Digital Communications II

— The Internet —

Jon Crowcroft (thanks to Steve Hand)

Michaelmas Term 2002

http://www.cl.cam.ac.uk/users/jac22/

Recommended Reading

- Srinivsan Keshav. (1997). *An Engineering Approach to Computer Networking*. Addison-Wesley Pub Co; (1st ed.); ISBN: 0201634422

- *Alternative to Keshav:* Bruce S. Davie & Larry L. Peterson & David Clark (1999). *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers (2nd ed.); ISBN: 1558605142

- W. Richard Stevenes (1994) *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley Pub Co, (1st ed.); ISBN: 0201633469

- *Alternative to Stevens:* Douglas Comer (2000). *Internetworking with TCP/IP Vol. I: Principles, Protocols, and Architecture* Prentice Hall (4th ed.); ISBN: 0130183806

- *Backround:* Balachander Krishnamurthy & Jennifer Rexford (2001) *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*. Addison-Wesley Pub Co (1st ed.); ISBN: 0201710889

- Leffler S.J. et al.

The Design and Implementation of the 4.3BSD UNIX Operating System
Wokingham, 1986.

- FreeBSD source code (BSD Net 2)
- Linux source code (not so good) but see also Jon Crowcroft's forthcoming book:-)
- Internet RFCs, FYIs and drafts.
- The web (e.g. `http://freesoft.org/CIE/index.htm`)

Quick Recap

Background from Operating Systems Courses

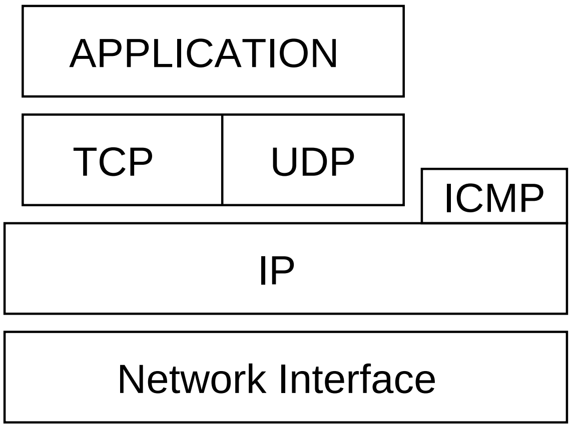
- Interrupts
- Scheduling
- Processes
- Concurrency
- Software Interrupts

Background from Digital Communications I

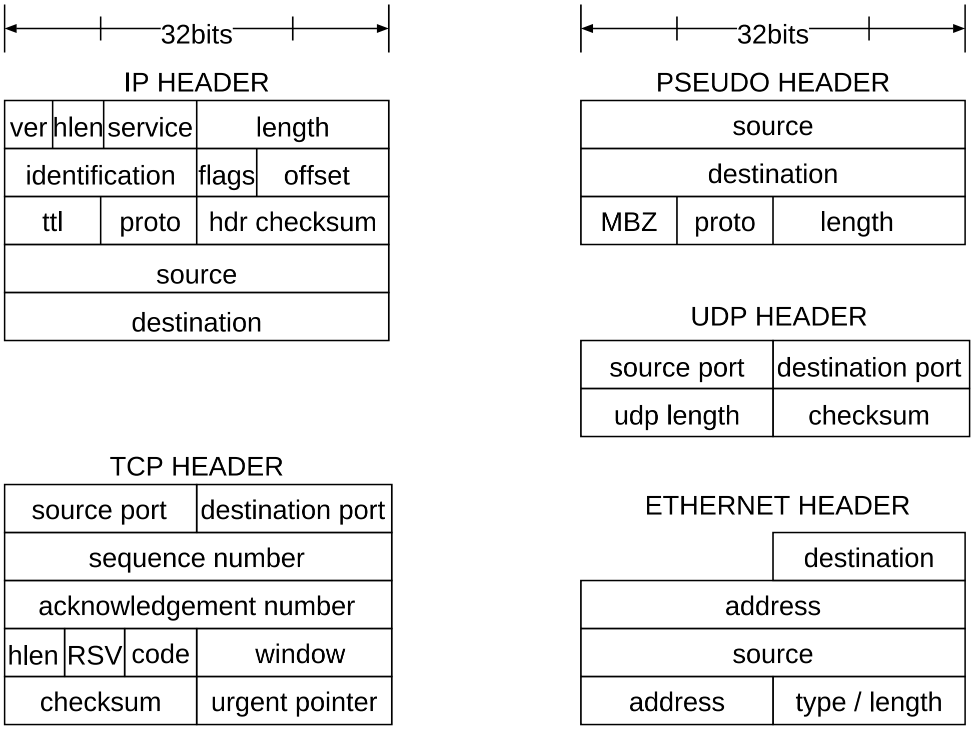
- Layering, Channels, Multiplexing
- TCP/IP Stack (UDP, TCP, IP, ICMP)
- IP on Ethernet (ARP)
- Addresses and Routing
- TCP (windows, flow control, ACKs etc)

General Organisation of IP

How the protocols fit together



What the protocol headers look like



Communications Units

Communications uses S.I. units.

- For example a network clocked at 8MHz moving 1 bit per clock period is a 8Mb/s network.
- To transfer one "megabyte" of information will take 1.048576 seconds.
- $K=10^3$, $M=10^6$, $G=10^9$
- In computer science, K only equals 2^{10} where address lines are involved.

People often get this wrong – be careful. Use "octet".

Data Representation

- Endian
- Size of "integer"
- Compression vs. ease of access
- Floating point
- Complex data. . .

What's different about networking?

- Failures are different
- Binding model is different
- Interrupts can be unexpected
- Security model is different
  - Must protect against corrupt data
  - Must protect against malevolent packets

Outline

- Introduction and Recap.
- BSD Unix: sockets, buffering, interfaces.
- IP: addressing, forwarding, checksum, fragments
- Connectionless protocols: ICMP, UDP, NFS.
- TCP/IP: basic operation, congestion schemes.
- QoS and the Internet.
- Routing protocols: distance vector, link state.
- Multicast: basic model, routing.
- Conclusion.

Sockets Abstraction

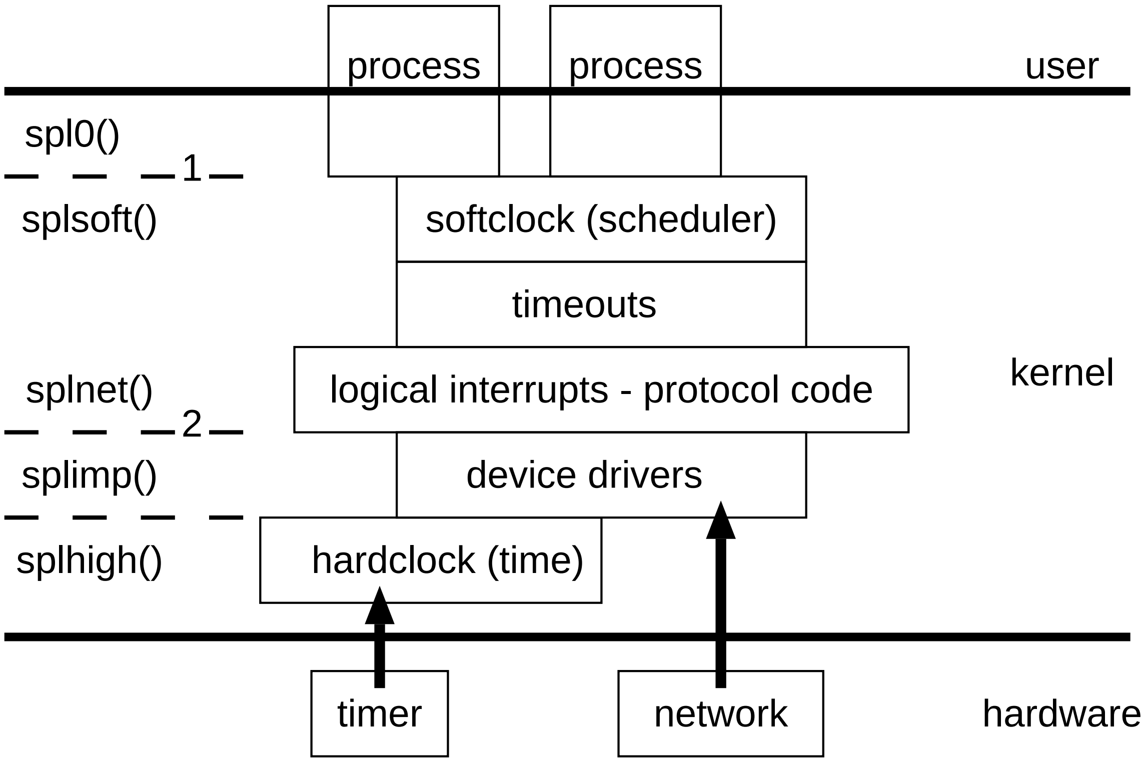
In Unix, everything is a file [descriptor] $\Rightarrow$ represent communications endpoints as fds also:

- `socket`: to create a new socket
- `bind`: for setting or allocating local address
- `listen`, `connect`, `accept`: for making connections
- `recv`, `send` (and `read`, `write`): for receiving or sending data on connection-oriented sockets
- `recvfrom`, `sendto`: for receiving or sending data on connectionless sockets.
- `select`: for user-level demultiplexing.

In Unix, processes are untrusted with virtual time and virtual memory $\Rightarrow$ kernel must handle:

- demultiplexing,
- device access,
- retransmissions, and
- data buffering.

BSD Unix concurrency



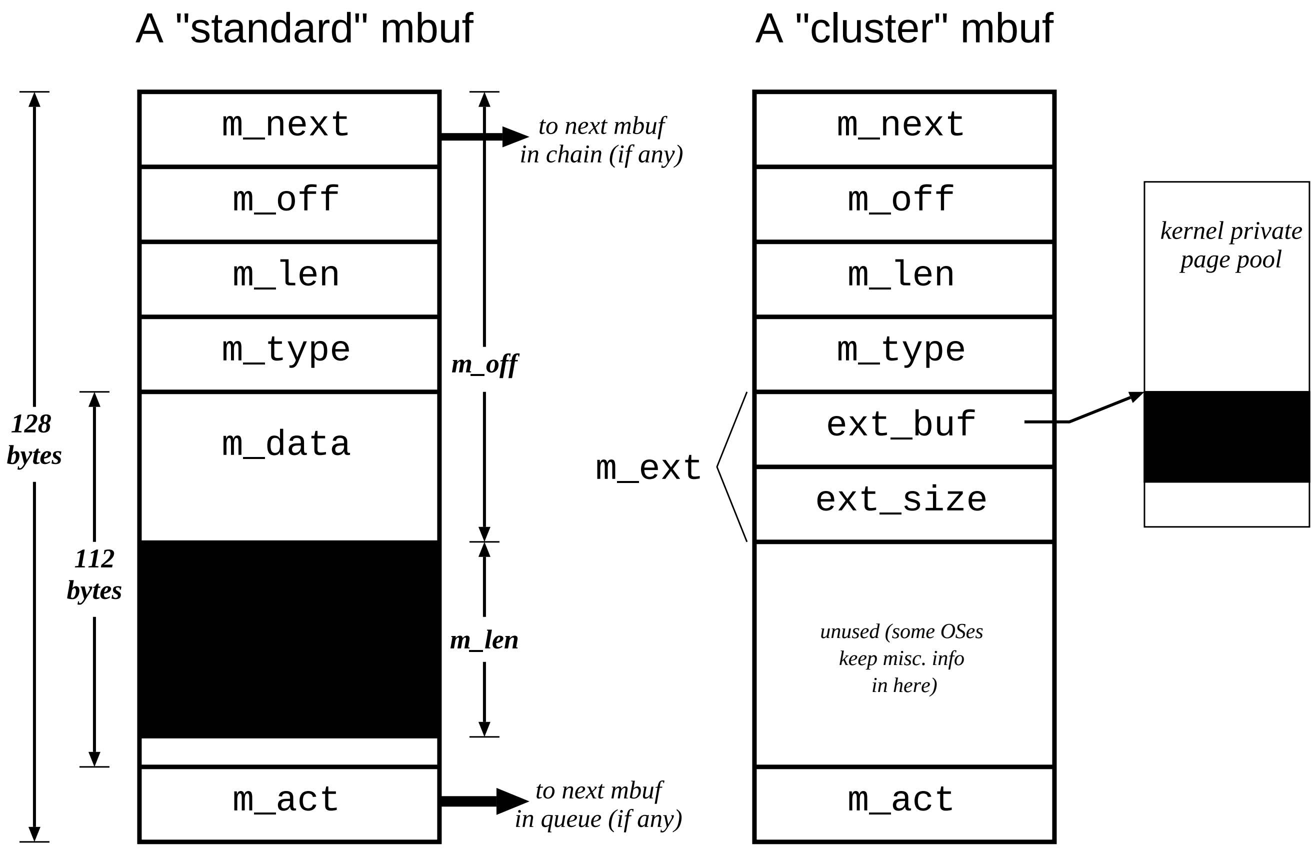
Several points to note:

- Clock interrupts are never missed.
- Device processing is done at raised IPL, other protocol processing is done at a lower level.
- (un)interruptible sleeps.
- How do input and output routines interact?

Buffering

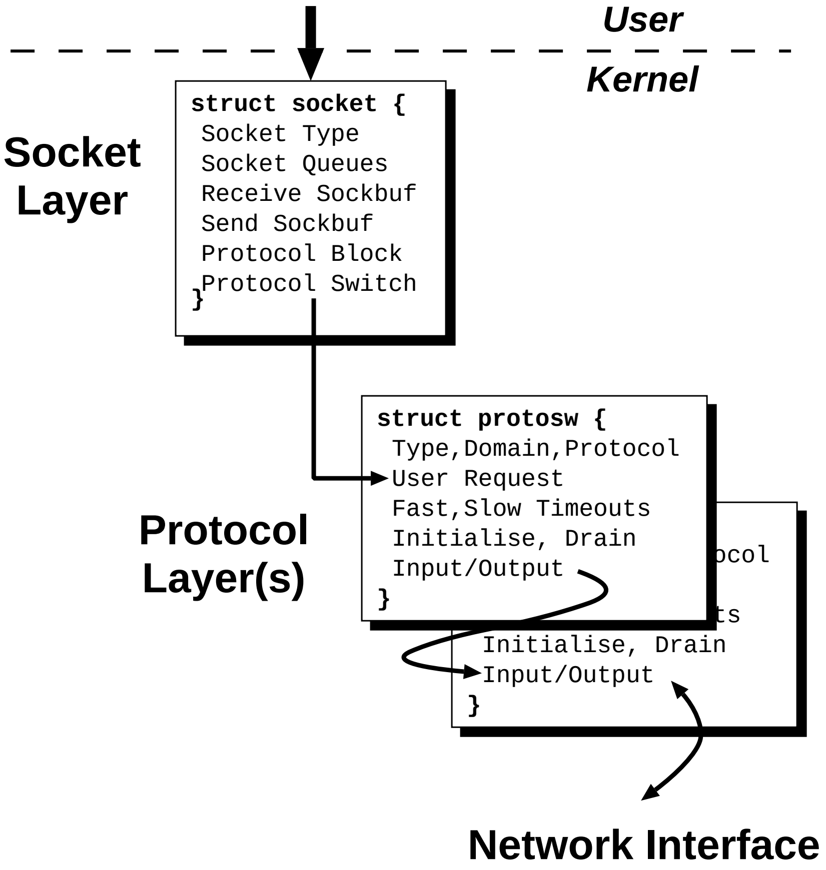
- Buffering for networks different than for block devices since:
  - data typically variable size.
  - protocols typically *layered*
- Want more flexible scheme: e.g. *mbufs* (4.3BSD).
- Two level hierarchy of linked structures:
  - `m_next` links mbufs into a *chain*.
  - `m_act` links chains into *lists* or *queues*.
- Fixed size (128 or 256 bytes) $\Rightarrow$ alloc/free is easy.
- Use `m_off` and `m_len` to "top and tail": handy for nested protocol data.
- Large data is held elsewhere: both *cluster* mbufs and nasty "loaned mbuf" kludge.
- Problems: copying (socket level, transport level, (device driver?)).
- Is zero-copy possible?

Mbufs



- Alloc/free through `m_get()`, `m_free()`.
- Variable size management with `m_adj()`.
- Copy with `m_copy()` : can use remap in case of clusters.

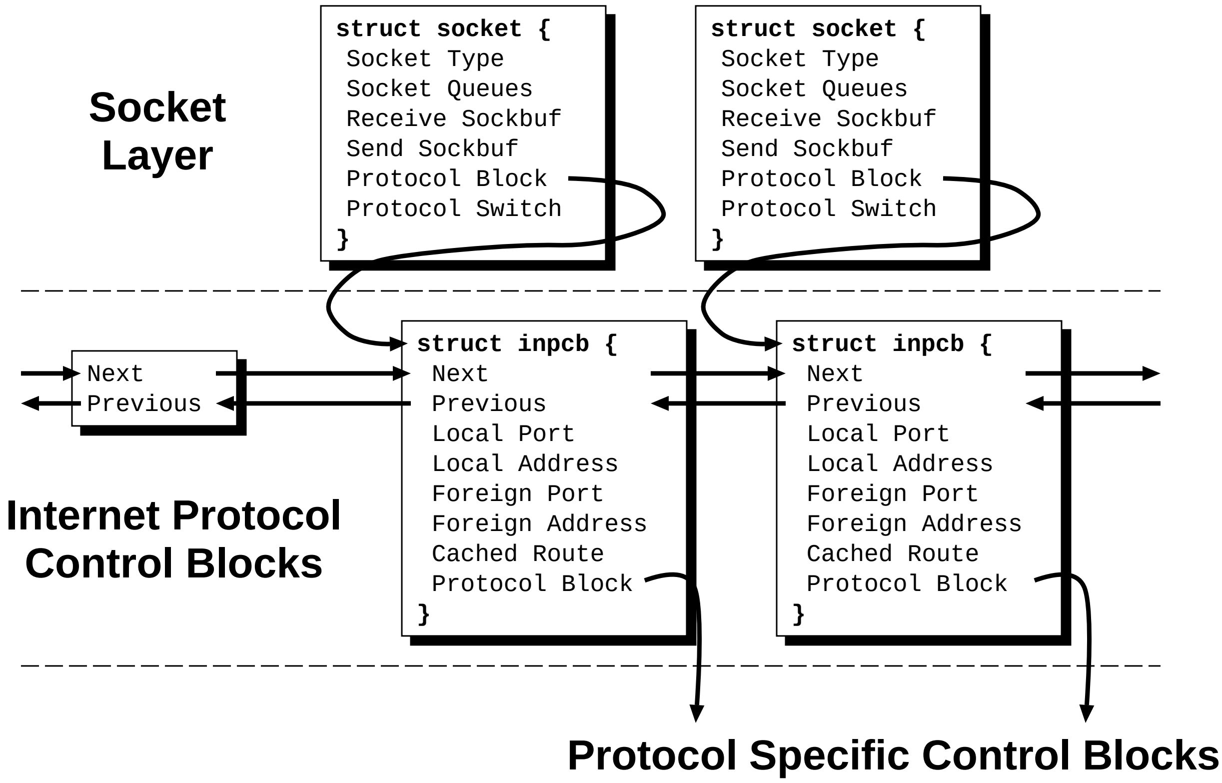
Implementation: Higher Levels



- Library provides socket interface to applications.
- Syscalls trap to kernel socket layer.
- "Object-oriented" : each protocol represented by a *protocol switch* structure.
- Network interfaces represented by own structure.

Implementation: INPCB

For all internet sockets have an INPCB (protocol control block):



- Stored as a doubly linked list per protocol.
- Lists searched on every packet arrival $\Rightarrow$ keeps a single "cache" entry at the front of the list.

Implementation: Network Interface

Network device drivers have an output routine:

- Called at `splimp`
- Given an mbuf chain to send
- Responsible for encapsulation
- May have to deal with loop-back and/or broadcast
- May upcall for address resolution (ARP)

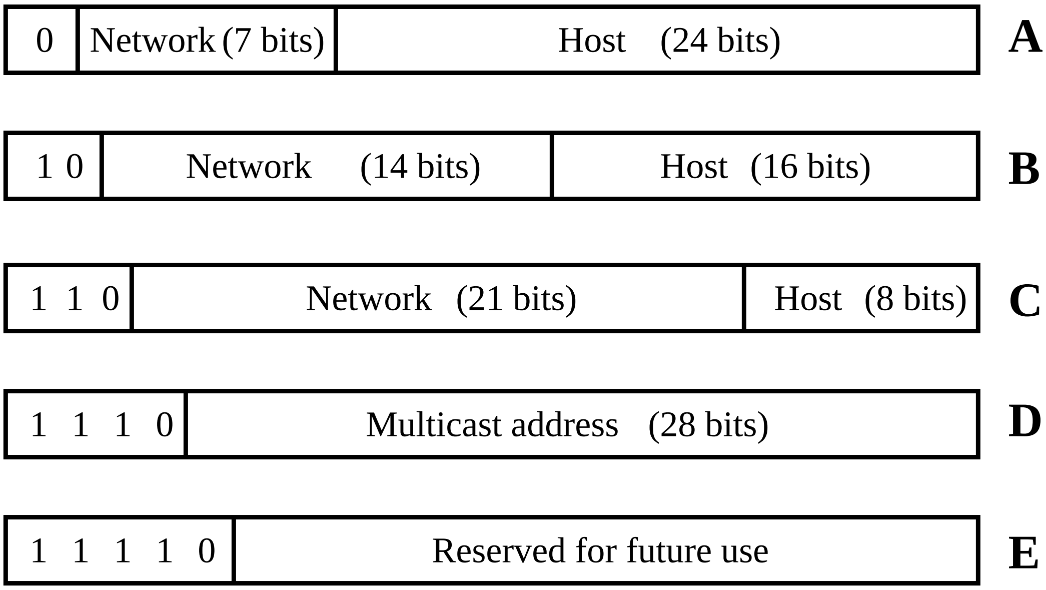
On receive interrupt, device drivers:

1. demux packet to find protocol stack input queue
2. add packet to queue
3. request a software interrupt for the appropriate protocol input handling routine.

Q: how does protocol output identify interface to use?

IP Addresses

- In IPv4, addresses are 32-bits.
- An address identifies a network or an IP host.
- Addresses separated into 5 “classes”, **A**–**E**.



- Also have magic addresses:
  - host all zeros $\Rightarrow$ refers to network itself.
  - host all ones $\Rightarrow$ broadcast address for a (specified) network.
  - network all zeros $\Rightarrow$ “this network”.

– network $127 (= 01111111_2) \Rightarrow$ loopback.
– all bits ones $\Rightarrow$ 'limited' broadcast
– all bits zeros $\Rightarrow$ "this host".

IP Forwarding (I)

- Forwarding (and routing) based on *prefixes*.
- Keep a table mapping IP addresses to *next hop*:
  – IP address is usually a network...
  – Next hop includes outgoing interface and next IP address (if applicable).
  – Can use default route to keep table small.
- To forward an IP datagram:
  1. compute network prefix (using class info).
  2. if directly connected, send datagram directly.
  3. if any host or network route matches, send to next hop via outgoing interface.
  4. else use default route.
- Hosts must forward every outgoing IP packet (why?) $\Rightarrow$ want it to be fast.

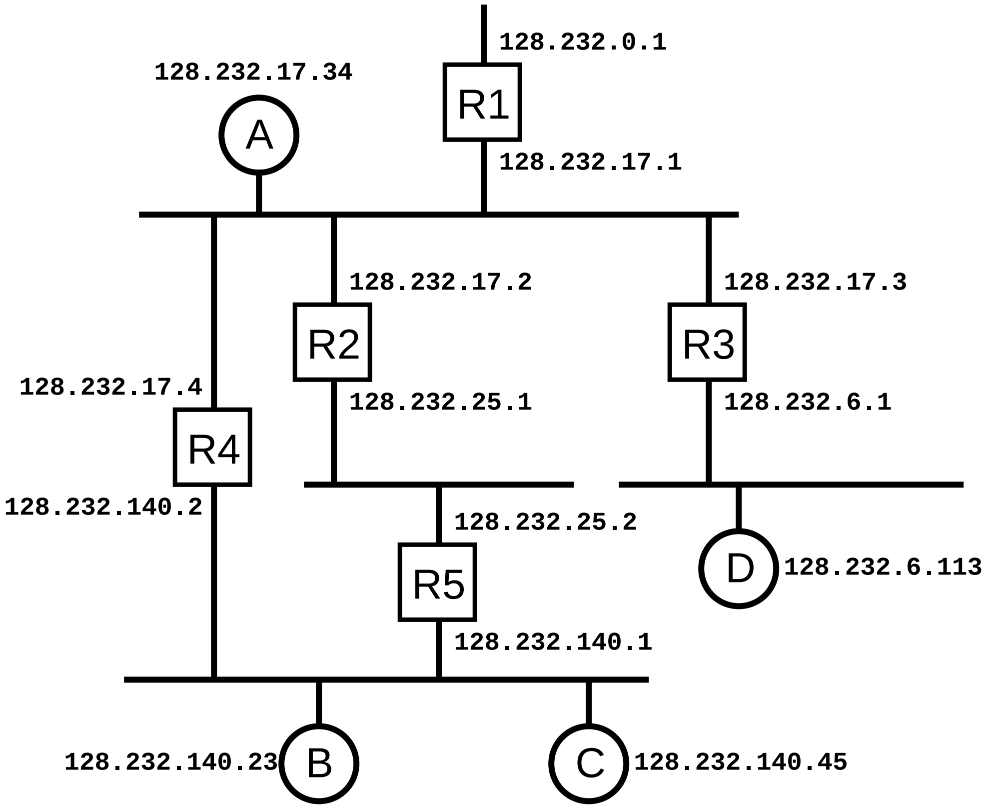
IP Forwarding (II): Subnetting

- Original IP addressing (as just described) has three fixed 'IP network' sizes ⇒ fragmentation.
- Subnetting improves this by relaxing the 1-to-1 physical address ↔ IP network mapping, e.g:
  1. Allocate organisation class-B address
  2. Inside org have multiple physical networks distinguished by *subnet*, e.g.

	14 bits	k bits	16-k bits
10	Network	Subnet	Host

  3. Forward between these using *subnet routing*.
  4. Outside world oblivious.
- To implement use *subnet masks*: e.g. $(16 + k)$ `1`'s followed by $(16 - k)$ `0`'s.
- Routing table now holds mask for each entry.
- When checking a route first `AND` destination IP addresses with associated subnet mask.

Subnetting Example



Routing table at host B:

Network	Netmask	Next Hop	I/f
128.232.140.0	255.255.255.0	Direct	eth0
128.232.17.34	255.255.255.255	128.232.140.1	eth0
128.232.25.0	255.255.255.0	128.232.140.1	eth0
127.0.0.0	255.0.0.0	Direct	lo
0.0.0.0	0.0.0.0	128.232.140.2	eth0

IP Header Checksum

IP datagrams have a checksum over the header:

- every incoming packet must be validated
- every outgoing packet must have checksum recomputed

⇒ must be *fast*.

Q: Why don't we checksum data?

Checksum algorithm

Defined in RFC1071:

1. split the input into 16-bit words
2. add them together using 1s complement math
3. complement the result.

To implement need to consider carefully:

- the difference between 1s and 2s complement arithmetic
- host endian
- what the target CPU is capable of

What is 2s complemement sum?

- Top bit has value -2^n. (e.g. -32768).
- Problem is loss of carry out of the top bit. Consider:

$$\begin{array}{r} \texttt{0x4000} \\ \texttt{0x4000} \\ \texttt{0x4000} \\ \texttt{0x4000} \\ \hline \texttt{0x0000} \end{array} \qquad\qquad \begin{array}{r} \texttt{0x8000} \\ \texttt{0x8000} \\ \texttt{0x8000} \\ \texttt{0x8000} \\ \hline \texttt{0x0000} \end{array}$$

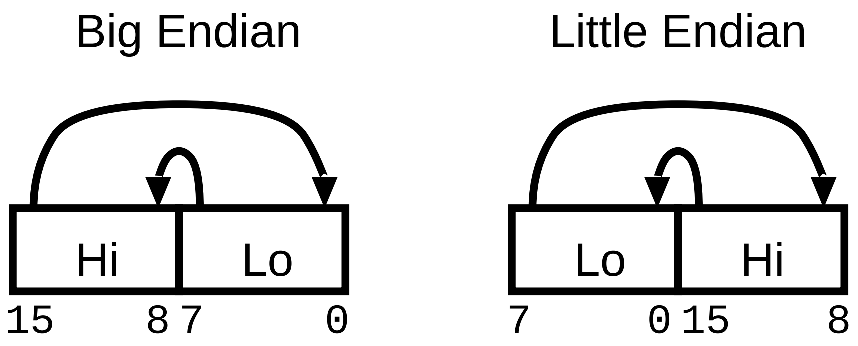
What is 1s complement sum?

- Top bit has value $-(2^n - 1)$ (e.g. -32767)
- Two values for zero: initial and computed.
- E.g.

$$\begin{array}{r} \texttt{0xffff} \\ \texttt{0x0001} \\ \hline \end{array}$$

- Carry is not lost.

Endian Independence



- Hence to compute:

2s complement	
Big	Little
sum	swap
	sum

1s complement	
Big	Little
sum	sum
carry	carry

- Example:

```
0x4510
0x0050
0xB0DB
0x4000
0x363B + 1
```

```
0x1045
0x5000
0xDBB0
0x0040
0x3C35 + 1
```

- NB: initial checksum computation assumes header checksum field zero ⇒ "full" checksum MBZ (since store complement in header).

Checksum Implementation (I)

Here's how Richard Black implemented it for ARM:

```
        sub     r3, r3, #16
loop:
        ldmia   r2!, {r6-r9}
        adds    r4, r4, r6
        adcs    r4, r4, r7
        adcs    r4, r4, r8
        adcs    r4, r4, r9
        adcs    r4, r4, #0
        adc     r4, r4, #0
        subs    r3, r3, #16
        bxx     loop
        add     r3, r3, #16
```

Draw your own conclusions.

Exercises for the reader:

1. What should the xx be in "bxx loop"?
2. Why does he add zero twice?
3. What do you need to do to the result before comparing it with the checksum field in the packet header?

Checksum Implementation (II)

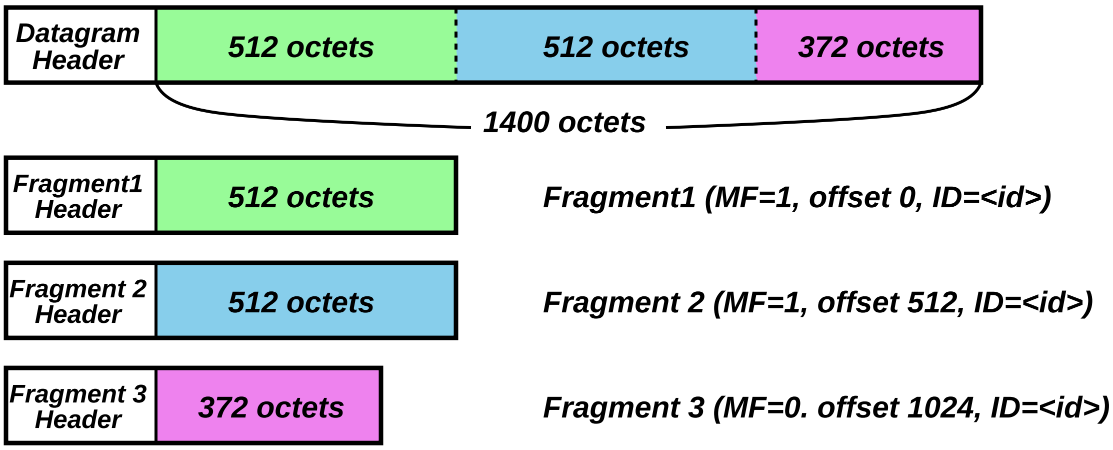
And for x86 (from linux):

```
static inline unsigned short
ip_fast_csum(unsigned char * iph, unsigned int ihl) {
    unsigned int sum;

    __asm__ __volatile__("
        movl (%1), %0
        subl $4, %2
        jbe 2f
        addl 4(%1), %0
        adcl 8(%1), %0
        adcl 12(%1), %0
1:      adcl 16(%1), %0
        lea 4(%1), %1
        decl %2
        jne 1b
        adcl $0, %0
        movl %0, %2
        shrl $16, %0
        addw %w2, %w0
        adcl $0, %0
        notl %0
2:
        "
    /* Since the input registers which are loaded with iph and ipl
       are modified, we must also specify them as outputs, or gcc
       will assume they contain their original values. */
    : "=r" (sum), "=r" (iph), "=r" (ihl)
    : "1" (iph), "2" (ihl));
    return(sum);
}
```

Lots more examples in linux source, BSD source, or RFC 1071. . .

IP Fragmentation



IP Packets may be up to $2^{16}-1$ bytes in length. To fit over networks with smaller MTUs they can be fragmented on eight byte boundaries. Header contains:

- "More Fragments" bit (to let you know that you need to reassemble)
- Fragment Offset field (multiples of 8) says where this IP fragment comes in the IP packet.
- ID field (so can put fragments together again).
- Length in header refers to the length of the fragment (unfortunately. . .).
- Optional "Don't fragment" bit.

IP Reassembly

If receive an 'incomplete' IP packet then:

1. Lookup (ID, . . .) in the frag map.
2. If no match found:
   - create a new frag list
   - map (ID, . . .) to this new list
   - start the reassembly timer.
3. Add packet to the relevant fragment list.
4. If all fragments now present, cancel timer, unmap frag list and present it to upper layer.
5. If timer expires, discard fragment list (and fragments), and send back an ICMP.

Some questions:

- Why do we leave reassembly to final receiver?
- What value should the reassembly timeout have?
- If two fragments overlap, which bits do you throw away? Why is this important?
- Which is the best order to transmit fragments?

Internet Control Message Protocol

- Internet version of "exceptions"
- Delivered to source of datagram
- Many types, some are:
  - echo request / echo reply
  - destination unreachable (protocol, port host)
  - redirect (only to directly connected host)
  - time to live exceeded (to stop infinite cycle)
  - cannot fragment
  - reassembly failed
  - source quench
- Note: ICMP is an example of a *layering violation* (although one which is probably justified)

UDP

When a packet arrives:

- check header.
- check checksum.
- Search INPCB to get socket(s).
- Add data to socket (may be dropped by socket buffer code)
- Prod socket (in case of `select()`, etc.)

To transmit a packet:

- Optionally does a `bind()`
- Optionally does a `connect()`
- Builds a UDP header, links on new mbuf.
- Calculates UDP checksum including "pseudo header" of IP source and destination. Why?
- Optionally disconnects
- Optionally unbinds

Last two stages can cause problems for ICMP delivery.

Pseudo Headers

IP checksum field is for the header only.

- ✔ Routers only have to check a little bit.
- ✔ Protocols are free to use whatever check they like.
- ✘ Higher level packets may be indistinguishable from their headers.

The "end-to end-argument" applies here; higher level protocol wants to know where the ends are.

- UDP and TCP checksum fields include the addition of a pseudo header which is never actually sent in the data.
- Consists of four fields: source, destination, length and protocol.
- Protects against data and headers being switched in some bogus router.
- Causes problems for some multi-homed hosts. . .

NFS

The Network File System. Is built over Sun RPC and XDR. One of the “arguments” to the RPC call is disk data of up to 8K.

⇒ Lots of fragmentation

⇒ Timing retransmissions is a problem

⇒ Only sensible in the local area

⇒ Stateless servers and idempotent requests (at least once semantics)

⇒ Maps file names to file handles (which contains device and inode number)

⇒ Locking and coherency is a problem (not done mostly)

⇒ File handles are frequently guessable — security concerns.

⇒ Need multiple outstanding requests per client for performance.

Routing

Routing in the Internet is difficult:

- Potentially $2^{21} + 2^{14} + 2^7 \approx 2^{21}$ networks
- No geographical organisation of numbers or networks (to be fixed in IPv6)
- Rich interconnection
- Policy, security, . . . considerations.

In the beginning it was very simple.

- Single backbone "arpanet"
- Manual control of routing tables at individual sites.
- Default route towards the backbone
- Backbone knows where all the sites are.
- Backbone uses GGP (or equivalent).

The ROADS Problem

- Subnetting meant class-B nets being used up rapidly ⇒ introduced *supernetting*: allow n contiguous class-C nets to be handled as one.
- Once done this, trivial extension to deal with arbitrary length network prefixes.
- This is called *classless routing* (or CIDR).

Hence routers have routing tables which deal with networks with variable length prefixes.

Route lookup now involves *longest prefix match*.

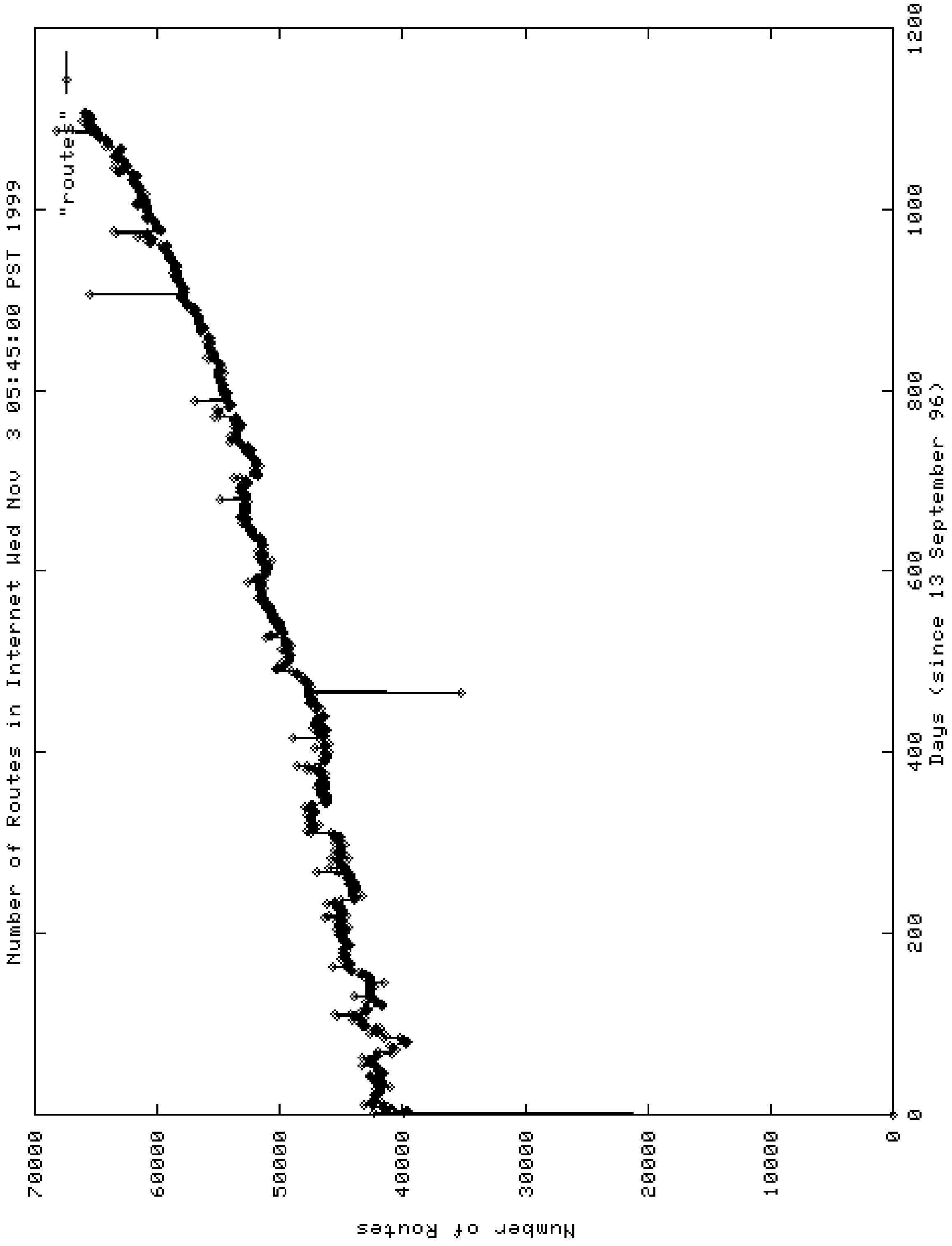
Management Complexity

Individual "sites" became too complex for static management ⇒ development of *autonomous system*:

- Set of routers in same organisation, under same management.
- No policy or security considerations internally...

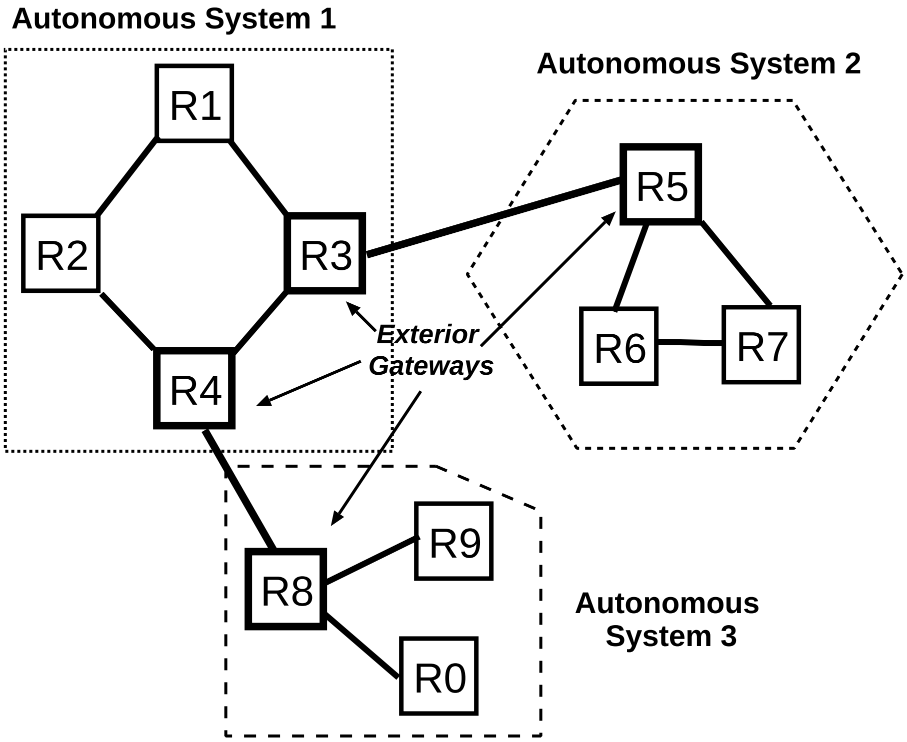
Thus two stage routing is possible:

1. determine the "next AS-hop" (and the relevant gateway which connects to it)
2. determine how to get there within your own AS.



Number of Routes in Internet Wed Nov 3 05:45:00 PST 1999
"routes"
Number of Routes
70000
60000
50000
40000
30000
20000
10000
0
0
200
400
600
800
1000
1200
Days (since 13 September 96)

The Big Picture



Today pretty much everyone uses BGP-4:

- Nominated exterior gateways talk E-BGP with a peer in another AS.
- Interior gateways talk I-BGP to handle AS route info (e.g. updates from peers)
- *All* gateways also talk some *interior gateway protocol* (IGP).
- (this is required to underpin BGP)

Distance Vector

- Every router discovers which other routers are on the same networks as it.
- Tells each of them the networks which it knows how to reach and the distance (in hops) to get there.
- Listens for updates from them to add to its tables.
- Sends regularly
- If it doesn't hear for a while then assumes that router is dead and removes all associated routes from its table.
- Lots of subtleties in practice . . .
- Respond slowly to change
- Messages can get large.

Converges to correct state (bit like a big distributed Dijkstra's algorithm)

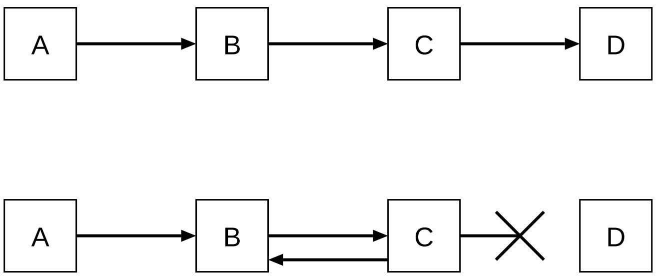
Examples:

- RIP
- HELLO

RIP

Suppose $d(i,j)$ represents the "distance" (or "metric") from i to j. When a message arrives at A from B advertising a route to C. A calculates $d(A,B)+d(B,C)$. If this is less than the current $d(A,C)$ **or** B is the current next hop for C, then $d(A,C)$ is updated to that value and B is set up be the next hop for C.

Consider this diagram. What happens when the link from C to D breaks?



C must time out the route to D and set the cost to infinity. It will then believe an advertisement for D from B.

Packets go round in circles. B and C metrics increase. This is called "counting to infinity". How does this get terminated?

Split Horizon

- Never send an advertisement on the same interface as the current route.

- This would stop B advertising D to C, but doesn't solve the problem in general topology.

. . . with poisoned reverse.

- Do send an advertisement but with infinite cost

- Puts packet sizes up again.

- Better, but still get loops.

Triggered updates

- When changing a metric send updates immediately rather than waiting for next cycle.

- Need to be careful to avoid broadcast storms.

- Has an effect on other routing behaviour

Still get counting to infinity sometimes.

Not a very efficient protocol, but Berkeley made code available in `routed` so used quite a bit. Early versions could only deal with networks, but newer (RIP v2) can deal with subnets too.

Link State

- Every router knows complete topology.
- Every router knows all its peers.
- Each router monitors the aliveness of its links/peers.
- Routers send information about change in aliveness of links/peers.
- Messages are small
- Responds quickly to change in working-ness.
- Manual change in configuration

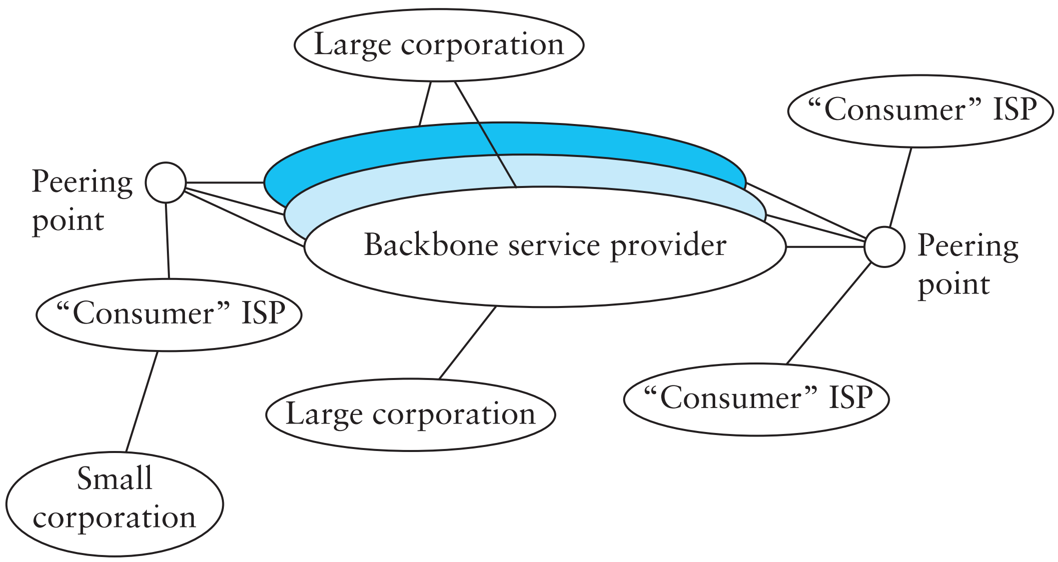
Everyone calculates correct state.

Everyone calculates their own Dijkstra's algorithm.

Examples:

- Open Shortest Path First (OSPF), developed by the IETF (see RFC 1247).
- Intermediate-System to Intermediate-System (IS-IS), originally from the OSI routing suite.

BGP-4: Model and Terminology



Define:

- *local traffic* as traffic that originates or terminates on nodes within an AS, and
- *transit traffic* as traffic that passes through an AS.

Then can define the following AS types:

- *stub AS*: has a single connection to one other AS; carries local traffic only
- *multihomed AS*: has connections to more than one AS but refuses to carry transit traffic
- *transit AS*: has connections to more than one AS; carries both transit and local traffic

BGP-4: Operation

Each AS has one or more exterior gateways (or border routers), and at least one *BGP speaker*.

BGP speakers communicate by sending messages over TCP; there are four different message types:

- OPEN: this is used to initialise a BGP session between peers
- KEEPALIVE: a simple ‘ping’ to check if the BGP session is alive
- NOTIFICATION: used for error reporting.
- UPDATE: used to advertise or withdraw prefixes

Algorithm is *path vector*: essentially distance vector, but deals with complete *paths* rather than single hops. That is, advertisements look a bit like:

128.232.2/18 is reachable via {AS1203, AS20, AS42}

In general the shortest AS-path advertised for a prefix will be used.

Each path can also include *attributes* such as:

- LOCAL-PREF: a locally valid metric,
- MULTI-EXIT-DISCRIMINATOR (MED): which exterior gateway to use to get to the first AS.

Routing: Summary

In the modern world, IP addresses are essentially *classless*: i.e. networks are represented by a prefix.

At least three different kinds of 'routing' take place:

1. End-systems use subnetting, default routes and (perhaps) ICMP redirection and/or RIP snooping

2. Interior gateways use an IGP (typically RIP or OSPF) for intra-domain routes:
   - typically concerned with efficient use of network capacity
   - policies/metrics reflect internal load $\Rightarrow$ routes are typically symmetric.
   - unaware of other ASs.

3. Exterior gateways use an EGP (typically BGP-4) for inter-domain routes:
   - (almost) unaware of interior structure
   - concerned with loop-free paths.
   - policies/metrics reflect economic cost $\Rightarrow$ routes tend to be asymmetric.

Key to performance is building efficient *forwarding information bases* (FIBs) from routing tables.

TCP

TCP is the most important of the internet protocols. Sufficiently so that most “suits” don’t realise there’s a difference between TCP and IP.

Why?

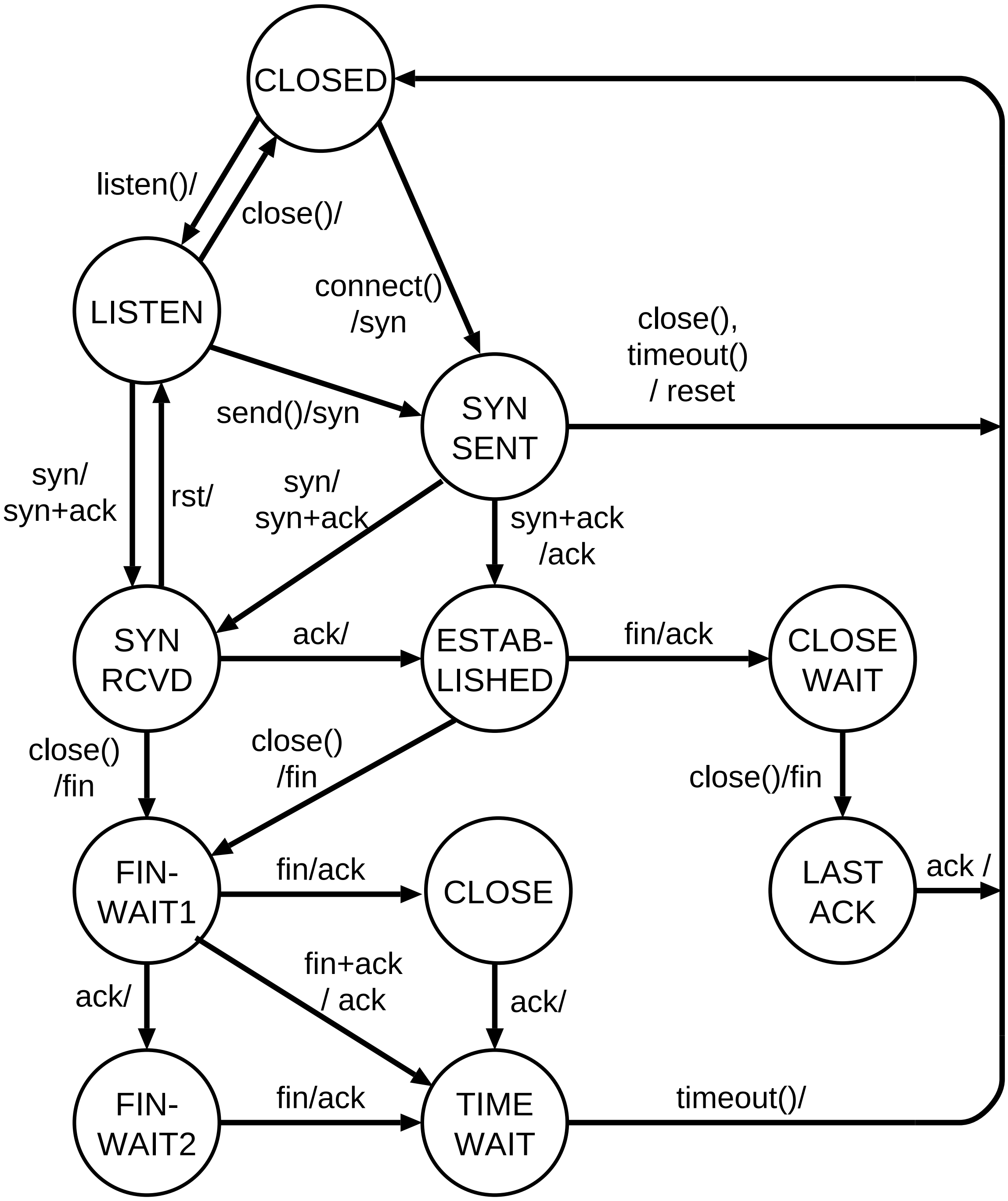
- Only standard protocol with built in error-recovery
- Only standard protocol with built in flow-control
- Only standard protocol with built in congestion-control
- Used for email.
- Used for telnet and rlogin.
- Used for ftp.
- Used for the web.

A connection is identified by the pair of endpoints. Servers may accept multiple connections from different clients to the same port number. This leads to well known service ports for various purposes. See `/etc/services`.

A connection is full duplex with ACKs and control potentially piggy-backed on data messages flowing in the other direction.

It measures the network to determine how fast to send, flow control is simplex.

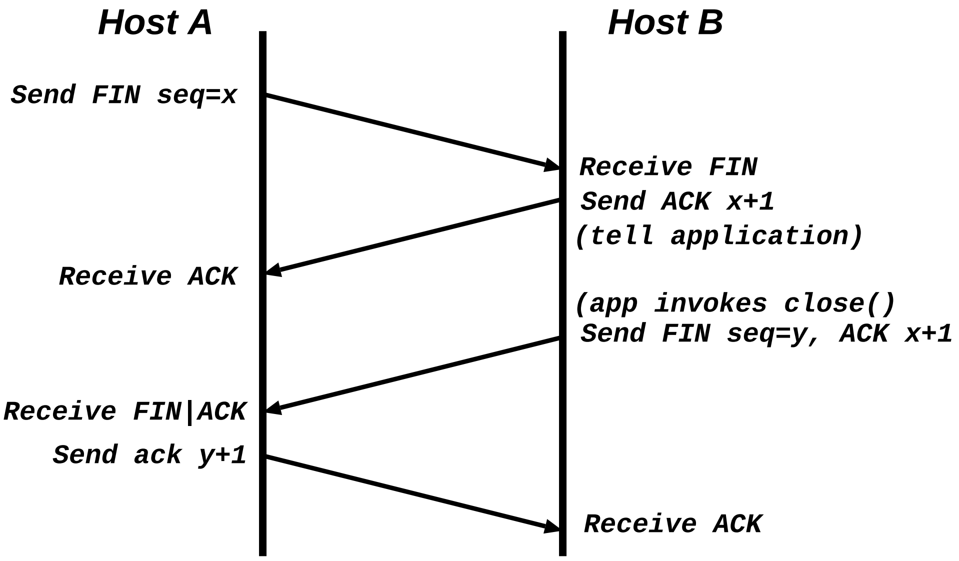
TCP State Diagram



Arcs show input causing transition and generated output. See RFC (or – better – books) for full details.

(Note that TCP supports bidirectional connect. This is never used in practice.)

Connection tear-down is subtle:



A TCB (TCP Control Block) is kept as well as a INPCB for management of all the TCP specific states, retransmissions, and timers etc, etc, etc.

INPCB and TCPCB allocated as soon as SYN arrives – problem?

Maximum Segment Size

You want to send data in packets as large as possible to amortise per-packet overheads, but a very large packet may get fragmented.

If a TCP packet gets fragmented and one of the fragments gets lost then the whole packet will have to be sent again - a waste of bandwidth (because TCP retransmits data for reliability on a per-packet basis).

So, to be efficient, TCP needs to know the MSS it can use without causing fragmentation. Worse, this can change because packet routes can change.

So it guesses. Looks at underlying network likely to be used, looks at routing information. Tries to predict if remote machine is on same net or not. If so uses MTU of interface, if not uses 576. Typically signal MSS to peer during connection setup.

This is an example of where layering *must* be violated for sensible implementation reasons.

Various techniques for determining the "current" MTU across the network exist. See RFCs for details.

Acknowledgements

Acknowledgements:

- Are cumulative. Specify next byte in entire stream.
- May be lost without causing retransmissions
- May arrive out of order
- Do not indicate additional data arrived — i.e. holes.

Sender must decide how much data to re-transmit.

- All data from indicated point?
- Just one packet worth?

Which does TCP use?

When does it do it?

Retransmission

Deciding when to retransmit a segment of data is crucial to the operation of TCP.

How long do you wait for the absence of an ACK before transmitting again? Need to know an estimate for round trip time. Can vary from $< 1ms$ in the LAN to $> 1s$ in the WAN. Factors:

- Physical distance
- Bandwidth delays
- Congestion delays
- Routing changes
- Congestion changes

Some examples from bescot.cl.cam.ac.uk:

- labes.cl.cam.ac.uk: 0% packet loss.
  round-trip (ms) min/avg/max = 0/0/0
- anugpo.anu.edu.au: 53% packet loss.
  round-trip (ms) min/avg/max = 473/491/546

TCP operates by using an adaptive retransmission algorithm.

Adaptive Retransmission (I)

Initial Algorithm

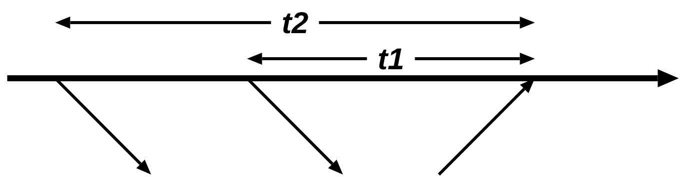
- Monitor the RTT using weighting:

$$\begin{aligned}\texttt{EstimatedRTT} &= \alpha\times\texttt{EstimatedRTT} + \\ &\quad (1-\alpha)\times\texttt{SampleRTT}\end{aligned}$$

- Timeout based on this:

$$\texttt{Timeout} = \beta\times\texttt{EstimatedRTT}$$

How to associate ACKs with Packets? An ACK states where in the stream receiver has got to, not which packet put it there. For a retransmitted packet which time interval do you take $t1$ or $t2$?



Answer?

This is the Karn/Partridge Algorithm: update RTT only with ACKs for unambiguous segments; in the absence of RTT samples:

$$\texttt{Timeout} = \gamma\times\texttt{Timeout}$$

What values to use for α, β and γ?
Common choices are $\alpha = 0.9$, $\beta = 2$, $\gamma = 2$.

Adaptive Retransmission (II)

Problems with high variance due to load.
E.g. queueing theory suggests that

$$\sigma \propto \frac{1}{1-L}$$

where σ is the variation in round trip time and L, $0 \leq L \leq 1$, is the load on the network.

$\Rightarrow$ using $\beta = 2$ means can only handle situations where most packets arrive within a factor of 2 of the RTT. This corresponds to a load of only 30%...

So a new algorithm (Jacobson/Karels):

$$\begin{array}{lcl}
\texttt{Difference} & = & \texttt{SampleRTT - EstimatedRTT} \\
\texttt{EstimatedRTT} & = & \texttt{EstimatedRTT} + \delta \times \texttt{Difference} \\
\texttt{Deviation} & = & \texttt{Deviation} + \rho \times (|\texttt{DIFF}| - \texttt{DEV}) \\
\texttt{Timeout} & = & \texttt{EsimatedRTT} + \eta \times \texttt{Deviation}.
\end{array}$$

What is the relation between α and δ ?

Choose values of $delta = 2^{-3}$, $\rho = 2^{-2}$ and $\eta = 2, 3, 4$

The above algorithm can then be executed *very* efficiently using scaled integer arithmetic.

Push

For an idle session, don't normally send data until a full packet's worth is waiting.

- Good for bandwidth
- Bad for telnet

Push operation allows sender to override this behaviour.

Silly Window Syndrome

Consider code like this:

```
do read(fd, &c, 1); while(f(c));
```

What will this do on the network?

Three ways to avoid this:

1. Delayed window advertisements. When opening the window from zero, don't advertise until the minimum of one MSS or $\frac{1}{2}$ of buffer space.
2. Delayed ACKs. When receiving data, delay an ACK for at most one clock if the window is "too small".
3. *Nagle's algorithm*: When sending, don't send less than a full MSS if there is data outstanding.

Congestion Collapse

Until now we've assumed packet loss was "random" (e.g. transmission errors) $\Rightarrow$ retransmitting packets as soon as possible is a good idea.

But suppose there is *congestion* in the network?

- $\Rightarrow$ packets will be queued at some router
- $\Rightarrow$ ACKs will take longer to come back
- $\Rightarrow$ sender will timeout and retransmit
- $\Rightarrow$ additional packets added to queue
- $\Rightarrow$ congestion is worsened

This is a positive feedback loop $\Rightarrow$ a **BAD** thing.

Hence need to manage congestion. This can be done at one or both of the following:

1. in hosts at edge of network (transport protocol)
2. at routers within the network (queueing discipline)

Queueing Disciplines (I)

Each router in the network implements a *queueing discipline*. This comprises two parts:

1. a scheduling discipline: determines the order in which packets are transmitted.

2. a drop policy: chooses which packets are dropped.

Most widely used discipline is *FIFO with drop-tail*, which implements a best-effort service. In this case, congestion control must take place at the edges.

Can add differentiation by using *priorities*, e.g.

- use TOS field in the IP header to distinguish between e.g. high and low priority packets.

- have one FIFO queue per priority level

- serve queues in strict priority order.

Problems:

- starvation of low priority packets

- (and no well defined incentive to use them)

- no guarantees about behaviour $\Rightarrow$ doesn't help us with managing congestion.

Queueing Disciplines (II)

A better scheme might be *fair queueing* (FQ):

- have one FIFO queue per *flow* ($\simeq$ set of packets comprising single connection)

- service queues in a round-robin fashion

$\Rightarrow$ not possible for a single (ab)user to steal more than $1/n^{th}$ of the link.

Points to note:

- doesn't specify a particular drop policy, but can assume drop-tail per (fixed-sized) queue.

- cost of classifying flows may be non-trivial (although can apply to *traffic classes* instead)

- still need to do end-to-end congestion control (since n varies with time and location) but at least not worried about 'driller-killers' or 'turbo TCP'

- trivial extension to *weighted* fair queueing (WFQ) also allows flow differentiation.

WFQ coming soon to a router near you...

TCP Congestion Control

Basic idea:

- assume as little as possible (i.e. best-effort network with FIFO or FQ)
- require each source to detect congestion, and to adjust its transmission rate accordingly.

Can be split into two sub-problems:

1. determining the available capacity in the first place
2. adjusting to changes in the capacity.

For the latter, introduce idea of a *congestion window*: an upper bound on utilised advertised window, i.e.

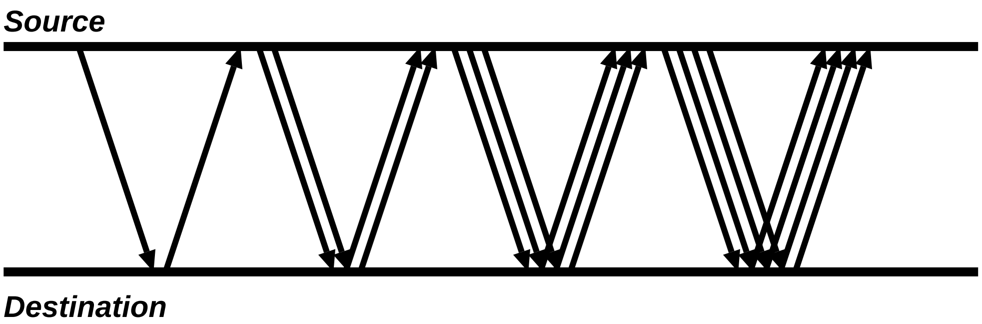
$$\mathtt{swin} = \mathtt{MIN}(\mathtt{awin}, \mathtt{cwin})$$

Then if congestion increases, reduce `cwin`; if congestion decreases, increase `cwin`.

For the former, assume that any packet loss signals congestion. Hence if lose a packet, reduce `cwin`.

This will prevent the positive feedback problem.

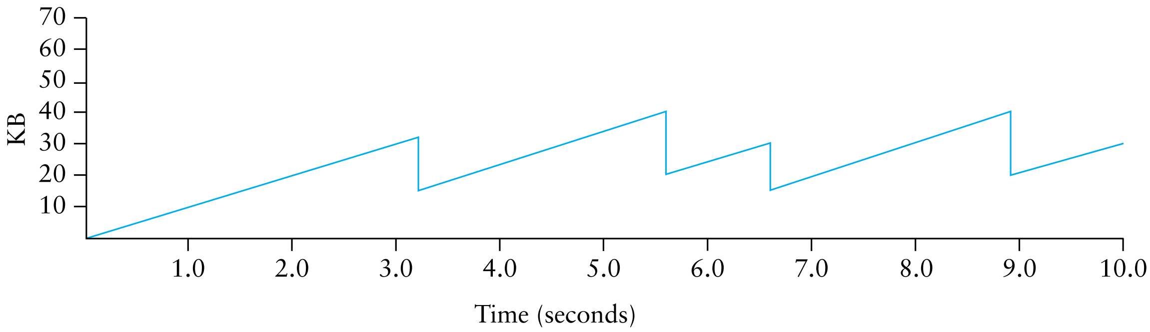
Additive Increase, Multiplicative Decrease



By how much should we increase/reduce `cwin`? TCP uses *additive increase, multiplicative decrease* (AIMD):

- Increment `cwin` by one packet per RTT; or, in practice, increment `cwin` a little for every ACK received, e.g. `cwin = cwin + (MSS * (MSS/cwin));`
- Divide `cwin` by two whenever a timeout occurs

This gives us the characteristic "sawtooth" of TCP:



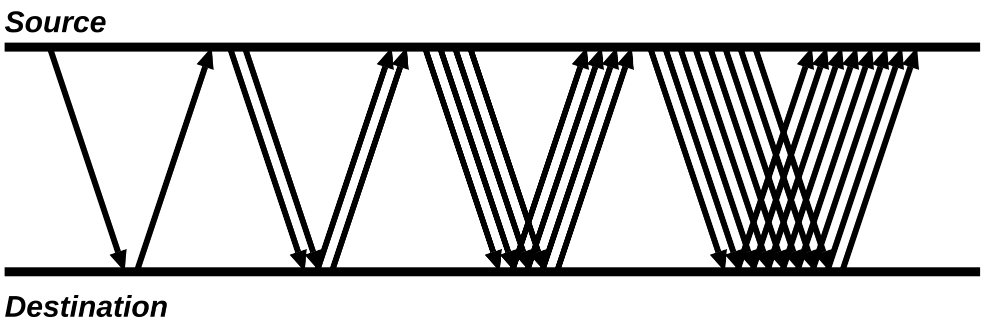
Slow Start

Additive increase is fine when we're near the correct operating point. But it takes too long to ramp up a connection starting from scratch.

Hence TCP uses (badly named) *slow start*:

- When starting a new connection increment the congestion window for every received ACK.
- Only $\log_2 n$ RTTs to open window to size n.

That is, increases congestion window exponentially rather than linearly:

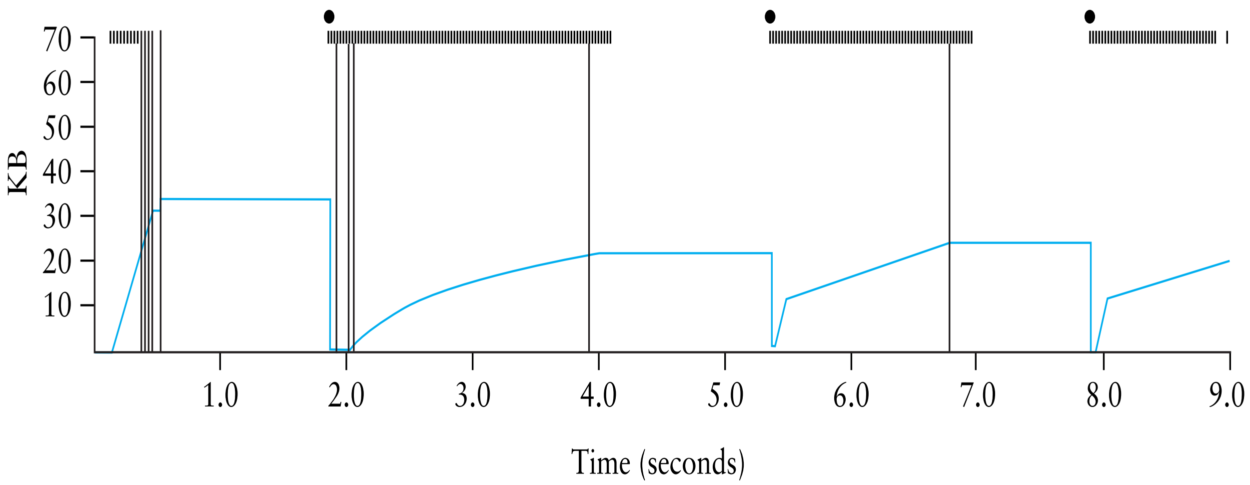


Slow start is also used for *congestion recovery*:

- when timeout, set `ssthresh = swin/2; cwin = MSS;`
- use slow start to reopen `cwin` until hit `ssthresh`
- once hit `ssthresh`, move to additive increase.

An Example

The below plots the congestion window of a TCP connection over time. Short bars represent packets being transmitted; long bars represent packets which will be lost; bullets represent timeouts



Several points to note:

- rapid increase at the beginning (*slow start*).
- *congestion avoidance* after timeout at ~2s, but then more losses ⇒ moves to additive mode.
- after timeouts at ~5.5s and ~8s:
  - `ssthresh` set to `swin/2`, `cwin` set to `MSS`,
  - slow start up to `ssthresh`, then linear until loss
  - flat until timeout

Fast Retransmit & Recovery

So far seem 'original' form of TCP congestion control. One obvious problem: coarse timeouts $\Rightarrow$ idle periods.

To solve this use *fast retransmit*:

- observe that receiver generally emits an ACK for every packet received.
- if loss or reordering occurs, cannot bump ACK value $\Rightarrow$ receiver will issue a *duplicate ACK*.
- if receive k duplicate ACKs, assume loss and initiate retransmit (in practice, k is 3)

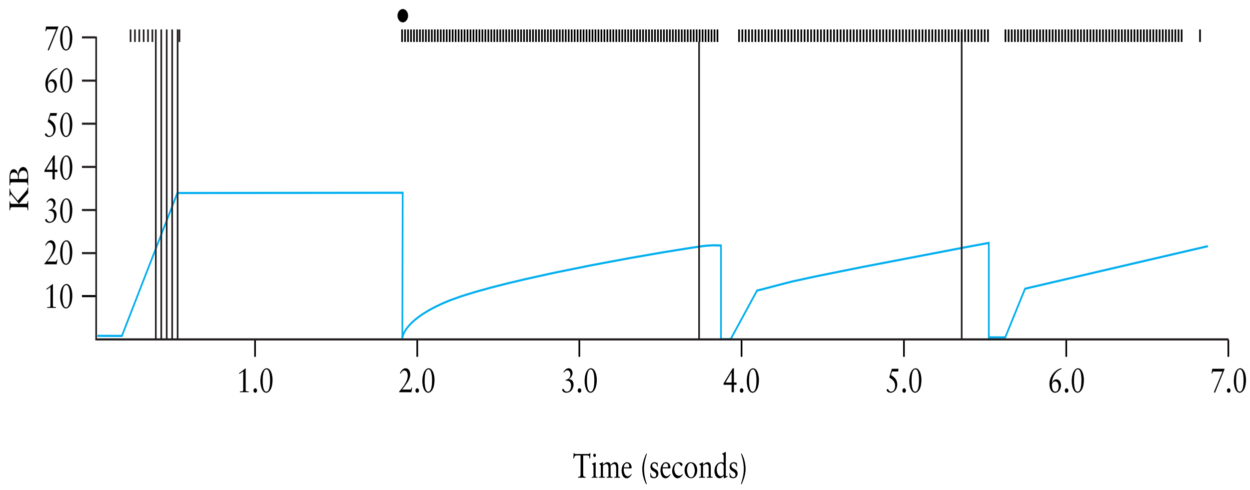
Can improve even more by using ACKs still in pipe to clock out data:

- when congestion signalled via duplicate ACKs, set `ssthresh = swin / 2; cwin = ssthresh + 3*MSS`
- each time another duplicate ACK is received, increment `cwin` and send another packet if allowed
- when a 'real' ACK arrives, set `cwin = ssthresh` and resume additive increase.

This is called *fast recovery*.

Example w/ Fast Retransmit

The below plots the congestion window of a TCP connection over time. Short bars represent packets being transmitted; long bars represent packets which will be lost; bullets represent timeouts



Notice that most "long flat' periods now removed.

It is not possible to avoid *all* timeout situations:

- if window is small (or loss large), then won't have 3 correctly received packets!
- this particularly affects initial slow start.

Avoiding Congestion

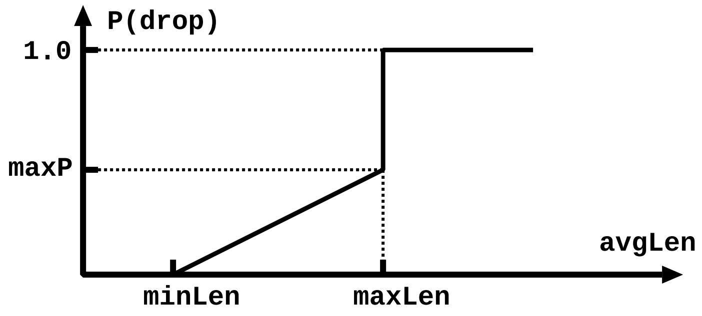
TCP attempts to control congestion when it occurs.

An alternative strategy is to attempt to avoid congestion occurring in the first place; e.g. packet marking in DNA:

- reserve a bit (the "DECbit") in the packet header.
- a router sets this bit if it is 'congested': that is, if the average queue length is ≥ 1.
- (average queue length is measured over last busy/idle cycle, plus current busy cycle)
- receiver copies the bit into its ACK
- sender adjusts its congestion window as follows:
  - if $\geq 50\%$ of the last window's worth of packets had the bit set, set `cwin = 0.875 * cwin;`
  - otherwise, set `cwin = cwin + MSS;`
- (this is just AIMD again with different values)

Advantage: no loss is required to detect (and react to) congestion. A similar scheme - "explicit congestion notification" (ECN) - is currently proposed for TCP.

Random Early Detection



Shceme proposed for the Internet by Floyd/Jacobson:

- like DECbit, each router monitors its queues to detect imminent congestion
- unlike DECbit, the router by default *drops* packets rather than mark them

Intuitively: hope to get TCP streams backing off 'earlier' than normal, and hence control load.

Decision on when to drop/mark also different:

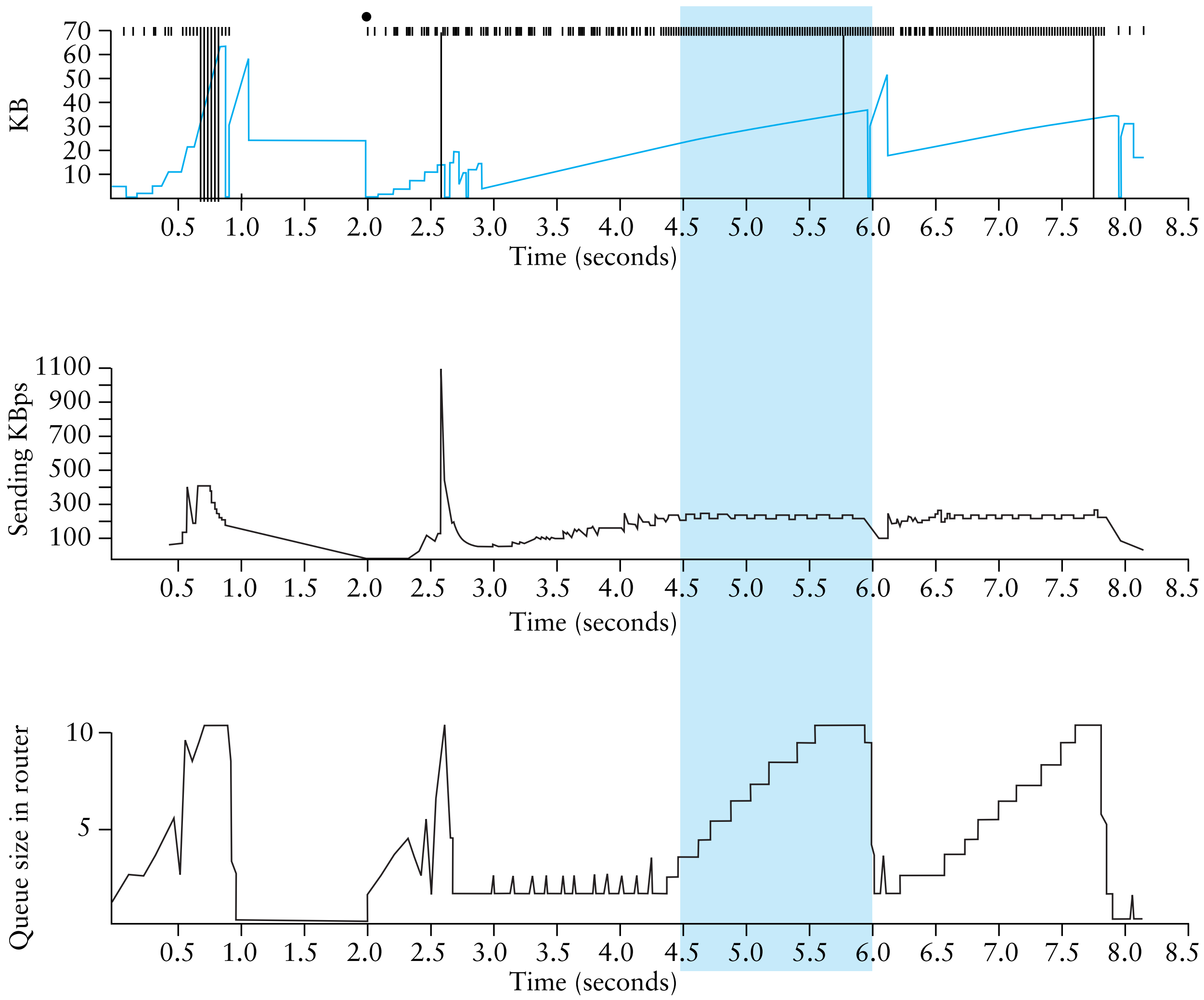
- RED computes an average queue length using a weighted running average:

  `avgLen = (1-w) * avgLen + (w * sampleLen}`

- Uses 2 threshholds `minLen` $\leq$ `maxLen` and probability `maxP` < 1 to probabilistically drop packets.
- choice of parameters non-trivial. . .

TCP Vegas: Intuition

Three graphs synchronised in time; top: congestion window; middle: observed throughput; bottom: buffer space at router.



Observe shaded area: congestion window increases, throughput doesn't — since a queue is building.

TCP Vegas: Algorithm

First define a flow's `BaseRTT` to be the RTT of a packet when the flow is not congested. Then:

```
ExpectedRate = CongestionWin / BaseRTT
```

In practice we don't know `BaseRTT`, so we use the smallest sample RTT ever observed for this flow.

We also compute `ActualRate` by:

- Noting when a given (distinguished) packet is sent
- Counting the number of bytes sent until the ACK is received.
- Dividing by the sample RTT.

Compute `Diff = ExpectedRate - ActualRate` ; we know that `Diff` is ≥ 0 by definition of `BaseRTT`.

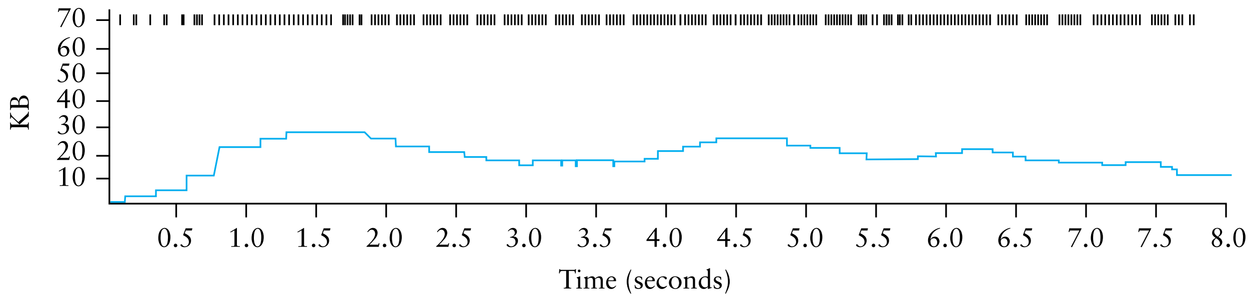
Define two threshholds $\alpha < \beta$ (intuitively corresponding to having 'too little' or 'too much' data in the network)

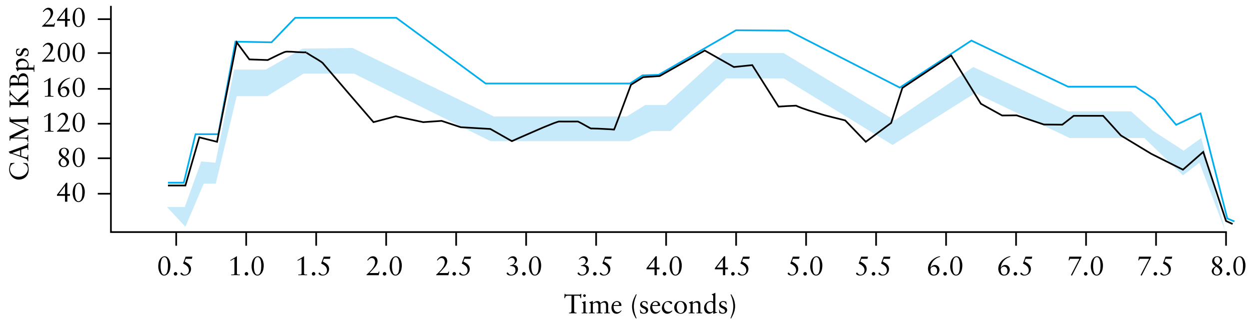
- if `Diff` $< \alpha$, we linearly increase `CongestionWin`
- if `Diff` $> \beta$, we linearly decrease `CongestionWin`
- otherwise we leave `CongestionWin` alone.

Use multiplicative decrease iff a timeout occurs.

TCP Vegas: Behaviour

Two graphs synchronised in time; top: congestion window; bottom: expected (blue line) and actual (black line) throughput. The shaded area is the region between the α and β threshholds.





Points to notice:

- Linear increase and linear decrease.
- Much smoother — no sawtooth!

Congestion: Summary

- One of the major problems with the Internet today (and source of many papers):
  - essentially all packet loss is due to congestion
  - unless have sensible queueing disciplines, abuse is possible (why play the game?)
- Simple AIMD scheme works to a certain extent, but does require repeated packet loss
- Explicit marking schemes combined with policing routers may allow us to avoid (most) packet loss.
- TCP Vegas is saner, and much more friendly — perhaps too much so?
- Congestion pricing scheme is yet another approach (as will be discussed later in the course)
- Still no guarantees. . .

Conclusion

"The Internet is a crock":

- lots of things can go wrong:
  - IP Spoofing
  - SYN attacks.
  - Open UDP ports.
  - Ping-of-death.
  - Teardrop
  - ...
- Implementation is tricky.
- QoS is pretty non-existent.
- Economics are poorly understood.

But: it's still one of the best (only?) things going...