The Network Stack

L41 Lecture 3
Dr Graeme Jenkinson
25 February 2018
This time: Introduction to Network Stacks

Rapid tour across hardware and software:
• Networking and the sockets API
• Network-stack design principles: 1980s vs. today
• Memory flow across hardware and software
• Network-stack construction and work flows
• Recent network-stack research
• The Transmission Control Protocol (TCP)
  • The TCP state machine
  • TCP congestion control
  • TCP implementations and performance
  • The evolving TCP stack
  • Labs 4 + 5 on TCP
• Wrapping up the L41 lecture series
Networking: A key OS function (1)

• Communication between computer systems
  • Local-Area Networks (LANs)
  • Wide-Area Networks (WANs)

• A network stack provides:
  • Sockets API and extensions
  • Interoperable, feature-rich, high-performance protocol implementations (e.g., IPv4, IPv6, ICMP, UDP, TCP, SCTP, 802.1, 802.11, ...)
  • Security functions (e.g., cryptographic tunneling, firewalls...)
  • Device drivers for Network Interface Cards (NICs)
  • Monitoring and management interfaces (BPF, ioctl)
  • Plethora of support libraries (e.g., DNS)
Networking: A key OS function (2)

• Dramatic changes over 30 years:
  1980s: Early packet-switched networks, UDP+TCP/IP, Ethernet
  1990s: Large-scale migration to IP; Ethernet VLANs
  2000s: 1-Gigabit, then 10-Gigabit Ethernet; 802.11; GSM data
  2010s: Large-scale deployment of IPv6; 40/100-Gbps Ethernet
  ... billions $\rightarrow$ trillions of devices?

• Vanishing technologies
  • UUCP, IPX/SPX, ATM, token ring, SLIP, ...
The Berkeley Sockets API (1983)

- close()
- read()
- write()
...

- accept()
- bind()
- connect()
- getsockopt()
- listen()
- recv()
- select()
- send()
- setsockopt()
- socket()
...

• The Design and Implementation of the 4.3BSD Operating System
  • (but APIs/code first appeared in 4.2BSD)

• Now universal TCP/IP (POSIX, Windows)

• Kernel-resident network stack serves networking applications via system calls

• Reuses file-descriptor abstraction
  • Same API for local and distributed IPC
  • Simple, synchronous, copying semantics
  • Blocking/non-blocking I/O, select()

• Multi-protocol (e.g., IPv4, IPv6, ISO, ...)
  • TCP-focused but not TCP-specific
  • Cross-protocol abstractions and libraries
  • Protocol-specific implementations
  • “Portable” applications
BSD network-stack principles (1980s-1990s)

Multi-protocol, packet-oriented network research framework:

- **Object-oriented**: multiple protocols, socket types, but one API
  - **Protocol-independent**: streams vs. datagrams, sockets, socket buffers, socket addresses, network interfaces, routing table, packets
  - **Protocol-specific**: connection lists, address/routing specialization, routing, transport protocol itself – encapsulation, decapsulation, etc.

- **Packet-oriented**:
  - Packets and packet queueing as fundamental primitives
  - Best effort: If there is a failure (overload, corruption), drop the packet
  - Work hard to maintain packet source ordering
  - Differentiate ‘receive’ from ‘deliver’ and ‘send’ from ‘transmit’
  - Heavy focus on TCP functionality and performance
  - Middle-node (forwarding), not just edge-node (I/O), functionality
  - High-performance packet capture: Berkeley Packet Filter (BPF)
FreeBSD network-stack principles (1990s-2010s)

All of the 1980s features and also ...

- **Hardware:**
  - Multi-processor scalability
  - NIC offload features (checksums, TSO/LRO, full TCP)
  - Multi-queue network cards with load balancing/flow direction
  - Performance to 10s or 100s of Gigabit/s
  - Wireless networking

- **Protocols:**
  - Dual IPv4/IPv6
  - Pluggable congestion control
  - Security/privacy: firewalls, IPSec, ...

- **Software model:**
  - Flexible memory model integrates with VM for zero-copy
  - Network-stack virtualisation
  - Userspace networking via netmap
Memory flow in hardware

- **Key idea:** follow the memory
  - Historically, memory copying is avoided due to instruction count
  - Today, memory copying is avoided due to cache footprint
- Recent Intel CPUs push and pull DMA via the LLC ("DDIO")
  - If we differentiate ‘send’ and ‘transmit’, ‘receive’ vs. ‘deliver’, is this a good idea?
  - ... it depends on the latency between DMA and processing
Memory flow in software

- Socket API implies **one software-driven copy** to/from user memory
  - Historically, zero-copy VM tricks for socket API ineffective

- Network buffers cycle through the slab allocator
  - Receive: allocate in NIC driver, free in socket layer
  - Transmit: allocate in socket layer, free in NIC driver

- **DMA performs second copy;** can affect cache/memory bandwidth
  - NB: what if packet-buffer working set is larger than the cache?
The mbuf abstraction

- **Unit of work allocation and distribution** throughout the stack
- mbuf chains represent in-flight packets, streams, etc.
  - Operations: alloc, free, prepend, append, truncate, enqueue, dequeue
  - Internal or external data buffer (e.g., VM page)
  - Reflects bi-modal packet-size distribution (e.g., TCP ACKs vs data)
- Similar structures in other OSes – e.g., skbuff in Linux
Send/receive paths in the network stack

Application
- recv()
- send()

System call layer
- recv()
- send()
- sosend()
- sbappend()
- tcp_send()
- tcp_output()
- ether_output()
- em_start()
- em_entr()

Socket layer
- recv()
- sbappend()
- tcp_reass()
- tcp_input()

TCP layer
- tcp_reass()
- tcp_input()
- tcp_send()
- tcp_output()

IP layer
- ip_input()
- ip_output()

Link layer
- ether_input()
- ether_output()

Device driver
- em_intr()
- em_start()
- em_entr()

L41 Lecture 3 – The Network Stack
Forwarding path in the network stack

IP layer
- IP forward (ip_forward())
- IP output (ip_output())
- IP input (ip_input())

Link layer
- Ether output (ether_output())
- Ether input (ether_input())
- EM intr (em_intr())

Device driver
- EM start (em_start())
- EM entr (em_entr())

NIC
Work dispatch: input path

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Linker layer + driver</th>
<th>IP</th>
<th>TCP + Socket</th>
<th>Socket</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Receive, validate checksum</td>
<td>Validate checksum, strip IP header</td>
<td>Look up socket</td>
<td>Reassemble segments, deliver to socket</td>
<td>Data stream to application</td>
</tr>
<tr>
<td></td>
<td>Interpret and strips link layer header</td>
<td>Validate checksum, strip TCP header</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Deferred dispatch**: ithread $\rightarrow$ netisr thread $\rightarrow$ user thread
- **Direct dispatch**: ithread $\rightarrow$ user thread
  - Pros: reduced latency, better cache locality, drop early on overload
  - Cons: reduced parallelism and work placement opportunities
Work dispatch: output path

- Fewer deferred dispatch opportunities implemented
  - (Deferred dispatch on device-driver handoff in new iflib KPIs)
- Gradual shift of work from software to hardware
  - Checksum calculation, segmentation, ...

L41 Lecture 3 – The Network Stack
Work dispatch: TOE input path

- Kernel provides socket buffers and resource allocation
- Remainder, including state, retransmissions, etc., in NIC
- But: two network stacks? Less flexible/updateable structure?
  - Better with an explicit HW/SW architecture – e.g., Microsoft Chimney

L41 Lecture 3 – The Network Stack
Netmap: a novel framework for fast packet I/O
Luigi Rizzo, USENIX ATC 2012 (best paper).

- Map NIC buffers directly into user process memory
- Not the sockets API: Zero copy to/from application
- System calls initiate DMA, block for NIC events
- Packets can be reinjected into normal stack
- Ships in FreeBSD; patch available for Linux
- Userspace network stack can be **specialised** to task (e.g., packet forwarding)
Network stack specialisation for performance

- 30 years since the network-stack design developed
- Massive changes in architecture, micro-architecture, memory...
  - Optimising compilers
  - Cache-centered CPUs
  - Multiprocessing, NUMA
  - DMA, multiqueue
  - 10 Gigabit/s Ethernet
- Performance lost to ‘generality’ throughout stack
- Revisit fundamentals through clean-slate stack
- Orders-of-magnitude performance gains
The Transmission Control Protocol (TCP)

TCP principles and properties

- Network may delay, (reorder), drop, corrupt packets
- TCP: Reliable, ordered, stream transport protocol over IP
  - Three-way handshake: SYN / SYN-ACK / ACK (mostly!)
  - Sequence numbers ACK’d
  - Round-Trip Time (RTT) measured to time out loss
  - Data retransmitted on loss
  - Flow control via advertised window size in ACKs
  - Congestion control (‘fairness’) detects congestion via loss
TCP congestion control and avoidance

- 1986 Internet CC collapse
  - 32Kbps → 40bps
- Van Jacobson, SIGCOMM 1988
  - Don’t send more data than the network can handle!
  - Conservation of packets via ACK clocking
  - Exponential retransmit timer, slow start, aggressive receiver ACK, and dynamic window sizing on congestion
- ECN (RFC 3168), ABC (RFC 3465), Compound (Tan, et al, INFOCOM 2006), Cubic (Rhee and Xu, ACM OSR 2008)
TCP time/sequence graphs

- Extracted from TCP packet traces (e.g., via `tcpdump`)
- Visualize windows, congestion response, buffering, RTT, etc:
  - X: Time
  - Y: Sequence number
- We can extract this data from the network stack directly using Dtrace
  - Allows correlation/plotting with respect to other variables / events
## Evolving BSD/FreeBSD TCP implementation

<table>
<thead>
<tr>
<th>Year</th>
<th>Version</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1983</td>
<td>4.2BSD</td>
<td>BSD sockets, TCP/IP implementation</td>
</tr>
<tr>
<td>1986</td>
<td>4.3BSD</td>
<td>VJ/Karels congestion control</td>
</tr>
<tr>
<td>1999</td>
<td>FreeBSD 3.1</td>
<td>sendfile(2)</td>
</tr>
<tr>
<td>2000</td>
<td>FreeBSD 4.2</td>
<td>TCP accept filters</td>
</tr>
<tr>
<td>2001</td>
<td>FreeBSD 4.4</td>
<td>TCP ISN randomisation</td>
</tr>
<tr>
<td>2002</td>
<td>FreeBSD 4.5</td>
<td>TCP SYN cache/cookies</td>
</tr>
<tr>
<td>2003</td>
<td>FreeBSD 5.0-5.1</td>
<td>IPv6, TCP TIMEWAIT state reduction</td>
</tr>
<tr>
<td>2004</td>
<td>FreeBSD 5.2-5.3</td>
<td>TCP host cache, SACK, fine-grained locking</td>
</tr>
<tr>
<td>2008</td>
<td>FreeBSD 6.3</td>
<td>TCP LRO, TSO</td>
</tr>
<tr>
<td>2008</td>
<td>FreeBSD 7.0</td>
<td>T/TCP removed, socket-buffer autosizing</td>
</tr>
<tr>
<td>2009</td>
<td>FreeBSD 7.1</td>
<td>Read-write locking, full TCP offload (TOE)</td>
</tr>
<tr>
<td>2009</td>
<td>FreeBSD 8.0</td>
<td>TCP ECN</td>
</tr>
<tr>
<td>2012</td>
<td>FreeBSD 9.0</td>
<td>Pluggable TCP congestion control, connection groups</td>
</tr>
</tbody>
</table>

- Which changes have protocol-visible effects vs. only code?
Lect. 5 - Send/receive paths in the network stack

- Application
- System call layer
  - System call layer functions
  - send
    - sosend
    - sbappend
- Socket layer
  - Socket layer functions
  - recv
    - soreceive
    - sbappend
  - send
    - sosend
    - sbappend
- TCP layer
  - TCP layer functions
  - tcp_reass
  - tcp_input
  - tcp_send
  - tcp_output
- IP layer
  - IP layer functions
  - ip_input
  - ip_output
- Link layer
  - Link layer functions
  - ether_input
  - ether_output
- Device driver
  - Device driver functions
  - em_intr
  - em_start
  - em_ent
Data structures – sockets, control blocks

Socket and Socket Buffers
- socket
- so_pcb
- so_proto
- Listen state, accept filter
- Receive socket buffer
- Send socket buffer

Internet Protocol Control Blocks
- inpcb
  - inp_pppcb
  - List/hash entries
  - IP/port 4-tuple
  - IP options
  - Flow/RSS state

TCP Protocol Control Blocks
- tcpcb
  - Reassembly Q
  - Timers
  - Sequence state
  - Common CC state
  - Per-CC state
  - SACK state
  - TOE state

Protocol Description
...

L41 Lecture 3 – The Network Stack
Denial of Service (DoS) – state minimisation

- Yahoo!, Amazon, CNN taken down by SYN floods in February 2000
- D. Borman: **TCP SYN cache** – minimise state for new connections
- D. Bernstein: **SYN cookies** – eliminate state entirely – at a cost
- J. Lemon: **TCP TIMEWAIT reduction** – minimise state during close
- J. Lemon: **TCP TIMEWAIT recycle** – release state early under load

*Figure 3: Time needed to connect() to remote system.*
TCP connection lookup tables

- Global list of connections for monitoring (e.g., netstat)
- Connections are installed in a global hash table for lookup
- Separate (similar) hash table for port-number allocations
- Tables protected by global read-write lock as reads dominate
  - New packets are more frequent than new connections
Lect. 5 - Work dispatch: input path

- **Deferred dispatch**: ithread $\rightarrow$ netisr thread $\rightarrow$ user thread
- **Direct dispatch**: ithread $\rightarrow$ user thread
  - **Pros**: reduced latency, better cache locality, drop early on overload
  - **Cons**: reduced parallelism and work placement opportunities
• Network bandwidth growth > CPU frequency growth
• Locking overhead (space, contention) substantial
  • Getting ‘speedup’ is hard!
• Evaluate different strategies for TCP processing parallelisation
  • Message-based parallelism
  • Connection-based parallelism (threads)
  • Connection-based parallelism (locks)
• Coalescing locks over connections:
  • reduces overhead
  • increases parallelism
FreeBSD connection groups, RSS

- **Connection groups** blend MsgP and ConnP-L models
  - PCBs assigned to group based on 4-tuple hash
  - Lookup requires group lock, not global lock
  - Global lock retained for 4–tuple reservation (e.g., setup, teardown)

- **Problem:** have to look at TCP headers (cache lines) to place work!

- **Microsoft:** NIC **Receive-Side Scaling (RSS)**
  - Multi-queue NICs deliver packets to queues using hash of 4-tuple
  - Align connection groups with RSS buckets / interrupt routing
Performance: dispatch model and locking

Varying dispatch strategy – bandwidth

- 2010 8-core x86 multicore server
- TCP LRO disabled (maximise PPS)
- Configurations:
  1. 1 queue (no dispatch), 1 thread on 1 core
  2. 1 queue (SW dispatch), 8 threads on 8 cores
  3. 8 queues (HW dispatch), 8 threads on 8 cores
Architectural $\rightarrow$ micro-architectural + I/O optimisation

• Hardware, software, protocol co-design causes change to optimisation approach over time:
  • Counting instructions $\rightarrow$ counting cache misses
  • Reducing lock contention $\rightarrow$ cache-line contention
  • Adding locking $\rightarrow$ identifying new parallelism
  • Work ordering, classification, and distribution
  • Vertically integrated distribution and affinity
  • NIC offload of further protocol layers, crypto
  • DMA/cache interactions

• Convergence of networking and storage technologies?
Labs 4 + 5: TCP

- From abstract to concrete understanding of TCP
  - Use tools such as tcpdump and DUMMYNET
  - Explore effects of latency on TCP performance

- Lab 4 – TCP state machine and latency
  - Measure the TCP state machine in practice
  - Start looking at TCP latency vs. bandwidth (DUMMYNET)
  - At what transfer sizes are different latencies masked?

- Lab 5 – TCP congestion control
  - Draw time-sequence-number diagrams
  - Explore OS buffering strategies
  - Explore slow-start vs. steady state as latency changes
  - Explore OS and microarchitectural performance interactions
L41 lecture wrap-up

• Goal: Deeper understanding of OS design and implementation
  • Evolving architectural and microarchitectural foundations
  • Evolving OS design principles
  • Evolving tradeoffs in OS design
  • Case study: The process model
  • Case study: Network-stack abstractions
  • Quick explorations of past and current research

• Goal: Gain practical experience analysing OS behaviour

• Goal: Develop scientific analysis and writing skills

• Feel free to get in touch to learn more!