU 1500
107     $P_{crit} = 150Kbytes/sec$ . So, if you need greater peak
108    rates , use alpha with  $HZ=1000$  :-)
109 */

```

Figure 1073: Token Bucket Filter Queue - Algorithm/Notes:
linux/net/sched/sch_tbf.c

```

111 struct tbf_sched_data
112 {
113     /* Parameters */
114     u32      limit;          /* Maximal length of backlog: bytes */
115     u32      buffer;        /* Token bucket depth/rate: MUST BE >= MTU/B
116                             */
117     u32      mtu;
118     u32      max_size;
119     struct qdisc_rate_table *R_tab;
120     struct qdisc_rate_table *P_tab;
121
122     /* Variables */
123     long     tokens;        /* Current number of B tokens */
124     long     ptokens;       /* Current number of P tokens */
125     psched_time_t t_c;      /* Time check-point */
126     struct timer_list wd_timer; /* Watchdog timer */
127 };
128 #define L2T(q,L) ((q)->R_tab->data[(L)>>(q)->R_tab->rate.cell_log])
129 #define L2T_P(q,L) ((q)->P_tab->data[(L)>>(q)->P_tab->rate.cell_log])

```

Figure 1074: Token Bucket Filter Queue - Data: linux/net/sched/sch_tbf.c

```

131 static int
132 tbf_enqueue(struct sk_buff *skb, struct Qdisc* sch)
133 {
134     struct tbf_sched_data *q = (struct tbf_sched_data *)sch->data;
135
136     if (skb->len > q->max_size)
137         goto drop;
138     __skb_queue_tail(&sch->q, skb);
139     if ((sch->stats.backlog += skb->len) <= q->limit) {
140         sch->stats.bytes += skb->len;
141         sch->stats.packets++;
142         return 0;
143     }
144
145     /* Drop action: undo the things that we just did,
146      * i.e. make tail drop
147      */
148
149     __skb_unlink(skb, &sch->q);
150     sch->stats.backlog -= skb->len;
151
152 drop:
153     sch->stats.drops++;
154     #ifdef CONFIG_NET_CLS_POLICE
155     if (sch->reshape_fail==NULL || sch->reshape_fail(skb, sch))
156     #endif
157         kfree_skb(skb);
158     return NET_XMIT_DROP;
159 }

```

Figure 1075: TBF - Enqueue: linux/net/sched/sch_tbf.c

```

161 static int
162 tbf_requeue(struct sk_buff *skb, struct Qdisc* sch)
163 {
164     __skb_queue_head(&sch->q, skb);
165     sch->stats.backlog += skb->len;
166     return 0;
167 }
168
169 static int
170 tbf_drop(struct Qdisc* sch)
171 {
172     struct sk_buff *skb;
173
174     skb = __skb_dequeue_tail(&sch->q);
175     if (skb) {
176         sch->stats.backlog -= skb->len;
177         sch->stats.drops++;
178         kfree_skb(skb);
179         return 1;
180     }
181     return 0;
182 }

```

Figure 1076: TBF - Requeue and Drop: `linux/net/sched/sch_tbf.c`

```
184 static void tbf_watchdog(unsigned long arg)
185 {
186     struct Qdisc *sch = (struct Qdisc*)arg;
187
188     sch->flags &= ~TCQ_F_THROTTLED;
189     netif_schedule(sch->dev);
190 }
```

Figure 1077: TBF - Watchdog: `linux/net/sched/sch_tbf.c`

```

192 static struct sk_buff *
193 tbf_dequeue(struct Qdisc* sch)
194 {
195     struct tbf_sched_data *q = (struct tbf_sched_data *)sch->data;
196     struct sk_buff *skb;
197
198     skb = __skb_dequeue(&sch->q);
199
200     if (skb) {
201         psched_time_t now;
202         long toks;
203         long ptoks = 0;
204
205         PSCHED_GET_TIME(now);
206
207         toks = PSCHED_TDIFF_SAFE(now, q->t_c, q->buffer, 0);
208
209         if (q->P_tab) {
210             ptoks = toks + q->ptokens;
211             if (ptoks > (long)q->mtu)
212                 ptoks = q->mtu;
213             ptoks -= L2T_P(q, skb->len);
214         }
215         toks += q->tokens;
216         if (toks > (long)q->buffer)
217             toks = q->buffer;
218         toks -= L2T(q, skb->len);

```

Figure 1078: TBF - Dequeue: `linux/net/sched/sch_tbf.c`


```

220     if ((toks|ptoks) >= 0) {
221         q->t_c = now;
222         q->tokens = toks;
223         q->ptokens = ptoks;
224         sch->stats.backlog -= skb->len;
225         sch->flags &= ~TCQ_F_THROTTLED;
226         return skb;
227     }
228
229     if (!netif_queue_stopped(sch->dev)) {
230         long delay = PSCHED_US2JIFFIE(max(-toks, -ptoks));
231
232         if (delay == 0)
233             delay = 1;
234
235         mod_timer(&q->wd_timer, jiffies+delay);
236     }
237
238     /* Maybe we have a shorter packet in the queue,
239     which can be sent now. It sounds cool,
240     but, however, this is wrong in principle.
241     We MUST NOT reorder packets under these circumstances.
242
243     Really, if we split the flow into independent
244     subflows, it would be a very good solution.
245     This is the main idea of all FQ algorithms
246     (cf. CSZ, HPPQ, HFSC)
247     */
248     _skb_queue_head(&sch->q, skb);
249
250     sch->flags |= TCQ_F_THROTTLED;
251     sch->stats.overlimits++;
252 }
253 return NULL;
254 }

```

Figure 1079: TBF - Dequeue: linux/net/sched/sch_tbf.c

```
257 static void
258 tbf_reset (struct Qdisc* sch)
259 {
260     struct tbf_sched_data *q = (struct tbf_sched_data *)sch->data;
261
262     skb_queue_purge(&sch->q);
263     sch->stats.backlog = 0;
264     PSCHED_GET_TIME(q->t_c);
265     q->tokens = q->buffer;
266     q->ptokens = q->mtu;
267     sch->flags &= ~TCQ_F_THROTTLED;
268     del_timer(&q->wd_timer);
269 }
```

Figure 1080: TBF - Reset: `linux/net/sched/sch_tbf.c`

```

271 static int tbf_change(struct Qdisc* sch, struct rtattr *opt)
272 {
273     int err = -EINVAL;
274     struct tbf_sched_data *q = (struct tbf_sched_data *)sch->data;
275     struct rtattr *tb[TCA_TBF_PTAB];
276     struct tc_tbf_qopt *qopt;
277     struct qdisc_rate_table *rtab = NULL;
278     struct qdisc_rate_table *ptab = NULL;
279     int max_size, n;
280
281     if (rtattr_parse(tb, TCA_TBF_PTAB, RTA_DATA(opt), RTA_PAYLOAD(opt)) ||
282         tb[TCA_TBF_PARAMS-1] == NULL ||
283         RTA_PAYLOAD(tb[TCA_TBF_PARAMS-1]) < sizeof(*qopt))
284         goto done;
285
286     qopt = RTA_DATA(tb[TCA_TBF_PARAMS-1]);
287     rtab = qdisc_get_rtab(&qopt->rate, tb[TCA_TBF_RTAB-1]);
288     if (rtab == NULL)
289         goto done;
290
291     if (qopt->peakrate.rate) {
292         if (qopt->peakrate.rate > qopt->rate.rate)
293             ptab = qdisc_get_rtab(&qopt->peakrate, tb[TCA_TBF_PTAB-1])
294             ;
295         if (ptab == NULL)
296             goto done;
297     }
298
299     for (n = 0; n < 256; n++)
300         if (rtab->data[n] > qopt->buffer) break;
301     max_size = (n << qopt->rate.cell_log)-1;
302     if (ptab) {
303         int size;
304
305         for (n = 0; n < 256; n++)
306             if (ptab->data[n] > qopt->mtu) break;
307         size = (n << qopt->peakrate.cell_log)-1;
308         if (size < max_size) max_size = size;
309     }
310     if (max_size < 0)
311         goto done;
312
313     sch_tree_lock(sch);
314     q->limit = qopt->limit;
315     q->mtu = qopt->mtu;
316     q->max_size = max_size;
317     q->buffer = qopt->buffer;
318     q->tokens = q->buffer;
319     q->ptokens = q->mtu;
320     rtab = xchg(&q->R_ttab, rtab);
321     ptab = xchg(&q->P_ttab, ptab);
322     sch_tree_unlock(sch);
323     err = 0;
324 done:
325     if (rtab)
326         qdisc_put_rtab(rtab);
327     if (ptab)

```

Figure 1081: TBF - Change: `linux/net/sched/sch_tbf.c`

```

328     return err;
329 }
330
331 static int tbf_init (struct Qdisc* sch, struct rtattr *opt)
332 {
333     int err;
334     struct tbf_sched_data *q = (struct tbf_sched_data *)sch->data;
335
336     if (opt == NULL)
337         return -EINVAL;
338
339     MOD_INC_USE_COUNT;
340
341     PSCHED_GET_TIME(q->t_c);
342     init_timer (&q->wd_timer);
343     q->wd_timer.function = tbf_watchdog;
344     q->wd_timer.data = (unsigned long)sch;
345
346     if ((err = tbf_change(sch, opt)) != 0) {
347         MOD_DEC_USE_COUNT;

```

Figure 1082: TBF - Init: linux/net/sched/sch_tbf.c

```

349     return err;
350 }
351
352 static void tbf_destroy(struct Qdisc *sch)
353 {
354     struct tbf_sched_data *q = (struct tbf_sched_data *)sch->data;
355
356     del_timer(&q->wd_timer);
357
358     if (q->P_tab)
359         qdisc_put_rtab(q->P_tab);
360     if (q->R_tab)
361         qdisc_put_rtab(q->R_tab);

```

Figure 1083: TBF - Destroy: `linux/net/sched/sch_tbf.c`

```

193 static unsigned int
194 ing_hook(unsigned int hook, struct sk_buff **pskb,
195          const struct net_device *indev,
196          const struct net_device *outdev,
197          int (*okfn)(struct sk_buff *))
198 {
199
200     struct Qdisc *q;
201     struct sk_buff *skb = *pskb;
202     struct net_device *dev = skb->dev;
203     int fwres=NF_ACCEPT;
204
205     DPRINTK("ing_hook: skb=%s dev=%s len=%u\n",
206            skb->sk ? "(owned)" : "(unowned)",
207            skb->dev ? (*pskb)->dev->name : "(no dev)",
208            skb->len);
209
210     /*
211     revisit later: Use a private since lock dev->queue_lock is also
212     used on the egress (might slow things for an iota)
213     */
214
215     if (dev->qdisc_ingress) {
216         spin_lock(&dev->queue_lock);
217         if ((q = dev->qdisc_ingress) != NULL)
218             fwres = q->enqueue(skb, q);
219         spin_unlock(&dev->queue_lock);
220     }
221
222     return fwres;
223 }

```

Figure 1084: Ingress Queueing - Ingress Hook:
linux/net/sched/sch_ingress.c

```

225 /* after ipt_filter */
226 static struct nf_hook_ops ing_ops =
227 {
228     { NULL, NULL},
229     ing_hook,
230     PF_INET,
231     NF_IP_PRE_ROUTING,
232     NF_IP_PRI_FILTER + 1
233 };
234
235 int ingress_init (struct Qdisc *sch, struct rtattr *opt)
236 {
237     struct ingress_qdisc_data *p = PRIV(sch);
238
239     DPRINTK("ingress_init(sch=%p,[qdisc=%p],opt=%p)\n", sch, p, opt);
240     memset(p, 0, sizeof(*p));
241     p->filter_list = NULL;
242     p->q = &noop_qdisc;
243 #ifndef MODULE
244     if ( nf_register_hook (&ing_ops) < 0 ) {
245         printk("Unable to register ingress\n");
246         goto error;
247     }
248 #endif
249     DPRINTK("ingress_init:qdisc=%p\n", sch);
250     MOD_INC_USE_COUNT;
251     return 0;
252 #ifndef MODULE
253 error:
254 #endif
255     return -EINVAL;
256 }
257
258
259 static void ingress_reset (struct Qdisc *sch)
260 {
261     struct ingress_qdisc_data *p = PRIV(sch);
262
263     DPRINTK("ingress_reset(sch=%p,[qdisc=%p])\n", sch, p);
264
265     /*
266     #if 0
267     */
268     /* for future use */
269     qdisc_reset (p->q);
270     /*
271     #endif
272     */
273 }

```

Figure 1085: Ingress Queueing - Operations and Initialisation:
linux/net/sched/sch_ingress.c

```

280 static void ingress_destroy(struct Qdisc *sch)
281 {
282     struct ingress_qdisc_data *p = PRIV(sch);
283     struct tcf_proto *tp;
284
285     DPRINTK("ingress_destroy(sch=%p,[qdisc=%p])\n", sch, p);
286     while (p->filter_list) {
287         tp = p->filter_list;
288         p->filter_list = tp->next;
289         tp->ops->destroy(tp);
290     }
291     memset(p, 0, sizeof(*p));
292     p->filter_list = NULL;
293
294     #if 0
295     /* for future use */
296     qdisc_destroy(p->q);
297     #endif
298
299     #ifndef MODULE
300     nf_unregister_hook(&ing_ops);
301     #endif
302
303     MOD_DEC_USE_COUNT;
304 }

```

Figure 1086: Ingress Queueing - Destroy: `linux/net/sched/sch_ingress.c`


```
41 struct prio_sched_data
42 {
43     int bands;
44     struct tcf_proto * filter_list ;
45     u8 prio2band[TC_PRIO_MAX+1];
46     struct Qdisc *queues[TCQ_PRIO_BANDS];
47 };
```

Figure 1087: Priority Queue: `linux/net/sched/sch.prio.c`

```

50 static __inline__ unsigned prio_classify (struct sk_buff *skb, struct Qdisc *sch)
51 {
52     struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
53     struct tcf_result res;
54     u32 band;
55
56     band = skb->priority;
57     if (TC_H_MAJ(skb->priority) != sch->handle) {
58         if (!q->filter_list || tc_classify (skb, q->filter_list, &res)) {
59             if (TC_H_MAJ(band))
60                 band = 0;
61             return q->prio2band[band&TC_PRIO_MAX];
62         }
63         band = res.classid;
64     }
65     band = TC_H_MIN(band) - 1;
66     return band < q->bands ? band : q->prio2band[0];
67 }

```

Figure 1088: Priority Classifier: `linux/net/sched/sch_prio.c`

```

69 static int
70 prio_enqueue(struct sk_buff *skb, struct Qdisc* sch)
71 {
72     struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
73     struct Qdisc *qdisc;
74     int ret;
75
76     qdisc = q->queues[prio_classify(skb, sch)];
77
78     if ((ret = qdisc->enqueue(skb, qdisc)) == 0) {
79         sch->stats.bytes += skb->len;
80         sch->stats.packets++;
81         sch->q.qlen++;
82         return 0;
83     }
84     sch->stats.drops++;
85     return ret;
86 }

```

Figure 1089: Priority Enqueue: `linux/net/sched/sch_prio.c`

```

88
89 static int
90 prio_requeue(struct sk_buff *skb, struct Qdisc* sch)
91 {
92     struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
93     struct Qdisc *qdisc;
94     int ret;
95
96     qdisc = q->queues[prio_classify(skb, sch)];
97
98     if ((ret = qdisc->ops->requeue(skb, qdisc)) == 0) {
99         sch->q.qlen++;
100         return 0;
101     }
102     sch->stats.drops++;
103     return ret;
104 }
105
106
107 static struct sk_buff *
108 prio_dequeue(struct Qdisc* sch)
109 {
110     struct sk_buff *skb;
111     struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
112     int prio;
113     struct Qdisc *qdisc;
114
115     for (prio = 0; prio < q->bands; prio++) {
116         qdisc = q->queues[prio];
117         skb = qdisc->dequeue(qdisc);
118         if (skb) {
119             sch->q.qlen--;
120             return skb;
121         }
122     }
123     return NULL;
124 }
125

```

Figure 1090: Priority Requeue and Dequeue: `linux/net/sched/sch_prio.c`

```

127 static int
128 prio_drop(struct Qdisc* sch)
129 {
130     struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
131     int prio;
132     struct Qdisc *qdisc;
133
134     for (prio = q->bands-1; prio >= 0; prio--) {
135         qdisc = q->queues[prio];
136         if (qdisc->ops->drop(qdisc)) {
137             sch->q.len--;
138             return 1;
139         }
140     }
141     return 0;
142 }

```

Figure 1091: Priority Drop: `linux/net/sched/sch_prio.c`

```

169 static int prio_tune(struct Qdisc *sch, struct rtattr *opt)
170 {
171     struct prio_sched_data *q = (struct prio_sched_data *)sch->data;
172     struct tc_prio_qopt *qopt = RTA_DATA(opt);
173     int i;
174
175     if (opt->rta_len < RTA_LENGTH(sizeof(*qopt)))
176         return -EINVAL;
177     if (qopt->bands > TCQ_PRIO_BANDS || qopt->bands < 2)
178         return -EINVAL;
179
180     for (i=0; i<=TC_PRIO_MAX; i++) {
181         if (qopt->priomap[i] >= qopt->bands)
182             return -EINVAL;
183     }
184
185     sch_tree_lock(sch);
186     q->bands = qopt->bands;
187     memcpy(q->prio2band, qopt->priomap, TC_PRIO_MAX+1);
188
189     for (i=q->bands; i<TCQ_PRIO_BANDS; i++) {
190         struct Qdisc *child = xchg(&q->queues[i], &noop_qdisc);
191         if (child != &noop_qdisc)
192             qdisc_destroy(child);
193     }
194     sch_tree_unlock(sch);
195
196     for (i=0; i<=TC_PRIO_MAX; i++) {
197         int band = q->prio2band[i];
198         if (q->queues[band] == &noop_qdisc) {
199             struct Qdisc *child;
200             child = qdisc_create_dflt(sch->dev, &pfifo_qdisc_ops);
201             if (child) {
202                 sch_tree_lock(sch);
203                 child = xchg(&q->queues[band], child);
204
205                 if (child != &noop_qdisc)
206                     qdisc_destroy(child);
207                 sch_tree_unlock(sch);
208             }
209         }
210     }
211     return 0;
212 }

```

Figure 1092: Priority - Tune: `linux/net/sched/sch_prio.c`

```

41  /*      Stochastic Fairness Queuing algorithm.
42          =====
43
44      Source:
45      Paul E. McKenney "Stochastic Fairness Queuing",
46      IEEE INFOCOMM'90 Proceedings, San Francisco, 1990.
47
48      Paul E. McKenney "Stochastic Fairness Queuing",
49      "Interworking: Research and Experience", v.2, 1991, p.113-131.
50
51
52      See also:
53      M. Shreedhar and George Varghese "Efficient Fair
54      Queuing using Deficit Round Robin", Proc. SIGCOMM 95.
55
56
57      This is not the thing that is usually called (W)FQ nowadays.
58      It does not use any timestamp mechanism, but instead
59      processes queues in round-robin order.
60
61      ADVANTAGE:
62
63      - It is very cheap. Both CPU and memory requirements are minimal.
64
65      DRAWBACKS:
66
67      - "Stochastic" -> It is not 100% fair.
68      When hash collisions occur, several flows are considered as one.
69
70      - "Round-robin" -> It introduces larger delays than virtual clock
71      based schemes, and should not be used for isolating interactive
72      traffic from non-interactive. It means, that this scheduler
73      should be used as leaf of CBQ or P3, which put interactive traffic
74      to higher priority band.
75
76      We still need true WFQ for top level CSZ, but using WFQ
77      for the best effort traffic is absolutely pointless:
78      SFQ is superior for this purpose.
79
80      IMPLEMENTATION:
81      This implementation limits maximal queue length to 128;
82      maximal mtu to 2^15-1; number of hash buckets to 1024.
83      The only goal of this restrictions was that all data
84      fit into one 4K page :-). Struct sfq_sched_data is
85      organized in anti-cache manner: all the data for a bucket
86      are scattered over different locations. This is not good,
87      but it allowed me to put it into 4K.
88
89      It is easy to increase these values, but not in flight. */

```

Figure 1093: Stochastic Fairness Queuing Scheduler - Introductory Remarks:
linux/net/sched/sch_sfq.c

```

91 #define SFQ_DEPTH          128
92 #define SFQ_HASH_DIVISOR  1024
93
94 /* This type should contain at least SFQ_DEPTH*2 values */
95 typedef unsigned char sfq_index;
96
97 struct sfq_head
98 {
99     sfq_index    next;
100    sfq_index    prev;
101 };
102
103 struct sfq_sched_data
104 {
105     /* Parameters */
106     int          perturb_period;
107     unsigned    quantum;        /* Allotment per round: MUST BE >= MTU */
108
109     /* Variables */
110     struct timer_list perturb_timer;
111     int          perturbation;
112     sfq_index    tail;          /* Index of current slot in round */
113     sfq_index    max_depth;     /* Maximal depth */
114
115     sfq_index    ht[SQ_HASH_DIVISOR]; /* Hash table */
116     sfq_index    next[SQ_DEPTH];     /* Active slots link */
117     short        allot[SQ_DEPTH];    /* Current allotment per slot */
118     unsigned short hash[SQ_DEPTH];   /* Hash value indexed by slots */
119     struct sk_buff_head qs[SQ_DEPTH]; /* Slot queue */
120     struct sfq_head dep[SQ_DEPTH*2]; /* Linked list of slots, indexed by
121     depth */
};

```

Figure 1094: SFQ - Data Structures: linux/net/sched/sch_sfq.c


```

1  /*
2  * net/sched/sch_prio.c Simple 3-band priority "scheduler".
3  *
4  *       This program is free software; you can redistribute it and/or
5  *       modify it under the terms of the GNU General Public License
6  *       as published by the Free Software Foundation; either version
7  *       2 of the License, or (at your option) any later version.
8  *
9  * Authors:   Alexey Kuznetsov, <kuznet@ms2.inr.ac.ru>
10 * Fixes:    19990609: J Hadi Salim <hadi@nortelnetworks.com>:
11 *           Init -- EINVAL when opt undefined
12 */
13
14 #include <linux/config.h>
15 #include <linux/module.h>
16 #include <asm/uaccess.h>
17 #include <asm/system.h>
18 #include <asm/bitops.h>
19 #include <linux/types.h>
20 #include <linux/kernel.h>
21 #include <linux/sched.h>
22 #include <linux/string.h>
23 #include <linux/mm.h>
24 #include <linux/socket.h>
25 #include <linux/sockios.h>
26 #include <linux/in.h>
27 #include <linux/errno.h>
28 #include <linux/interrupt.h>
29 #include <linux/if_ether.h>
30 #include <linux/inet.h>

```

Figure 1095: Priority Queue: linux/net/sched/sch_prio.c

```

41  /*      Stochastic Fairness Queuing algorithm.
42  =====
43
44  Source:
45  Paul E. McKenney "Stochastic Fairness Queuing",
46  IEEE INFOCOMM'90 Proceedings, San Francisco, 1990.
47
48  Paul E. McKenney "Stochastic Fairness Queuing",
49  "Interworking: Research and Experience", v.2, 1991, p.113-131.
50
51
52  See also:
53  M. Shreedhar and George Varghese "Efficient Fair
54  Queuing using Deficit Round Robin", Proc. SIGCOMM 95.
55
56
57  This is not the thing that is usually called (W)FQ nowadays.
58  It does not use any timestamp mechanism, but instead
59  processes queues in round-robin order.
60
61  ADVANTAGE:
62
63  - It is very cheap. Both CPU and memory requirements are minimal.
64
65  DRAWBACKS:
66
67  - "Stochastic" -> It is not 100% fair.
68  When hash collisions occur, several flows are considered as one.
69
70  - "Round-robin" -> It introduces larger delays than virtual clock
71  based schemes, and should not be used for isolating interactive
72  traffic from non-interactive. It means, that this scheduler
73  should be used as leaf of CBQ or P3, which put interactive traffic
74  to higher priority band.
75
76  We still need true WFQ for top level CSZ, but using WFQ
77  for the best effort traffic is absolutely pointless:
78  SFQ is superior for this purpose.
79
80  IMPLEMENTATION:
81  This implementation limits maximal queue length to 128;
82  maximal mtu to 2^15-1; number of hash buckets to 1024.
83  The only goal of this restrictions was that all data
84  fit into one 4K page :-). Struct sfq_sched_data is
85  organized in anti-cache manner: all the data for a bucket
86  are scattered over different locations. This is not good,
87  but it allowed me to put it into 4K.
88
89  It is easy to increase these values, but not in flight. */

```

Figure 1096: Stochastic Fairness Queuing Scheduler - Introductory Remarks:
linux/net/sched/sch_sfq.c

```

91 #define SFQ_DEPTH          128
92 #define SFQ_HASH_DIVISOR  1024
93
94 /* This type should contain at least SFQ_DEPTH*2 values */
95 typedef unsigned char sfq_index;
96
97 struct sfq_head
98 {
99     sfq_index    next;
100    sfq_index    prev;
101 };
102
103 struct sfq_sched_data
104 {
105     /* Parameters */
106     int          perturb_period;
107     unsigned    quantum;        /* Allotment per round: MUST BE >= MTU */
108
109     /* Variables */
110     struct timer_list perturb_timer;
111     int          perturbation;
112     sfq_index    tail;          /* Index of current slot in round */
113     sfq_index    max_depth;     /* Maximal depth */
114
115     sfq_index    ht[SFQ_HASH_DIVISOR]; /* Hash table */
116     sfq_index    next[SFQ_DEPTH];     /* Active slots link */
117     short        allot[SFQ_DEPTH];    /* Current allotment per slot */
118     unsigned short hash[SFQ_DEPTH];   /* Hash value indexed by slots */
119     struct sk_buff_head qs[SFQ_DEPTH]; /* Slot queue */
120     struct sfq_head dep[SFQ_DEPTH*2]; /* Linked list of slots, indexed by
121     depth */
};

```

Figure 1097: SFQ - Data Structures: linux/net/sched/sch_sfq.c

```

123 static __inline__ unsigned sfq_fold_hash(struct sfq_sched_data *q, u32 h, u32 h1)
124 {
125     int pert = q->perturbation;
126
127     /* Have we any rotation primitives? If not, WHY? */
128     h ^= (h1<<pert) ^ (h1>>(0x1F - pert));
129     h ^= h>>10;
130     return h & 0x3FFF;
131 }
132
133 #ifndef IPPROTO_ESP
134 #define IPPROTO_ESP 50
135 #endif
136
137 static unsigned sfq_hash(struct sfq_sched_data *q, struct sk_buff *skb)
138 {
139     u32 h, h2;
140
141     switch (skb->protocol) {
142     case __constant_htons(ETH_P_IP):
143     {
144         struct iphdr *iph = skb->nh.iph;
145         h = iph->daddr;
146         h2 = iph->saddr ^ iph->protocol;
147         if (!(iph->frag_off & htons(IP_MF|IP_OFFSET)) &&
148             (iph->protocol == IPPROTO_TCP ||
149             iph->protocol == IPPROTO_UDP ||
150             iph->protocol == IPPROTO_ESP))
151             h2 ^= (((u32*)iph) + iph->ihl);
152         break;
153     }
154     case __constant_htons(ETH_P_IPV6):
155     {
156         struct ipv6hdr *iph = skb->nh.ipv6h;
157         h = iph->daddr.s6_addr32[3];
158         h2 = iph->saddr.s6_addr32[3] ^ iph->nexthdr;
159         if (iph->nexthdr == IPPROTO_TCP ||
160             iph->nexthdr == IPPROTO_UDP ||
161             iph->nexthdr == IPPROTO_ESP)
162             h2 ^= *(u32*)&iph[1];
163         break;
164     }
165     default:
166         h = (u32)(unsigned long)skb->dst ^ skb->protocol;
167         h2 = (u32)(unsigned long)skb->sk;
168     }
169     return sfq_fold_hash(q, h, h2);
170 }

```

Figure 1098: SFQ - Queue Hash Functions: `linux/net/sched/sch_sfq.c`

```

172 extern __inline__ void sfq_link (struct sfq_sched_data *q, sfq_index x)
173 {
174     sfq_index p, n;
175     int d = q->qs[x].qlen + SFQ_DEPTH;
176
177     p = d;
178     n = q->dep[d].next;
179     q->dep[x].next = n;
180     q->dep[x].prev = p;
181     q->dep[p].next = q->dep[n].prev = x;
182 }
183
184 extern __inline__ void sfq_dec(struct sfq_sched_data *q, sfq_index x)
185 {
186     sfq_index p, n;
187
188     n = q->dep[x].next;
189     p = q->dep[x].prev;
190     q->dep[p].next = n;
191     q->dep[n].prev = p;
192
193     if (n == p && q->max_depth == q->qs[x].qlen + 1)
194         q->max_depth--;
195
196     sfq_link (q, x);
197 }
198
199 extern __inline__ void sfq_inc(struct sfq_sched_data *q, sfq_index x)
200 {
201     sfq_index p, n;
202     int d;
203
204     n = q->dep[x].next;
205     p = q->dep[x].prev;
206     q->dep[p].next = n;
207     q->dep[n].prev = p;
208     d = q->qs[x].qlen;
209     if (q->max_depth < d)
210         q->max_depth = d;
211
212     sfq_link (q, x);
213 }

```

Figure 1099: SFQ - Queue Decrement/Increment Functions:
linux/net/sched/sch_sfq.c

```

215 static int sfq_drop(struct Qdisc *sch)
216 {
217     struct sfq_sched_data *q = (struct sfq_sched_data *)sch->data;
218     sfq_index d = q->max_depth;
219     struct sk_buff *skb;
220
221     /* Queue is full! Find the longest slot and
222        drop a packet from it */
223
224     if (d > 1) {
225         sfq_index x = q->dep[d+SFQ_DEPTH].next;
226         skb = q->qs[x].prev;
227         __skb_unlink(skb, &q->qs[x]);
228         kfree_skb(skb);
229         sfq_dec(q, x);
230         sch->q.qlen--;
231         sch->stats.drops++;
232         return 1;
233     }
234
235     if (d == 1) {
236         /* It is difficult to believe, but ALL THE SLOTS HAVE LENGTH 1. */
237         d = q->next[q->tail];
238         q->next[q->tail] = q->next[d];
239         q->allot[q->next[d]] += q->quantum;
240         skb = q->qs[d].prev;
241         __skb_unlink(skb, &q->qs[d]);
242         kfree_skb(skb);
243         sfq_dec(q, d);
244         sch->q.qlen--;
245         q->ht[q->hash[d]] = SFQ_DEPTH;
246         sch->stats.drops++;
247         return 1;
248     }
249
250     return 0;
251 }

```

Figure 1100: SFQ - Queue Drop Function: `linux/net/sched/sch_sfq.c`

```

253 static int
254 sfq_enqueue(struct sk_buff *skb, struct Qdisc* sch)
255 {
256     struct sfq_sched_data *q = (struct sfq_sched_data *)sch->data;
257     unsigned hash = sfq_hash(q, skb);
258     sfq_index x;
259
260     x = q->ht[hash];
261     if (x == SFQ_DEPTH) {
262         q->ht[hash] = x = q->dep[SFQ_DEPTH].next;
263         q->hash[x] = hash;
264     }
265     __skb_queue_tail(&q->qs[x], skb);
266     sfq_inc(q, x);
267     if (q->qs[x].qlen == 1) { /* The flow is new */
268         if (q->tail == SFQ_DEPTH) { /* It is the first flow */
269             q->tail = x;
270             q->next[x] = x;
271             q->allot[x] = q->quantum;
272         } else {
273             q->next[x] = q->next[q->tail];
274             q->next[q->tail] = x;
275             q->tail = x;
276         }
277     }
278     if (++sch->q.qlen < SFQ_DEPTH-1) {
279         sch->stats.bytes += skb->len;
280         sch->stats.packets++;
281         return 0;
282     }
283
284     sfq_drop(sch);
285     return NET_XMIT_CN;
286 }

```

Figure 1101: SFQ - Enqueue Function: `linux/net/sched/sch_sfq.c`

```

255 {
256     struct sfq_sched_data *q = (struct sfq_sched_data *)sch->data;
257     unsigned hash = sfq_hash(q, skb);
258     sfq_index x;
259
260     x = q->ht[hash];
261     if (x == SFQ_DEPTH) {
262         q->ht[hash] = x = q->dep[SFQ_DEPTH].next;
263         q->hash[x] = hash;
264     }
265     __skb_queue_tail(&q->qs[x], skb);
266     sfq_inc(q, x);
267     if (q->qs[x].qlen == 1) { /* The flow is new */
268         if (q->tail == SFQ_DEPTH) { /* It is the first flow */
269             q->tail = x;
270             q->next[x] = x;
271             q->allot[x] = q->quantum;
272         } else {
273             q->next[x] = q->next[q->tail];
274             q->next[q->tail] = x;
275             q->tail = x;
276         }
277     }
278     if (++sch->q.qlen < SFQ_DEPTH-1) {
279         sch->stats.bytes += skb->len;
280         sch->stats.packets++;
281         return 0;
282     }
283
284     sfq_drop(sch);
285     return NET_XMIT_CN;
286 }
287
288 static int
289 sfq_requeue(struct sk_buff *skb, struct Qdisc* sch)
290 {
291     struct sfq_sched_data *q = (struct sfq_sched_data *)sch->data;
292     unsigned hash = sfq_hash(q, skb);
293     sfq_index x;
294
295     x = q->ht[hash];
296     if (x == SFQ_DEPTH) {
297         q->ht[hash] = x = q->dep[SFQ_DEPTH].next;
298         q->hash[x] = hash;
299     }
300     __skb_queue_head(&q->qs[x], skb);
301     sfq_inc(q, x);
302     if (q->qs[x].qlen == 1) { /* The flow is new */
303         if (q->tail == SFQ_DEPTH) { /* It is the first flow */
304             q->tail = x;
305             q->next[x] = x;
306             q->allot[x] = q->quantum;
307         } else {
308             q->next[x] = q->next[q->tail];
309             q->next[q->tail] = x;
310             q->tail = x;
311         }
312     }
313     if (++sch->q.qlen < SFQ_DEPTH-1)
314         return 0;
315
316     sch->stats.drops++;
317     sfq_drop(sch);
318     return NET_XMIT_CN;
319 }

```

Figure 1102: SFQ - Requeue Function: linux/net/sched/sch_sfq.c


```

324 static struct sk_buff *
325 sfq_dequeue(struct Qdisc* sch)
326 {
327     struct sfq_sched_data *q = (struct sfq_sched_data *)sch->data;
328     struct sk_buff *skb;
329     sfq_index a, old_a;
330
331     /* No active slots */
332     if (q->tail == SFQ_DEPTH)
333         return NULL;
334
335     a = old_a = q->next[q->tail];
336
337     /* Grab packet */
338     skb = __skb_dequeue(&q->qs[a]);
339     sfq_dec(q, a);
340     sch->q.len--;
341
342     /* Is the slot empty? */
343     if (q->qs[a].qlen == 0) {
344         a = q->next[a];
345         if (a == old_a) {
346             q->tail = SFQ_DEPTH;
347             return skb;
348         }
349         q->next[q->tail] = a;
350         q->allot[a] += q->quantum;
351     } else if ((q->allot[a] -= skb->len) <= 0) {
352         q->tail = a;
353         a = q->next[a];
354         q->allot[a] += q->quantum;
355     }
356     return skb;
357 }

```

Figure 1103: SFQ - Dequeue Function: `linux/net/sched/sch_sfq.c`

```
359 static void
360 sfq_reset (struct Qdisc* sch)
361 {
362     struct sk_buff *skb;
363
364     while ((skb = sfq_dequeue(sch)) != NULL)
365         kfree_skb(skb);
366 }
```

Figure 1104: SFQ - Reset: `linux/net/sched/sch_sfq.c`

```

368 static void sfq_perturbation(unsigned long arg)
369 {
370     struct Qdisc *sch = (struct Qdisc*)arg;
371     struct sfq_sched_data *q = (struct sfq_sched_data *)sch->data;
372
373     q->perturbation = net_random()&0x1F;
374     q->perturb_timer.expires = jiffies + q->perturb_period;
375
376     if (q->perturb_period) {
377         q->perturb_timer.expires = jiffies + q->perturb_period;
378         add_timer(&q->perturb_timer);
379     }
380 }

```

Figure 1105: SFQ - Perturb Queue: `linux/net/sched/sch_sfq.c`

```

37  /*
38  How to setup it.
39  -----
40
41  After loading this module you will find a new device teqlN
42  and new qdisc with the same name. To join a slave to the equalizer
43  you should just set this qdisc on a device f.e.
44
45  # tc qdisc add dev eth0 root teql0
46  # tc qdisc add dev eth1 root teql0
47
48  That's all. Full PnP 8)
49
50  Applicability.
51  -----
52
53  1. Slave devices MUST be active devices, i.e., they must raise the tbusy
54  signal and generate EOI events. If you want to equalize virtual devices
55  like tunnels, use a normal eql device.
56
57  2. This device puts no limitations on physical slave characteristics
58  f.e. it will equalize 9600baud line and 100Mb ethernet perfectly :-)
59  Certainly, large difference in link speeds will make the resulting
60  equalized link unusable, because of huge packet reordering.
61  I estimate an upper useful difference as ~10 times.
62
63  3. If the slave requires address resolution, only protocols using
64  neighbour cache (IPv4/IPv6) will work over the equalized link.
65  Other protocols are still allowed to use the slave device directly,
66  which will not break load balancing, though native slave
67  traffic will have the highest priority. */

```

Figure 1106: Traffic Equalisation/Load Balance Queue - Initial Remarks:
linux/net/sched/sch_teql.c

```

67 struct teql_master
68 {
69     struct Qdisc_ops qops;
70     struct net_device dev;
71     struct Qdisc *slaves;
72     struct net_device_stats stats;
73 };
74
75 struct teql_sched_data
76 {
77     struct Qdisc *next;
78     struct teql_master *m;
79     struct neighbour *ncache;
80     struct sk_buff_head q;
81 };
82
83 #define NEXT_SLAVE(q) (((struct teql_sched_data*)((q)->data))->next)
84
85 #define FMASK (IFF_BROADCAST|IFF_POINTOPOINT|IFF_BROADCAST)

```

Figure 1107: TE Data: linux/net/sched/sch_teql.c

```

87  /* "teql*" qdisc routines */
88
89  static int
90  teql_enqueue(struct sk_buff *skb, struct Qdisc* sch)
91  {
92      struct net_device *dev = sch->dev;
93      struct teql_sched_data *q = (struct teql_sched_data *)sch->data;
94
95      __skb_queue_tail(&q->q, skb);
96      if (q->q.len <= dev->tx_queue_len) {
97          sch->stats.bytes += skb->len;
98          sch->stats.packets++;
99          return 0;
100     }
101
102     __skb_unlink(skb, &q->q);
103     kfree_skb(skb);
104     sch->stats.drops++;
105     return NET_XMIT_DROP;
106 }
107
108 static int
109 teql_requeue(struct sk_buff *skb, struct Qdisc* sch)
110 {
111     struct teql_sched_data *q = (struct teql_sched_data *)sch->data;
112
113     __skb_queue_head(&q->q, skb);
114     return 0;
115 }
116
117 static struct sk_buff *
118 teql_dequeue(struct Qdisc* sch)
119 {
120     struct teql_sched_data *dat = (struct teql_sched_data *)sch->data;
121     struct sk_buff *skb;
122
123     skb = __skb_dequeue(&dat->q);
124     if (skb == NULL) {
125         struct net_device *m = dat->m->dev.qdisc->dev;
126         if (m) {
127             dat->m->slaves = sch;
128             netif_wake_queue(m);
129         }
130     }
131     sch->q.len = dat->q.len + dat->m->dev.qdisc->q.len;
132     return skb;
133 }

```

Figure 1108: TE Enqueue,Requeue,Dequeue: `linux/net/sched/sch_teql.c`

```

135 static __inline__ void
136 teql_neigh_release (struct neighbour *n)
137 {
138     if (n)
139         neigh_release(n);
140 }
141
142 static void
143 teql_reset (struct Qdisc* sch)
144 {
145     struct teql_sched_data *dat = (struct teql_sched_data *)sch->data;
146
147     skb_queue_purge(&dat->q);
148     sch->q.len = 0;
149     teql_neigh_release (xchg(&dat->ncache, NULL));
150 }
151
152 static void
153 teql_destroy (struct Qdisc* sch)
154 {
155     struct Qdisc *q, *prev;
156     struct teql_sched_data *dat = (struct teql_sched_data *)sch->data;
157     struct teql_master *master = dat->m;
158
159     if ((prev = master->slaves) != NULL) {
160         do {
161             q = NEXT_SLAVE(prev);
162             if (q == sch) {
163                 NEXT_SLAVE(prev) = NEXT_SLAVE(q);
164                 if (q == master->slaves) {
165                     master->slaves = NEXT_SLAVE(q);
166                     if (q == master->slaves) {
167                         master->slaves = NULL;
168                         spin_lock_bh(&master->dev.queue_lock);
169                         qdisc_reset (master->dev.qdisc);
170                         spin_unlock_bh(&master->dev.queue_lock)
171                             ;
172                     }
173                 }
174                 skb_queue_purge(&dat->q);
175                 teql_neigh_release (xchg(&dat->ncache, NULL));
176                 break;
177             }
178         } while ((prev = q) != master->slaves);
179     }
180
181     MOD_DEC_USE_COUNT;
182 }

```

Figure 1109: TE Release, Reset, Destroy: `linux/net/sched/sch_teql.c`

```

184 static int teql_qdisc_init (struct Qdisc *sch, struct rtattr *opt)
185 {
186     struct net_device *dev = sch->dev;
187     struct teql_master *m = (struct teql_master*)sch->ops;
188     struct teql_sched_data *q = (struct teql_sched_data *)sch->data;
189
190     if (dev->hard_header_len > m->dev.hard_header_len)
191         return -EINVAL;
192
193     if (&m->dev == dev)
194         return -ELOOP;
195
196     q->m = m;
197
198     skb_queue_head_init(&q->q);
199
200     if (m->slaves) {
201         if (m->dev.flags & IFF_UP) {
202             if ((m->dev.flags&IFF_POINTOPOINT && !(dev->flags&
203                 IFF_POINTOPOINT))
204                 || (m->dev.flags&IFF_BROADCAST && !(dev->flags&
205                     IFF_BROADCAST))
206                 || (m->dev.flags&IFF_MULTICAST && !(dev->flags&
207                     IFF_MULTICAST))
208                 || dev->mtu < m->dev.mtu)
209                 return -EINVAL;
210         } else {
211             if (!(dev->flags&IFF_POINTOPOINT))
212                 m->dev.flags &= ~IFF_POINTOPOINT;
213             if (!(dev->flags&IFF_BROADCAST))
214                 m->dev.flags &= ~IFF_BROADCAST;
215             if (!(dev->flags&IFF_MULTICAST))
216                 m->dev.flags &= ~IFF_MULTICAST;
217             if (dev->mtu < m->dev.mtu)
218                 m->dev.mtu = dev->mtu;
219         }
220         q->next = NEXT_SLAVE(m->slaves);
221         NEXT_SLAVE(m->slaves) = sch;
222     } else {
223         q->next = sch;
224         m->slaves = sch;
225         m->dev.mtu = dev->mtu;
226         m->dev.flags = (m->dev.flags&~FMASK)|(dev->flags&FMASK);
227     }
228
229     MOD_INC_USE_COUNT;
230     return 0;
231 }

```

Figure 1110: TE Initialisation: linux/net/sched/sch_teql.c


```

230 /* "teql*" netdevice routines */
231
232 static int
233 _teql_resolve (struct sk_buff *skb, struct sk_buff *skb_res, struct net_device *dev)
234 {
235     struct teql_sched_data *q = (void*)dev->qdisc->data;
236     struct neighbour *mn = skb->dst->neighbour;
237     struct neighbour *n = q->n-cache;
238
239     if (mn->tbl == NULL)
240         return -EINVAL;
241     if (n && n->tbl == mn->tbl &&
242         memcmp(n->primary_key, mn->primary_key, mn->tbl->key_len) == 0) {
243         atomic_inc(&n->refcnt);
244     } else {
245         n = _neigh_lookup_errno(mn->tbl, mn->primary_key, dev);
246         if (IS_ERR(n))
247             return PTR_ERR(n);
248     }
249     if (neigh_event_send(n, skb_res) == 0) {
250         int err;
251         read_lock(&n->lock);
252         err = dev->hard_header(skb, dev, ntohs(skb->protocol), n->ha, NULL,
253                               skb->len);
254         read_unlock(&n->lock);
255         if (err < 0) {
256             neigh_release(n);
257             return -EINVAL;
258         }
259         teql_neigh_release (xchg(&q->n-cache, n));
260         return 0;
261     }
262     neigh_release(n);
263     return (skb_res == NULL) ? -EAGAIN : 1;
264 }
265
266 static __inline__ int
267 teql_resolve (struct sk_buff *skb, struct sk_buff *skb_res, struct net_device *dev)
268 {
269     if (dev->hard_header == NULL ||
270         skb->dst == NULL ||
271         skb->dst->neighbour == NULL)
272         return 0;
273     return _teql_resolve (skb, skb_res, dev);
274 }

```

Figure 1111: TE net device resolve: `linux/net/sched/sch_teql.c`

```

275 static int teql_master_xmit(struct sk_buff *skb, struct net_device *dev)
276 {
277     struct teql_master *master = (void*)dev->priv;
278     struct Qdisc *start, *q;
279     int busy;
280     int nores;
281     int len = skb->len;
282     struct sk_buff *skb_res = NULL;
283
284     start = master->slaves;
285
286 restart :
287     nores = 0;
288     busy = 0;
289
290     if ((q = start) == NULL)
291         goto drop;

```

Figure 1112: TE net device transmit: `linux/net/sched/sch_teql.c`

```

293     do {
294         struct net_device *slave = q->dev;
295
296         if (slave->qdisc_sleeping != q)
297             continue;
298         if (netif_queue_stopped(slave) || ! netif_running(slave)) {
299             busy = 1;
300             continue;
301         }
302
303         switch (teql_resolve(skb, skb_res, slave)) {
304         case 0:
305             if (spin_trylock(&slave->xmit_lock)) {
306                 slave->xmit_lock_owner = smp_processor_id();
307                 if (!netif_queue_stopped(slave) &&
308                     slave->hard_start_xmit(skb, slave) == 0) {
309                     slave->xmit_lock_owner = -1;
310                     spin_unlock(&slave->xmit_lock);
311                     master->slaves = NEXT_SLAVE(q);
312                     netif_wake_queue(dev);
313                     master->stats.tx_packets++;
314                     master->stats.tx_bytes += len;
315                     return 0;
316                 }
317                 slave->xmit_lock_owner = -1;
318                 spin_unlock(&slave->xmit_lock);
319             }
320             if (netif_queue_stopped(dev))
321                 busy = 1;
322             break;
323         case 1:
324             master->slaves = NEXT_SLAVE(q);
325             return 0;
326         default:
327             nores = 1;
328             break;
329         }
330         __skb_pull(skb, skb->nh.raw - skb->data);
331     } while ((q = NEXT_SLAVE(q)) != start);

```

Figure 1113: TE net device transmit ii: `linux/net/sched/sch_teql.c`

```

333     if (nores && skb_res == NULL) {
334         skb_res = skb;
335         goto restart ;
336     }
337
338     if (busy) {
339         netif_stop_queue(dev);
340         return 1;
341     }
342     master->stats.tx_errors++;
343
344 drop:
345     master->stats.tx_dropped++;
346     dev_kfree_skb(skb);
347     return 0;
348 }

```

Figure 1114: TE net device transmit iii: `linux/net/sched/sch_teql.c`

```

350 static int teql_master_open(struct net_device *dev)
351 {
352     struct Qdisc *q;
353     struct teql_master *m = (void*)dev->priv;
354     int mtu = 0xFFFFE;
355     unsigned flags = IFF_NOARP|IFF_MULTICAST;
356
357     if (m->slaves == NULL)
358         return -EUNATCH;
359
360     flags = FMASK;
361
362     q = m->slaves;
363     do {
364         struct net_device *slave = q->dev;
365
366         if (slave == NULL)
367             return -EUNATCH;
368
369         if (slave->mtu < mtu)
370             mtu = slave->mtu;
371         if (slave->hard_header_len > LL_MAX_HEADER)
372             return -EINVAL;
373
374         /* If all the slaves are BROADCAST, master is BROADCAST
375            If all the slaves are PtP, master is PtP
376            Otherwise, master is NBMA.
377            */
378         if (!(slave->flags&IFF_POINTOPOINT))
379             flags &= ~IFF_POINTOPOINT;
380         if (!(slave->flags&IFF_BROADCAST))
381             flags &= ~IFF_BROADCAST;
382         if (!(slave->flags&IFF_MULTICAST))
383             flags &= ~IFF_MULTICAST;
384     } while ((q = NEXT_SLAVE(q)) != m->slaves);
385
386     m->dev.mtu = mtu;
387     m->dev.flags = (m->dev.flags&~FMASK) | flags;
388     netif_start_queue(&m->dev);
389     MOD_INC_USE_COUNT;
390     return 0;
391 }

```

Figure 1115: TE net device open: linux/net/sched/sch_teql.c

```
393 static int teql_master_close(struct net_device *dev)
394 {
395     netif_stop_queue(dev);
396     MOD_DEC_USE_COUNT;
397     return 0;
398 }
399
400 static struct net_device_stats *teql_master_stats(struct net_device *dev)
401 {
402     struct teql_master *m = (void*)dev->priv;
403     return &m->stats;
404 }
```

Figure 1116: TE net device close and stats: `linux/net/sched/sch_teql.c`

```

406 static int teql_master_mtu(struct net_device *dev, int new_mtu)
407 {
408     struct teql_master *m = (void*)dev->priv;
409     struct Qdisc *q;
410
411     if (new_mtu < 68)
412         return -EINVAL;
413
414     q = m->slaves;
415     if (q) {
416         do {
417             if (new_mtu > q->dev->mtu)
418                 return -EINVAL;
419             } while ((q=NEXT_SLAVE(q)) != m->slaves);
420     }
421
422     dev->mtu = new_mtu;
423     return 0;
424 }

```

Figure 1117: TE net device mtu: `linux/net/sched/sch_teql.c`

```

426 static int teql_master_init (struct net_device *dev)
427 {
428     dev->open          = teql_master_open;
429     dev->hard_start_xmit = teql_master_xmit;
430     dev->stop          = teql_master_close;
431     dev->get_stats     = teql_master_stats;
432     dev->change_mtu    = teql_master_mtu;
433     dev->type          = ARPHRD_VOID;
434     dev->mtu           = 1500;
435     dev->tx_queue_len  = 100;
436     dev->flags         = IFF_NOARP;
437     dev->hard_header_len = LL_MAX_HEADER;
438     return 0;
439 }
440
441 static struct teql_master the_master = {
442 {
443     NULL,
444     NULL,
445     "",
446     sizeof (struct teql_sched_data),
447
448     teql_enqueue,
449     teql_dequeue,
450     teql_requeue,
451     NULL,
452
453     teql_qdisc_init ,
454     teql_reset ,
455     teql_destroy ,
456     NULL,
457 },};

```

Figure 1118: TE net device master initialisation:
linux/net/sched/sch_teql.c


```

459 #ifdef MODULE
460 int init_module(void)
461 #else
462 int _init teql_init (void)
463 #endif
464 {
465     int err;
466
467     rtnl_lock ();
468
469     the_master.dev.priv = (void*)&the_master;
470     err = dev_alloc_name(&the_master.dev, "teql%d");
471     if (err < 0)
472         return err;
473     memcpy(the_master.qops.id, the_master.dev.name, IFNAMSIZ);
474     the_master.dev.init = teql_master_init;
475
476     err = register_netdevice (&the_master.dev);
477     if (err == 0) {
478         err = register_qdisc (&the_master.qops);
479         if (err)
480             unregister_netdevice (&the_master.dev);
481     }
482     rtnl_unlock ();
483     return err;
484 }
485
486 #ifdef MODULE
487 void cleanup_module(void)
488 {
489     rtnl_lock ();
490     unregister_qdisc (&the_master.qops);
491     unregister_netdevice (&the_master.dev);
492     rtnl_unlock ();
493 }
494 #endif
495

```

Figure 1119: TE net device register: `linux/net/sched/sch_teql.c`

```

24 atomic_t net_profile_active ;
25 struct timeval net_profile_adjust ;
26
27 NET_PROFILE_DEFINE(total);
28
29 struct net_profile_slot * net_profile_chain = &net_prof_total;
30
31 #ifdef _alpha_
32 _u32 alpha_lo;
33 long alpha_hi;
34
35 static void alpha_tick(unsigned long);
36
37 static struct timer_list alpha_timer =
38     { NULL, NULL, 0, 0L, alpha_tick };
39
40 void alpha_tick(unsigned long dummy)
41 {
42     struct timeval dummy_stamp;
43     net_profile_stamp(&dummy_stamp);
44     alpha_timer.expires = jiffies + 4*HZ;
45     add_timer(&alpha_timer);
46 }
47
48 #endif

```

Figure 1120: Profiling - Timer: `linux/net/core/profile.c`

```

142 struct iphdr whitehole_iph;
143 int whitehole_count;
144
145 static int whitehole_xmit(struct sk_buff *skb, struct net_device *dev)
146 {
147     struct net_device_stats *stats;
148
149     stats = (struct net_device_stats *)dev->priv;
150     stats->tx_packets++;
151     stats->tx_bytes+=skb->len;
152
153     dev_kfree_skb(skb);
154     return 0;
155 }
156
157 static void whitehole_inject(unsigned long);
158 int whitehole_init(struct net_device *dev);
159
160 static struct timer_list whitehole_timer =
161     { NULL, NULL, 0, 0L, whitehole_inject };
162
163 static struct net_device whitehole_dev = {
164     "whitehole", 0x0, 0x0, 0x0, 0x0 , 0, 0, 0, 0, 0, NULL, whitehole_init, };

```

Figure 1121: Profiling- White Hole Interface: `linux/net/core/profile.c`

```

166 static int whitehole_open(struct net_device *dev)
167 {
168     whitehole_count = 100000;
169     whitehole_timer.expires = jiffies + 5*HZ;
170     add_timer(&whitehole_timer);
171     return 0;
172 }
173
174 static int whitehole_close(struct net_device *dev)
175 {
176     del_timer(&whitehole_timer);
177     return 0;
178 }
179
180 static void whitehole_inject(unsigned long dummy)
181 {
182     struct net_device_stats *stats = (struct net_device_stats *)whitehole_dev.priv;
183     extern int netdev_dropping;
184
185     do {
186         struct iphdr *iph;
187         struct sk_buff *skb = alloc_skb(128, GFP_ATOMIC);
188         if (!skb)
189             break;
190         skb_reserve(skb, 32);
191         iph = (struct iphdr*)skb_put(skb, sizeof(*iph));
192         skb->mac.raw = ((u8*)iph) - 14;
193         memcpy(iph, &whitehole_iph, sizeof(*iph));
194         skb->protocol = htons(ETH_P_IP);
195         skb->dev = &whitehole_dev;
196         skb->pkt_type = PACKET_HOST;
197         stats->rx_packets++;
198         stats->rx_bytes += skb->len;
199         netif_rx(skb);
200         whitehole_count--;
201     } while (netdev_dropping == 0 && whitehole_count > 0);
202     if (whitehole_count > 0) {
203         whitehole_timer.expires = jiffies + 1;
204         add_timer(&whitehole_timer);
205     }
206 }

```

Figure 1122: Profiling- White Hole Interface: `linux/net/core/profile.c`

```

208 static struct net_device_stats *whitehole_get_stats(struct net_device *dev)
209 {
210     struct net_device_stats *stats = (struct net_device_stats *) dev->priv;
211     return stats;
212 }
213
214 int _init whitehole_init(struct net_device *dev)
215 {
216     dev->priv = kmalloc(sizeof(struct net_device_stats), GFP_KERNEL);
217     if (dev->priv == NULL)
218         return -ENOMEM;
219     memset(dev->priv, 0, sizeof(struct net_device_stats));
220     dev->get_stats = whitehole_get_stats;
221     dev->hard_start_xmit = whitehole_xmit;
222     dev->open = whitehole_open;
223     dev->stop = whitehole_close;
224     ether_setup(dev);
225     dev->tx_queue_len = 0;
226     dev->flags |= IFF_NOARP;
227     dev->flags &= ~(IFF_BROADCAST|IFF_MULTICAST);
228     dev->iflink = 0;
229     whitehole_iph.ihl = 5;
230     whitehole_iph.version = 4;
231     whitehole_iph.ttl = 2;
232     whitehole_iph.saddr = in_aton("193.233.7.21");
233     whitehole_iph.daddr = in_aton("193.233.7.10");
234     whitehole_iph.tot_len = htons(20);
235     whitehole_iph.check = ip_compute_csum((void *)&whitehole_iph, 20);
236     return 0;
237 }

```

Figure 1123: Profiling- White Hole - Stats and Init:
linux/net/core/profile.c

```

239 int net_profile_register (struct net_profile_slot * slot)
240 {
241     cli ();
242     slot->next = net_profile_chain;
243     net_profile_chain = slot;
244     sti ();
245     return 0;
246 }
247
248 int net_profile_unregister (struct net_profile_slot * slot)
249 {
250     struct net_profile_slot **sp, *s;
251
252     for (sp = &net_profile_chain; (s = *sp) != NULL; sp = &s->next) {
253         if (s == slot) {
254             cli ();
255             *sp = s->next;
256             sti ();
257             return 0;
258         }
259     }
260     return -ESRCH;
261 }
262
263
264 int _init net_profile_init (void)
265 {
266     int i;
267
268     #ifdef CONFIG_PROC_FS
269     create_proc_read_entry("net/profile", 0, 0, profile_read_proc, NULL);
270     #endif
271
272     register_netdevice (&whitehole_dev);
273
274     printk("Evaluating net profiler cost ...");
275     #ifdef _alpha_
276     alpha_tick(0);
277     #endif
278     for (i=0; i<1024; i++) {
279         NET_PROFILE_ENTER(total);
280         NET_PROFILE_LEAVE(total);
281     }
282     if (net_prof_total.accumulator.tv_sec) {
283         printk("Too high!\n");
284     } else {
285         net_profile_adjust.tv_usec = net_prof_total.accumulator.tv_usec >> 10;
286         printk("%ld units\n", net_profile_adjust.tv_usec);
287     }
288     net_prof_total.hits = 0;
289     net_profile_stamp(&net_prof_total.entered);
290     return 0;
291 }
292
293 #endif

```

Figure 1124: Profiling- White Hole - Register/Unregister:
linux/net/core/profile.c

```
1 /* net/sched/sch_atm.c - ATM VC selection "queueing discipline" */  
2  
3 /* Written 1998-2000 by Werner Almesberger, EPFL ICA */
```

Figure 1125: ATM Scheduling - Author: `linux/net/sched/sch_atm.c`

```
37
38
39 /*
40 * The ATM queuing discipline provides a framework for invoking classifiers
41 * (aka "filters"), which in turn select classes of this queuing discipline.
42 * Each class maps the flow(s) it is handling to a given VC. Multiple classes
43 * may share the same VC.
44 *
45 * When creating a class, VCs are specified by passing the number of the open
46 * socket descriptor by which the calling process references the VC. The kernel
47 * keeps the VC open at least until all classes using it are removed.
48 *
49 * In this file, most functions are named atm_tc_* to avoid confusion with all
50 * the atm_* in net/atm. This naming convention differs from what's used in the
51 * rest of net/sched.
52 *
53 * Known bugs:
54 * - sometimes messes up the IP stack
55 * - any manipulations besides the few operations described in the README, are
56 *   untested and likely to crash the system
```

Figure 1126: ATM Scheduling - Introductory Remarks:
linux/net/sched/sch_atm.c


```

59
60
61 #define PRIV(sch) ((struct atm_qdisc_data *) (sch)->data)
62 #define VCC2FLOW(vcc) ((struct atm_flow_data *) ((vcc)->user_back))
63
64
65 struct atm_flow_data {
66     struct Qdisc      *q;          /* FIFO, TBF, etc. */
67     struct tcf_proto  *filter_list ;
68     struct atm_vcc    *vcc;        /* VCC; NULL if VCC is closed */
69     void (*old_pop)(struct atm_vcc *vcc, struct sk_buff *skb); /* chaining */
70     struct atm_qdisc_data *parent; /* parent qdisc */
71     struct socket     *sock;       /* for closing */
72     u32               classid;     /* x:y type ID */
73     int               ref;         /* reference count */
74     struct tc_stats   stats;
75     struct atm_flow_data *next;
76     struct atm_flow_data *excess; /* flow for excess traffic;
77                                     NULL to set CLP instead */
78     int               hdr_len;
79     unsigned char     hdr[0];      /* header data; MUST BE LAST */
80 };
81
82 struct atm_qdisc_data {
83     struct atm_flow_data link;      /* unclassified skbs go here */
84     struct atm_flow_data *flows;   /* NB: "link" is also on this
85                                     list */

```

Figure 1127: ATM Scheduling - Data Structures: `linux/net/sched/sch_atm.c`

```

88
89
90 /* ----- Class/flow operations ----- */
91
92
93 static int find_flow(struct atm_qdisc_data *qdisc, struct atm_flow_data *flow)
94 {
95     struct atm_flow_data *walk;
96
97     DPRINTK("find_flow(qdisc=%p, flow=%p)\n", qdisc, flow);
98     for (walk = qdisc->flows; walk; walk = walk->next)
99         if (walk == flow) return 1;
100     DPRINTK("find_flow: not found\n");
101     return 0;
102 }
103
104
105 static __inline__ struct atm_flow_data *lookup_flow(struct Qdisc *sch,
106             u32 classid)
107 {
108     struct atm_flow_data *flow;
109
110     for (flow = PRIV(sch)->flows; flow; flow = flow->next)
111         if (flow->classid == classid) break;

```

Figure 1128: ATM Scheduling - Finding flows i: `linux/net/sched/sch_atm.c`

```

114
115
116 static int atm_tc_graft(struct Qdisc *sch,unsigned long arg,
117     struct Qdisc *new,struct Qdisc **old)
118 {
119     struct atm_qdisc_data *p = PRIV(sch);
120     struct atm_flow_data *flow = (struct atm_flow_data *) arg;
121
122     DPRINTK("atm_tc_graft(sch=%p,[qdisc=%p],flow=%p,new=%p,old=%p)\n",sch,
123         p,flow,new,old);
124     if (! find_flow (p,flow)) return -EINVAL;
125     if (! new) new = &noop_qdisc;
126     *old = xchg(&flow->q,new);
127     if (*old) qdisc_reset (*old);
128     return 0;
129 }
130
131
132 static struct Qdisc *atm_tc_leaf(struct Qdisc *sch,unsigned long cl)
133 {
134     struct atm_flow_data *flow = (struct atm_flow_data *) cl;
135
136     DPRINTK("atm_tc_leaf(sch=%p,flow=%p)\n",sch,flow);

```

Figure 1129: ATM Scheduling - Trees of Flows: `linux/net/sched/sch_atm.c`

```

139
140
141 static unsigned long atm_tc_get(struct Qdisc *sch,u32 classid)
142 {
143     struct atm_qdisc_data *p __attribute__((unused)) = PRIV(sch);
144     struct atm_flow_data *flow;
145
146     DPRINTK("atm_tc_get(sch=%p,[qdisc=%p],classid=%x)\n",sch,p,classid);
147     flow = lookup_flow(sch, classid);
148     if (flow) flow->ref++;
149     DPRINTK("atm_tc_get: flow=%p\n",flow);
150     return (unsigned long) flow;
151 }
152
153
154 static unsigned long atm_tc_bind_filter (struct Qdisc *sch,
155     unsigned long parent, u32 classid)
156 {

```

Figure 1130: ATM Scheduling - more Finding Flows:
linux/net/sched/sch_atm.c

```
159
160
161 static void destroy_filters (struct atm_flow_data *flow)
162 {
163     struct tcf_proto * filter ;
164
165     while (( filter = flow->filter_list )) {
166         DPRINTK("destroy_filters: destroying filter %p\n",filter);
167         flow->filter_list = filter ->next;
168         filter ->ops->destroy(filter);
169     }
```

Figure 1131: ATM Scheduling - destroy filters: `linux/net/sched/sch_atm.c`

```

171
172
173 /*
174 * atm_tc_put handles all destructions, including the ones that are explicitly
175 * requested (atm_tc_destroy, etc.). The assumption here is that we never drop
176 * anything that still seems to be in use.
177 */
178
179 static void atm_tc_put(struct Qdisc *sch, unsigned long cl)
180 {
181     struct atm_qdisc_data *p = PRIV(sch);
182     struct atm_flow_data *flow = (struct atm_flow_data *) cl;
183     struct atm_flow_data **prev;
184
185     DPRINTK("atm_tc_put(sch=%p,[qdisc=%p],flow=%p)\n",sch,p,flow);
186     if (--flow->ref) return;
187     DPRINTK("atm_tc_put: destroying\n");
188     for (prev = &p->flows; *prev; prev = &(*prev)->next)
189         if (*prev == flow) break;
190     if (!*prev) {
191         printk(KERN_CRIT "atm_tc_put: class=%p not found\n",flow);
192         return;
193     }
194     *prev = flow->next;
195     DPRINTK("atm_tc_put: qdisc=%p\n",flow->q);
196     qdisc_destroy(flow->q);
197     destroy_filters(flow);
198     if (flow->sock) {
199         DPRINTK("atm_tc_put: f_count=%d\n",
200             file_count(flow->sock->file));
201         flow->vcc->pop = flow->old_pop;
202         sockfd_put(flow->sock);
203     }
204     if (flow->excess) atm_tc_put(sch,(unsigned long) flow->excess);
205     if (flow != &p->link) kfree(flow);
206     /*
207     * If flow == &p->link, the qdisc no longer works at this point and
208     * needs to be removed. (By the caller of atm_tc_put.)

```

Figure 1132: ATM Scheduling - destroy all filters: `linux/net/sched/sch_atm.c`

```
211
212
213 static void sch_atm_pop(struct atm_vcc *vcc, struct sk_buff *skb)
214 {
215     struct atm_qdisc_data *p = VCC2FLOW(vcc)->parent;
216
217     D2PRINTK("sch_atm_pop(vcc=%p,skb=%p,[qdisc=%p])\n",vcc,skb,p);
218     VCC2FLOW(vcc)->old_pop(vcc,skb);
```

Figure 1133: ATM Scheduling - pop a VC: `linux/net/sched/sch_atm.c`

```

221
222
223 static int atm_tc_change(struct Qdisc *sch, u32 classid, u32 parent,
224     struct rtattr **tca, unsigned long *arg)
225 {
226     struct atm_qdisc_data *p = PRIV(sch);
227     struct atm_flow_data *flow = (struct atm_flow_data *) *arg;
228     struct atm_flow_data *excess = NULL;
229     struct rtattr *opt = tca[TCA_OPTIONS-1];
230     struct rtattr *tb[TCA_ATM_MAX];
231     struct socket *sock;
232     int fd, error, hdr_len;
233     void *hdr;
234
235     DPRINTK("atm_tc_change(sch_%p,[qdisc_%p],classid_%x,parent_%x,"
236         "flow_%p,opt_%p)\n",sch,p,classid,parent,flow,opt);
237     /*
238      * The concept of parents doesn't apply for this qdisc.
239      */
240     if (parent && parent != TC_H_ROOT && parent != sch->handle)
241         return -EINVAL;
242     /*
243      * ATM classes cannot be changed. In order to change properties of the
244      * ATM connection, that socket needs to be modified directly (via the
245      * native ATM API. In order to send a flow to a different VC, the old
246      * class needs to be removed and a new one added. (This may be changed
247      * later.)
248      */
249     if (flow) return -EBUSY;
250     if (opt == NULL || rtattr_parse(tb,TCA_ATM_MAX,RTA_DATA(opt),
251         RTA_PAYLOAD(opt))) return -EINVAL;
252     if (!tb[TCA_ATM_FD-1] || RTA_PAYLOAD(tb[TCA_ATM_FD-1]) < sizeof(fd))
253         return -EINVAL;

```

Figure 1134: ATM Scheduling - Change ATM TC Parameters:
linux/net/sched/sch_atm.c


```

255     DPRINTK("atm_tc_change:fd,%d\n",fd);
256     if (tb[TCA_ATM_HDR-1]) {
257         hdr_len = RTA_PAYLOAD(tb[TCA_ATM_HDR-1]);
258         hdr = RTA_DATA(tb[TCA_ATM_HDR-1]);
259     }
260     else {
261         hdr_len = RFC1483LLC_LEN;
262         hdr = NULL; /* default LLC/SNAP for IP */
263     }
264     if (!tb[TCA_ATM_EXCESS-1]) excess = NULL;
265     else {
266         if (RTA_PAYLOAD(tb[TCA_ATM_EXCESS-1]) != sizeof(u32))
267             return -EINVAL;
268         excess = (struct atm_flow_data *) atm_tc_get(sch,
269             *(u32 *) RTA_DATA(tb[TCA_ATM_EXCESS-1]));
270         if (!excess) return -ENOENT;
271     }
272     DPRINTK("atm_tc_change:type,%d,payload,%d,hdr_len,%d\n",
273         opt->rta_type,RTA_PAYLOAD(opt),hdr_len);
274     if (!(sock = sockfd_lookup(fd,&error))) return error; /* f_count++ */
275     DPRINTK("atm_tc_change:f_count,%d\n",file_count(sock->file));
276     if (sock->ops->family != PF_ATMSVC && sock->ops->family !=
277         PF_ATMPVC) {
278         error = -EPROTOTYPE;
279         goto err_out;
280     }
281     /* @@@ should check if the socket is really operational or we'll crash
282     on vcc->send */
283     if (classid) {
284         if (TC_H_MAJ(classid ^ sch->handle)) {
285             DPRINTK("atm_tc_change:classid_mismatch\n");
286             error = -EINVAL;
287             goto err_out;
288         }
289         if (find_flow(p,flow)) {
290             error = -EEXIST;
291             goto err_out;

```

Figure 1135: ATM Scheduling - Change ATM TC Parameters ii:
linux/net/sched/sch_atm.c

```

291     }
292   }
293   else {
294     int i;
295     unsigned long cl;
296
297     for (i = 1; i < 0x8000; i++) {
298       classid = TC_H_MAKE(sch->handle,0x8000 | i);
299       if (!(cl = atm_tc_get(sch, classid))) break;
300       atm_tc_put(sch, cl);
301     }
302   }
303   DPRINTK("atm_tc_change: new id %x\n", classid);
304   flow = kmalloc(sizeof(struct atm_flow_data)+hdr_len, GFP_KERNEL);
305   DPRINTK("atm_tc_change: flow %p\n", flow);
306   if (!flow) {
307     error = -ENOBUFS;
308     goto err_out;
309   }
310   memset(flow, 0, sizeof(*flow));
311   flow->filter_list = NULL;
312   if (!(flow->q = qdisc_create_dflt(sch->dev, &pfifo_qdisc_ops)))
313     flow->q = &noop_qdisc;
314   DPRINTK("atm_tc_change: qdisc %p\n", flow->q);
315   flow->sock = sock;
316   flow->vcc = ATM_SD(sock); /* speedup */
317   flow->vcc->user_back = flow;
318   DPRINTK("atm_tc_change: vcc %p\n", flow->vcc);
319   flow->old_pop = flow->vcc->pop;
320   flow->parent = p;
321   flow->vcc->pop = sch_atm_pop;
322   flow->classid = classid;
323   flow->ref = 1;
324   flow->excess = excess;
325   flow->next = p->link.next;
326   p->link.next = flow;
327   flow->hdr_len = hdr_len;
328   if (hdr) memcpy(flow->hdr, hdr, hdr_len);
329   else {
330     memcpy(flow->hdr, llc_oui, sizeof(llc_oui));
331     ((u16 *) flow->hdr)[3] = htons(ETH_P_IP);
332   }
333   *arg = (unsigned long) flow;
334   return 0;
335 err_out:
336   if (excess) atm_tc_put(sch, (unsigned long) excess);
337   sockfd_put(sock);

```

Figure 1136: ATM Scheduling - Change ATM TC Parameters ii:
linux/net/sched/sch_atm.c

```

340
341
342 static int atm_tc_delete(struct Qdisc *sch, unsigned long arg)
343 {
344     struct atm_qdisc_data *p = PRIV(sch);
345     struct atm_flow_data *flow = (struct atm_flow_data *) arg;
346
347     DPRINTK("atm_tc_delete(sch=%p,[qdisc=%p],flow=%p)\n",sch,p,flow);
348     if (! find_flow (PRIV(sch),flow)) return -EINVAL;
349     if (flow->filter_list || flow == &p->link) return -EBUSY;
350     /*
351      * Reference count must be 2: one for "keepalive" (set at class
352      * creation), and one for the reference held when calling delete.
353      */
354     if (flow->ref < 2) {
355         printk(KERN_ERR "atm_tc_delete: flow->ref==%d\n",flow->ref);
356         return -EINVAL;
357     }
358     if (flow->ref > 2) return -EBUSY; /* catch references via excess, etc.*/
359     atm_tc_put(sch,arg);

```

Figure 1137: ATM Scheduling - Delete TC: linux/net/sched/sch_atm.c

```

362
363
364 static void atm_tc_walk(struct Qdisc *sch, struct qdisc_walker *walker)
365 {
366     struct atm_qdisc_data *p = PRIV(sch);
367     struct atm_flow_data *flow;
368
369     DPRINTK("atm_tc_walk(sch=%p,[qdisc=%p],walker=%p)\n",sch,p,walker);
370     if (walker->stop) return;
371     for (flow = p->flows; flow; flow = flow->next) {
372         if (walker->count >= walker->skip)
373             if (walker->fn(sch,(unsigned long) flow,walker) < 0) {
374                 walker->stop = 1;
375                 break;
376             }
377         walker->count++;

```

Figure 1138: ATM Scheduling - Walk TC: `linux/net/sched/sch_atm.c`

```

380
381
382 static struct tcf_proto ** atm_tc_find_tcf(struct Qdisc *sch,unsigned long cl)
383 {
384     struct atm_qdisc_data *p = PRIV(sch);
385     struct atm_flow_data *flow = (struct atm_flow_data *) cl;
386
387     DPRINTK("atm_tc_find_tcf(sch=%p,[qdisc=%p],flow=%p)\n",sch,p,flow);

```

Figure 1139: ATM Scheduling - Find TCF: `linux/net/sched/sch_atm.c`

```

393
394
395 static int atm_tc_enqueue(struct sk_buff *skb, struct Qdisc *sch)
396 {
397     struct atm_qdisc_data *p = PRIV(sch);
398     struct atm_flow_data *flow = NULL; /* @@@ */
399     struct tcf_result res;
400     int result;
401     int ret = NET_XMIT_POLICED;
402
403     D2PRINTK("atm_tc_enqueue(skb,%p,sch,%p,[qdisc,%p])\n",skb,sch,p);
404     result = TC_POLICE_OK; /* be nice to gcc */
405     if (TC_H_MAJ(skb->priority) != sch->handle ||
406         !(flow = (struct atm_flow_data *) atm_tc_get(sch,skb->priority)))
407         for (flow = p->flows; flow; flow = flow->next)
408             if (flow->filter_list) {
409                 result = tc_classify(skb,flow->filter_list,
410                                     &res);
411                 if (result < 0) continue;
412                 flow = (struct atm_flow_data *) res.class;
413                 if (!flow) flow = lookup_flow(sch,res.classid);
414                 break;

```

Figure 1140: ATM Scheduling - Enqueue i: `linux/net/sched/sch_atm.c`

```

415     }
416     if (!flow) flow = &p->link;
417     else {
418         if (flow->vcc)
419             ATM_SKB(skb)->atm_options = flow->vcc->atm_options;
420             /*@@@ looks good ... but it's not supposed to work :-*/)
421 #ifdef CONFIG_NET_CLS_POLICE
422     switch (result) {
423     case TC_POLICE_SHOT:
424         kfree_skb(skb);
425         break;
426     case TC_POLICE_RECLASSIFY:
427         if (flow->excess) flow = flow->excess;
428         else {
429             ATM_SKB(skb)->atm_options |=
430                 ATM_ATMOPT_CLP;
431             break;
432         }
433         /* fall through */
434     case TC_POLICE_OK:
435         /* fall through */
436     default:
437         break;
438     }

```

Figure 1141: ATM Scheduling - Enqueue ii: `linux/net/sched/sch_atm.c`

```

439 #endif
440     }
441     if (
442 #ifdef CONFIG_NET_CLS_POLICE
443         result == TC_POLICE_SHOT ||
444 #endif
445         (ret = flow->q->enqueue(skb,flow->q)) != 0) {
446         sch->stats.drops++;
447         if (flow) flow->stats.drops++;
448         return ret;
449     }
450     sch->stats.bytes += skb->len;
451     sch->stats.packets++;
452     flow->stats.bytes += skb->len;
453     flow->stats.packets++;
454     /*
455     * Okay, this may seem weird. We pretend we've dropped the packet if
456     * it goes via ATM. The reason for this is that the outer qdisc
457     * expects to be able to q->dequeue the packet later on if we return
458     * success at this place. Also, sch->q.qdisc needs to reflect whether
459     * there is a packet eligible for dequeuing or not. Note that the
460     * statistics of the outer qdisc are necessarily wrong because of all
461     * this. There's currently no correct solution for this.
462     */
463     if (flow == &p->link) {
464         sch->q.qlen++;
465         return 0;
466     }
467     tasklet_schedule(&p->task);

```

Figure 1142: ATM Scheduling - Enqueue iii: linux/net/sched/sch_atm.c


```

470
471
472 /*
473 * Dequeue packets and send them over ATM. Note that we quite deliberately
474 * avoid checking net_device's flow control here, simply because sch_atm
475 * uses its own channels, which have nothing to do with any CLIP/LANE/or
476 * non-ATM interfaces.
477 */
478
479
480 static void sch_atm_dequeue(unsigned long data)
481 {
482     struct Qdisc *sch = (struct Qdisc *) data;
483     struct atm_qdisc_data *p = PRIV(sch);
484     struct atm_flow_data *flow;
485     struct sk_buff *skb;
486
487     D2PRINTK("sch_atm_dequeue(sch=%p,[qdisc=%p])\n",sch,p);
488     for (flow = p->link.next; flow; flow = flow->next)
489         /*
490          * If traffic is properly shaped, this won't generate nasty
491          * little bursts. Otherwise, it may ... (but that's okay)
492          */
493         while ((skb = flow->q->dequeue(flow->q)) {
494             if (!atm_may_send(flow->vcc,skb->truesize)) {
495                 (void) flow->q->ops->requeue(skb,flow->q);
496                 break;
497             }
498             D2PRINTK("atm_tc_dequeue: sending on class %p\n",flow);
499             /* remove any LL header somebody else has attached */
500             skb_pull(skb,(char *) skb->nh.iph-(char *) skb->data);
501             if (skb_headroom(skb) < flow->hdr_len) {
502                 struct sk_buff *new;
503
504                 new = skb_realloc_headroom(skb,flow->hdr_len);
505                 dev_kfree_skb(skb);
506                 if (!new) continue;
507                 skb = new;
508             }
509             D2PRINTK("sch_atm_dequeue: ip=%p,data=%p\n",
510                 skb->nh.iph,skb->data);
511             ATM_SKB(skb)->vcc = flow->vcc;
512             memcpy(skb_push(skb,flow->hdr_len),flow->hdr,
513                 flow->hdr_len);
514             atomic_add(skb->truesize,&flow->vcc->tx_inuse);
515             ATM_SKB(skb)->iovcnt = 0;
516             /* atm.atm_options are already set by atm_tc_enqueue */
517             (void) flow->vcc->send(flow->vcc,skb);

```

Figure 1143: ATM Scheduling - Dequeue: linux/net/sched/sch_atm.c

```
520
521
522 static struct sk_buff *atm_tc_dequeue(struct Qdisc *sch)
523 {
524     struct atm_qdisc_data *p = PRIV(sch);
525     struct sk_buff *skb;
526
527     D2PRINTK("atm_tc_dequeue(sch=%p,[qdisc=%p])\n",sch,p);
528     tasklet_schedule(&p->task);
529     skb = p->link.q->dequeue(p->link.q);
530     if (skb) sch->q.qlen--;
```

Figure 1144: ATM Scheduling - TC Dequeue: `linux/net/sched/sch_atm.c`

```

532 }
533
534
535 static int atm_tc_requeue(struct sk_buff *skb, struct Qdisc *sch)
536 {
537     struct atm_qdisc_data *p = PRIV(sch);
538     int ret;
539
540     D2PRINTK("atm_tc_requeue(skb_%p,sch_%p,[qdisc_%p])\n",skb,sch,p);
541     ret = p->link.q->ops->requeue(skb,p->link.q);
542     if (!ret) sch->q.qlen++;
543     else {
544         sch->stats.drops++;
545         p->link.stats.drops++;
546     }

```

Figure 1145: ATM Scheduling - TC Requeue: `linux/net/sched/sch_atm.c`

```

549
550
551 static int atm_tc_drop(struct Qdisc *sch)
552 {
553     struct atm_qdisc_data *p = PRIV(sch);
554     struct atm_flow_data *flow;
555
556     DPRINTK("atm_tc_drop(sch=%p,[qdisc=%p])\n",sch,p);
557     for (flow = p->flows; flow; flow = flow->next)
558         if (flow->q->ops->drop && flow->q->ops->drop(flow->q))
559             return 1;

```

Figure 1146: ATM Scheduling - TC Drop: `linux/net/sched/sch_atm.c`

```
714
715
716 #ifdef MODULE
717 int init_module(void)
718 {
719     return register_qdisc (&atm_qdisc_ops);
720 }
721
722
723 void cleanup_module(void)
724 {
725     unregister_qdisc (&atm_qdisc_ops);
```

Figure 1147: ATM Scheduling - Initialise: `linux/net/sched/sch_atm.c`

```
37 void *BODY;
38 static struct qdisc_util * qdisc_list ;
39 static struct filter_util * filter_list ;
40
41 static int print_noqopt(struct qdisc_util *qu, FILE *f, struct rtattr *opt)
42 {
```

Figure 1148: TC - main: `tc/tc.c`

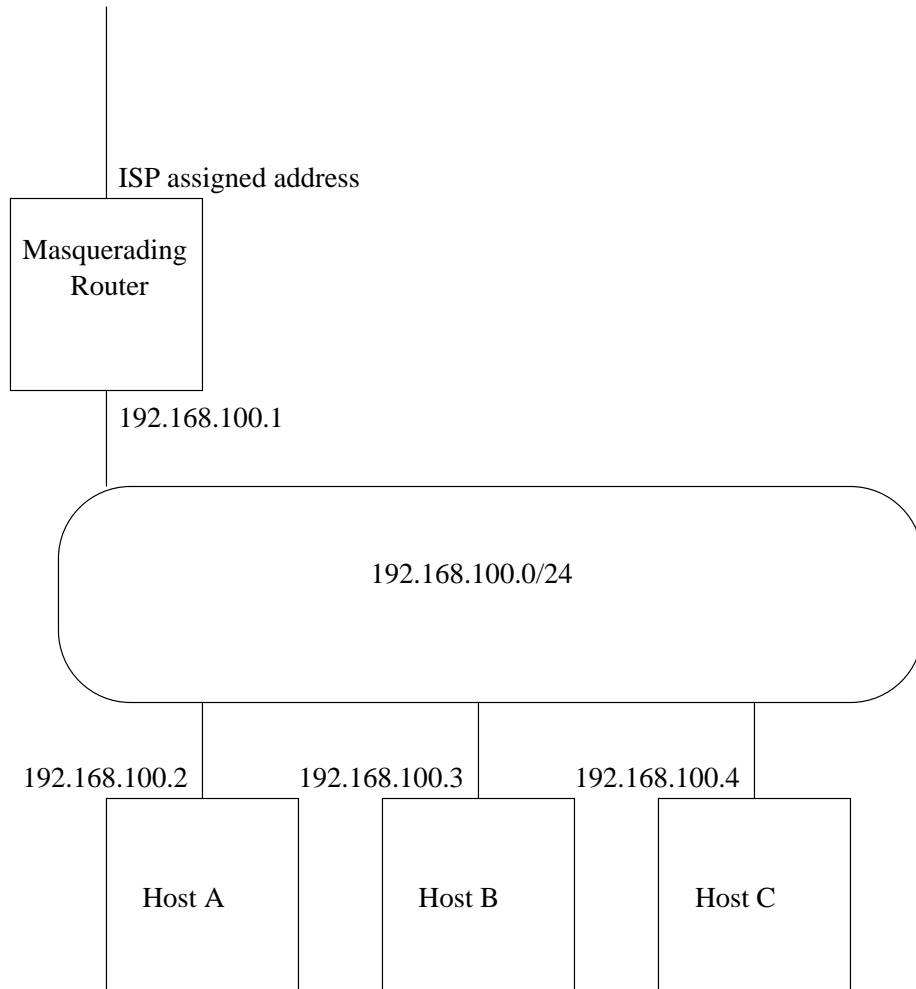


Figure 1149: Masquerading Router

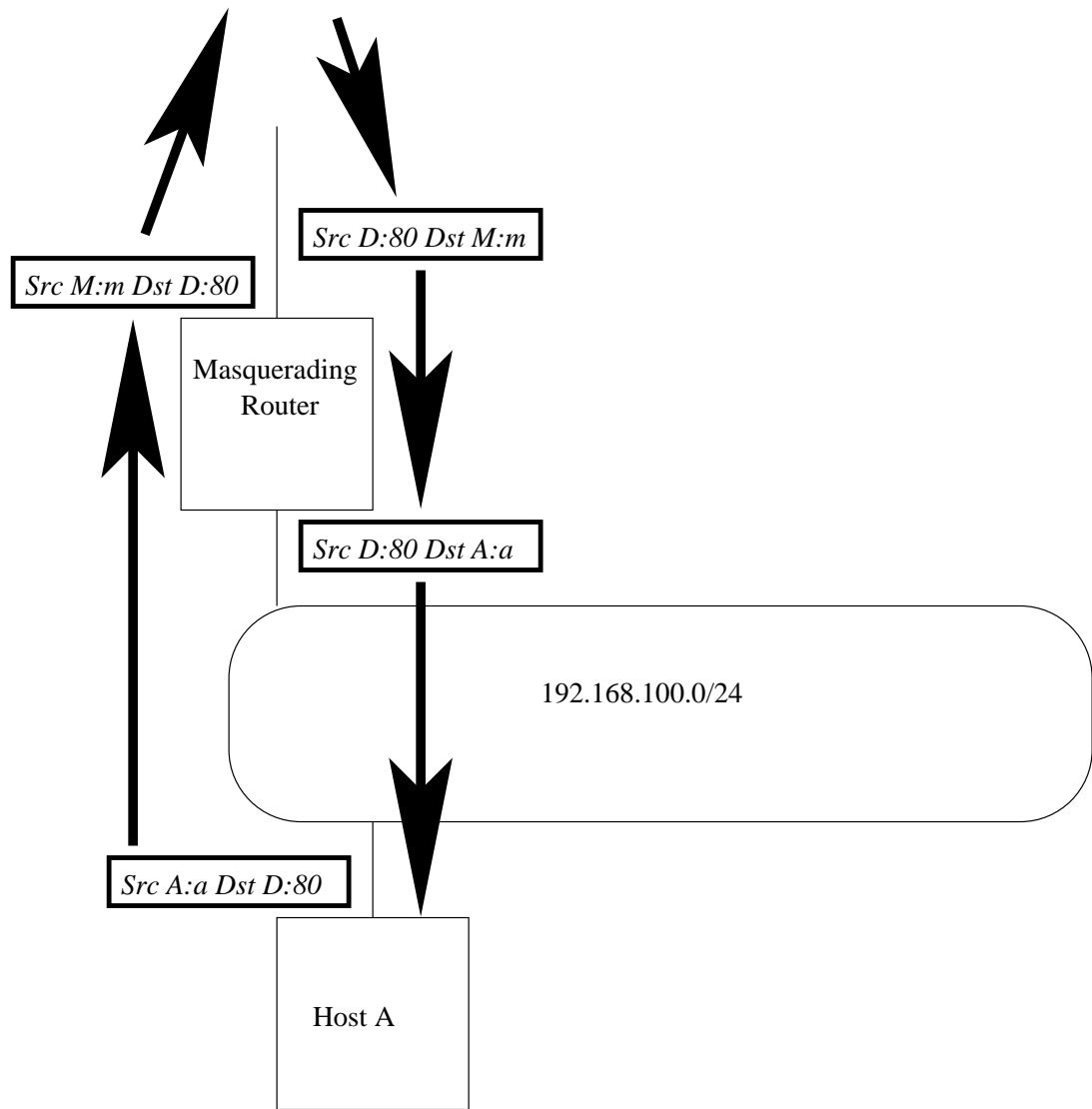


Figure 1150: Masquerading Example

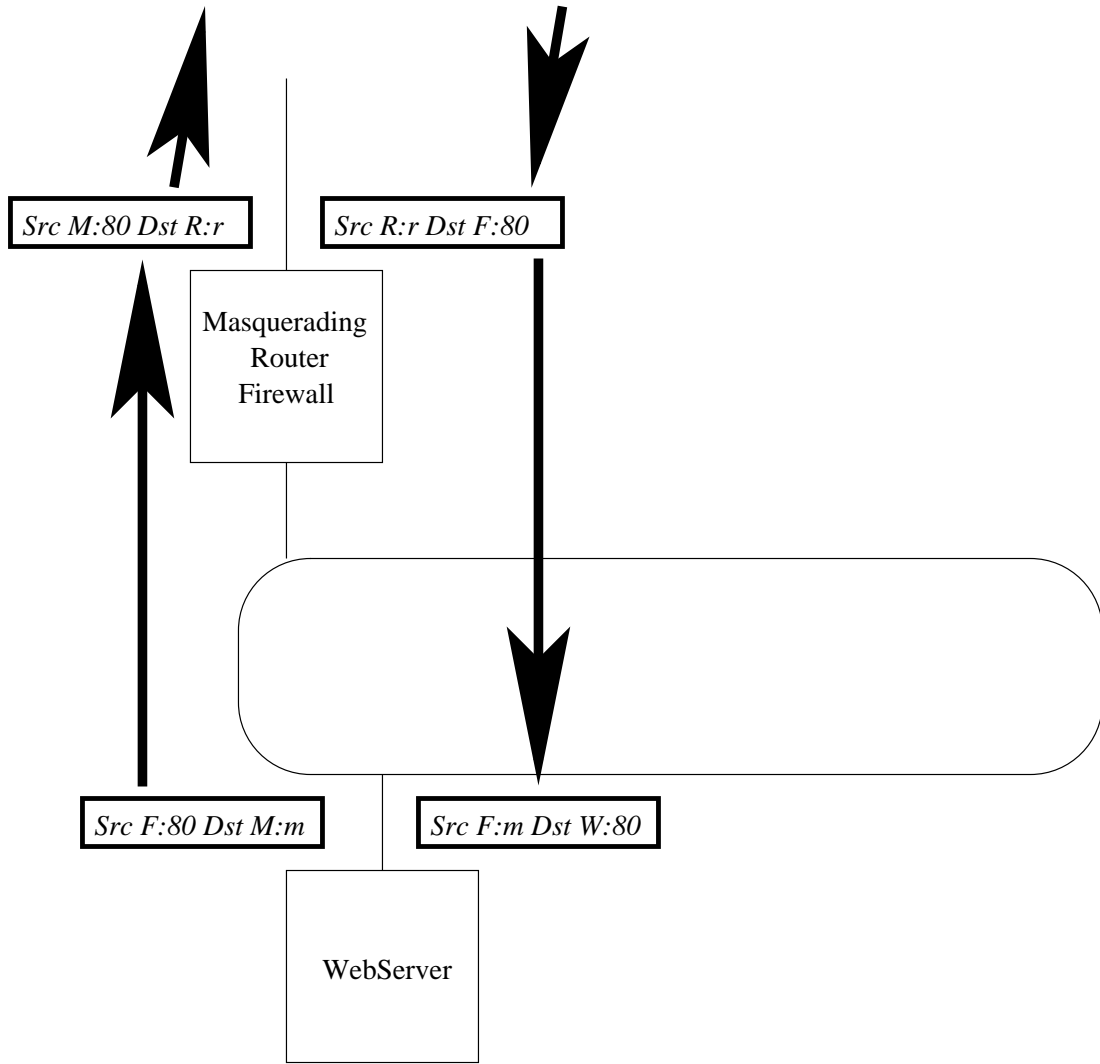


Figure 1151: Port forwarding with a firewall

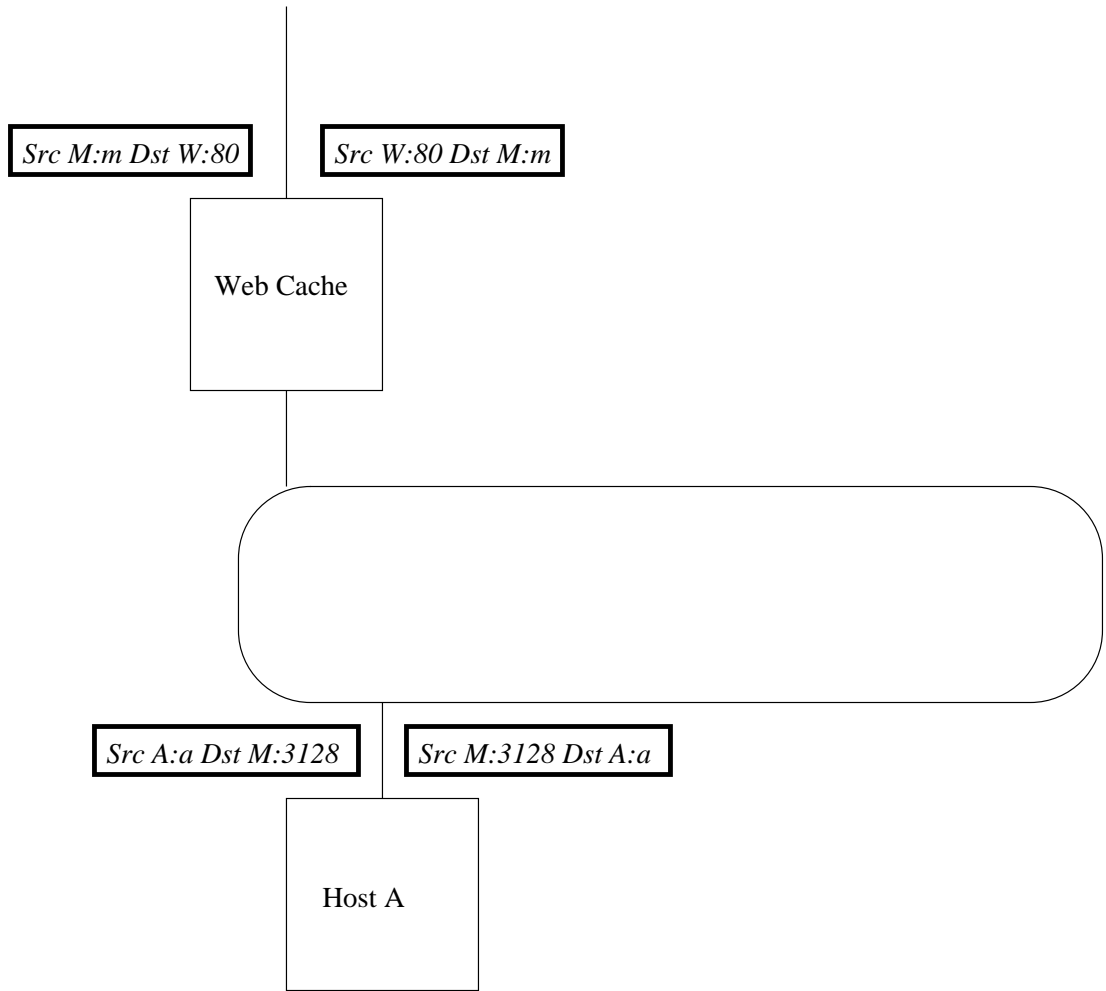


Figure 1152: A proxy cache

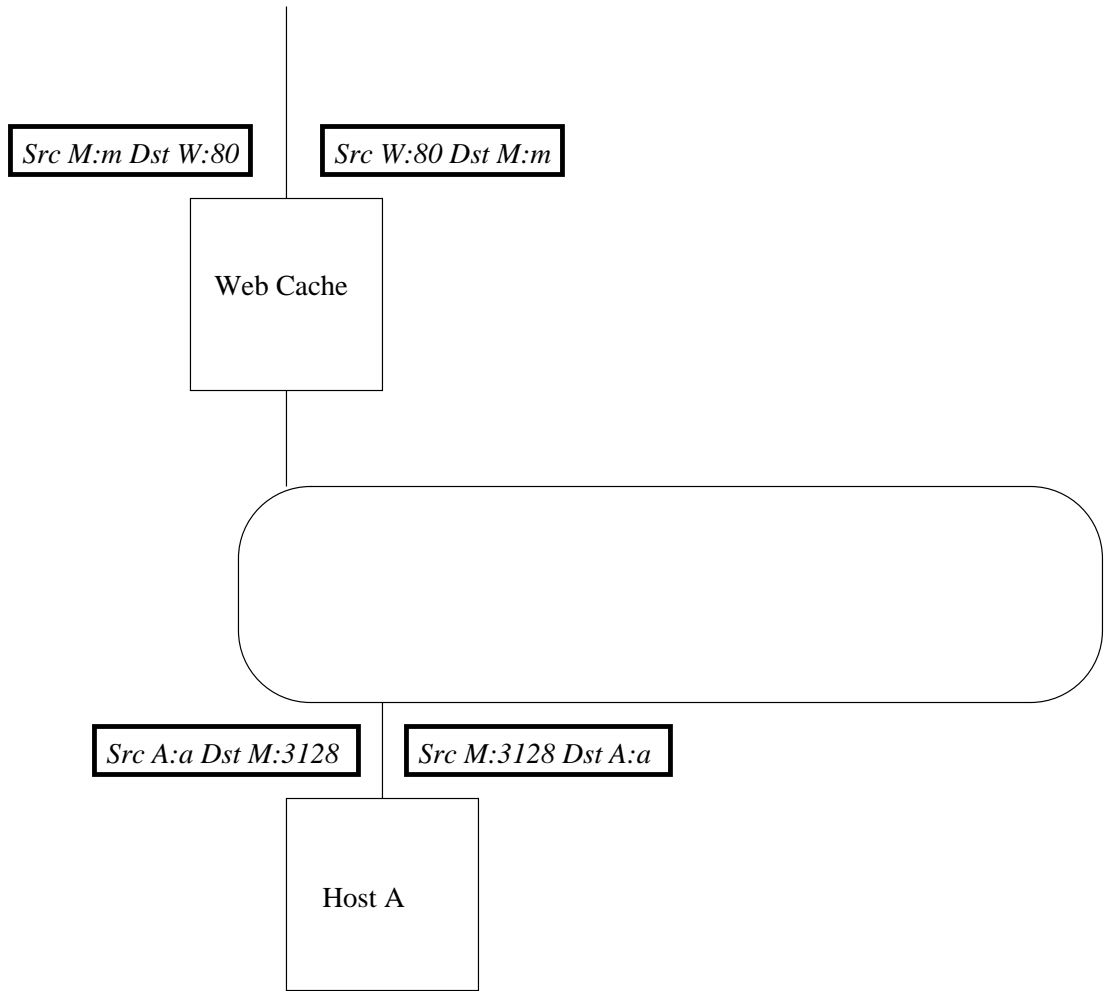
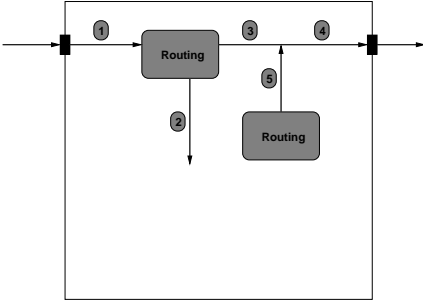


Figure 1153: A transparent proxy cache



List of Tables

Function	Intserv	Diffserv
signalling	from application	network management, application
granularity	flow	flow, source, site, aggregate
mechanism	destination address, protocol, port	packet class
scope	end-to-end	edge-to-edge

Table 1: Integrated Versus Differentiated Services

single processor requirement	non SMP implementation
spin lock irq	cli()
spin unlock irq	sti()
spin lock irqsave	save flags, cli()
spin unlock irqrestore	restore flags, sti()
read lock irq	cli()
read unlock irq	sti()
read lock irqsave	save flags, cli()
read unlock irqrestore	restore flags, sti()
write lock irq	cli()
write unlock irq	sti()
write lock irqsave	(save flags cli)
write lock irqrestore	(sti)

Table 2: Kernel Critical Region Serialisation

spin lock requirement SMP implementation use atomic test on one cpu only if clear, proceed otherwise loop, reading bit til clear and then loop back to atomic test and set.

Table 3: Spin Lock Outline

lock requirement	SMP implementation
read lock irq (rwl)	cli(), read lock(rwl)
read unlock irq (rwl)	read unlock(rwl), sti()
etc	etc

Table 4: Kernel spin read/write locks

<code>arch/</code>	processor hardware model platform dependent code
<code>drivers/</code>	device specific code
<code>fs/</code>	file systems code
<code>include/</code>	header files
<code>init/</code>	kernel startup code
<code>kernel/</code>	core code for processes, modules, scheduling
<code>lib/</code>	general purpose kernel support functions
<code>mm/</code>	memory management code
<code>net/</code>	the networking code

Table 5: Source Code Directory Tree Organisation

core/	common code for device access, filtering, netlink, socket glue etc
ipv4/	IP version 4 code, TCP code, UDP code, ICMP and IGMP and forwarding table code.
ipv6/	same as IPv4 but for IPv6!
sched/	packet scheduling for output link resource management, IP or non IP
packet/	raw packet access protocol family. below or non IP
khttpd/	misguided attempt to put the HTTP application in kernel
sunrpc/	implementation of Sun's Remote Procedure Call (for NFS)
wanrouter/	support for multi-protocol router
netlink/	protocol management protocol socket code
ethernet/	generic ethernet code (specific ethernet cards in ../dev/
802/	generic IEEE 802 link layer protocol code
atm/	generic Asynchronous Transfer Mode network/link code, including MPOA and IP specifics
ax25/	amateur packet radio X.25 link protocol code, below IP
x25/	wide area x.25 protocol "network" code, below or non IP
rose/	remote operations service code - layer on x.25, non IP
lapb/	Link Access procedure, balanced link layer protocol code, below IP
netrom/	netrom IP device code
bridge/	IEEE 802.1d spec bridge implementation below IP Level
ipx/	Novell's IPX protocol (not IP!)
unix/	Unix Domain Socket code - see chapter four. non IP
irda/	Infra-red digital access - that IR non IP device
decnet/	DECNET (old digital equipment corporation non IP protocol suite)
econet/	ECONET - cheap and cheerful (non IP) network
appletalk/	for talking to MACs using native (non IP) protocols

Table 6: Source Code Network Sub-Directory Tree Organisation

<code>linux/net/ipv4/netfilter/</code>	contains the IPv4 specific filter code - see chapter ten
--	--

Table 7: IPv4 Sub-Directory Tree Organisation

Protocol Family	Use
PF_UNIX,PF_LOCAL	Local communication
PF_INET	IPv4 Internet (discussed in chapter six)
PF_INET6	IPv6 Internet (discussed in chapter six)
PF_IPX	IPX - Novell
PF_NETLINK	Kernel user interface device (discussed in chapter seven and eight)
PF_X25	ITU-T X.25 / ISO-8208 protocol
PF_AX25	Amateur radio AX.25 protocol
PF_ATMPVC	Access to raw ATM PVCs
PF_APPLETALK	Appletalk
PF_PACKET	Low level packet interface (deprecated)

Table 8: Linux socket protocol families implemented

Type	Semantic
SOCK_STREAM	Reliable, (connection oriented) ordered, duplex byte stream delivery.
SOCK_DGRAM	Single Shot (connectionless, unreliable) packet.
SOCK_SEQPACKET	Packetized delivery, otherwise similar to stream
SOCK_RAW	Low level protocol access
SOCK_RDM	As with sequence packet, but not ordered
SOCK_PACKET	deprecated (like raw but different!)

Table 9: Linux socket protocol semantics

Function	Use
bind	associated a socket descriptor with a local name or address
connect	associate a socket descriptor with the <i>far end</i> of the protocol (has side effect of setting up connections for connection oriented protocols!)
listen	wait for a connection to be setup from a (any) <i>far end</i>
accept	agree to a connection setup requested from a now specific far end
shutdown	tear down the protocol state on this socket
getsockname	map from socket descriptor to the local name (address) bound to it
setsockopt	request some non default socket parameter settings
getsockopt	find out the current status of some socket parameters
ioctl	general purpose workaround system call on descriptors
send	send a buffer full of data on a socket
recv	receive up to a buffer full of data on a socket
read/readv	read some data into buffer(s)
write/writev	send some data from buffer(s)
select	check I/O status on a socket descriptor
socketpair	see pipes (equivalent)
getprotoent	see <code>/etc/protocols</code>

Table 10: Other socket API calls

Socket Option	Purpose
SO_KEEPALIVE	sets the protocol “pinging” the far end.
SO_OOBINLINE	moves <i>out-of-band</i> data passing into the in-band data stream
SO_RCVLOWAT and SO_SNDLOWAT	sets a receive (or send) low watermark on the before the socket layer
SO_RCVTIMEO and SO_SNDTIMEO	alter the timeout before reporting an error - fixed per protocol in linux
SO_BSDCOMPAT	for bug emulation
SO_PASSCRED	PF_UNIX domain specific
SO_PEERCREC	PF_UNIX domain specific
SO_BINDTODEVICE	on a multihomed machine, allows one to limit the interface packets
SO_DEBUG	For processes with adequate (root) rights, can debug
SO_REUSEADDR	in PF_INET, allows more than one process to <code>bind()</code> to given port
SO_TYPE	report (cannot set) the protocol type (as above) of the socket
SO_DONTROUTE	only allow local communication
SO_BROADCAST	set the broadcast address;
SO_SNDBUF	configures the maximum socket transmit side buffer size
SO_RCVBUF	configures the maximum socket receive side buffer space
SO_LINGER	says how long <code>close()</code> blocks for waiting to process queued messages
SO_PRIORITY	sets up priority associated with socket - see chapter nine for details!
SO_ERROR	return details of error
SO_ATTACH_FILTER	these are CONFIG_FILTER socket options - see chapter ten (libpcap)
SO_DETACH_FILTER	ditto

Table 11: Other socket API calls

TCP sysctl	Purpose
tcp_window_scaling	Enable RFC1323 TCP window scaling
tcp_sack	Enable RFC2018 TCP SACK
tcp_timestamps	Enable RFC1323 TCP timestamps
tcp_fin_timeout	seconds before FIN, before forcing socket closed
tcp_keepalive_probes	Maximum TCP keep-alive probes
tcp_keepalive_time	seconds before sending probe on idle connection
tcp_max_ka_probes	How many keep-alive probes are sent
tcp_stdurg	Enable the strict RFC793 interpretation of the TCP urgent-pointer field.
tcp_syncookies	Enable TCP syncookies. The kernel must be compiled with CONFIG_SYN_COOKIES
tcp_max_syn_backlog	Length of the per-socket backlog queue.
tcp_retries1	max count of SYN ACK retransmit
tcp_retries2	max count for retransmit data packet in established connection
tcp_syn_retries	max count for SYN (re) tries
tcp_retrans_collapse	Try to send full-sized packets during retransmit.

Table 12: TCP sysctls

TCP specific setsockopt	Purpose
TCP_NODELAY	disable the Nagle algorithm packets are always sent as soon as possible
TCP_MAXSEG	configure the MSS for a connection
TCP_CORK	Toggle sending of partial frames. Almost opposite of Nagle

Table 13: TCP setsockopt options

TCP specific setsockopt	Purpose
FIONREAD	Returns the amount of queued unread data in the receive buffer
SIOCATMARK	Returns true when the all urgent data has been already received
TIOCOUTQ	Returns the amount of unsent data in the socket send queue

Table 14: TCP ioctl options

W_m ; Maximum advertised receive window
T_0 ; Initial Timeout Value
RTT ; The Round Trip Time
b ; the number of packets acknowledged by 1 ACK
p ; the mean packet loss probability
B ; the throughput achieved by a TCP flow

Table 15: Terms in the Padhye TCP Equation

IPv4 Multicast Address ; 224.20.5.1 = 1110 0000 0001 0100 0001 0100 0000 0001
IANA MAC Address Prefix ; = 0000 0001 0000 0000 0101 1110 0— —
Final Ethernet multicast address ; 0000 0001 0000 0000 0101 1110 0100 0001 0100 0000 0001

Table 16: multicast addressing

d	worst case delay
b	token bucket size
g	allocated rate
N	hopcount
P_F	this flow maximum packet size
P_N	networks maximum packet size
r	link speed

Table 17: Terms in the Parekh TCP Equation

Interactive ; Telnet, X-Windows
Interactive bulk ; FTP, HTTP
Asynchronous ; E-mail, Voice-mail

Table 18: Elastic Applications

Tolerant ; Adaptive, Delay Adaptive
Tolerant ; Adaptive, Rate Adaptive
Intolerant ; Adaptive ;
Intolerant ; non-adaptive ;

Table 19: Inelastic Applications

ATM RSVP/Q.2931 or PNNI mapping QoS parameter mapping 802.1q FR
SMDS Native

Token Rate Bucket Depth The Tspec/Rspec parameters - Version — Maximum
Transmission Unit Token Bucket Rate — Token Bucket Size Maximum Trans-
mission Rate — Minimum Delay Noticed Maximum Delay Variation — Loss
Sensitivity Burst Loss Sensitivity — Loss Interval Quality of Guarantee —