L41 - Lecture 5: The Network Stack (1)

Dr Robert N. M. Watson

27 April 2015
Reminder: where we left off in Lent Term

Long, long ago, but in a galaxy not so far away:

- Lecture 3: The Process Model (1)
- Lecture 4: The Process Model (2)
- Lab 1: I/O performance
- Lab 2: IPC buffer size and probe effect
- Lab 3: Micro-architectural effects of IPC

Explore several (implied, and it turns out, incorrect) hypotheses:

- Larger I/O and IPC buffer sizes amortize system-call overheads, improving application performance
- Micro-architecture is irrelevant
- The probe effect doesn’t matter in real workloads
A key OS function: networking

- Communication between distributed computer systems
  - *Local-area networking* (LANs) and *wide-area networking* (WANs)

- A network stack provides:
  - Sockets API and extensions
  - Interoperable, feature-rich, high-performance protocol implementations (e.g., IPv4, IPv6, ICMP, UDP, TCP, SCTP, ...)
  - Device drivers for Network Interface Cards (NICs)
  - Monitoring and management interfaces (BPF, `ioctl`)
  - Plethora of support libraries (e.g., DNS)

- 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/s, then 10-Gigabit/s Ethernet; 802.11, GSM data
  - 2010s: Large-scale deployment of IPv6; 40/100-Gigabit/s Ethernet

- Vanishing technologies: UUCP, IPX/SPX, ATM, token ring, SLIP, ...
The Berkeley Sockets API

- Universal API for TCP/IP (POSIX, Windows, ...)
- *The Design and Implementation of the 4.3BSD Operating System* (although appeared in 4.2)
- Kernel-resident network stack serving userspace networking applications via system calls
- Reuse 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: ‘protocol’, ‘socket address’, ‘stream’, ‘datagram’, ...
- NB: ‘socket’ in BSD API is not the same as a ‘socket’ in the TCP RFC
Early BSD network-stack design principles

- Framework for network research
- Object-oriented: multiple protocols, multiple socket types, 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.
- Fundamentally packet-oriented:
  - Packets and packet queueing as fundamental primitives
  - 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 design principles

All of the 1980s features and also ...

- 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
- Dual-IPv4/IPv6
- Security/privacy: firewalls, IPsec, ...
- Flexible memory model integrates with VM for zero-copy
- Full network-stack virtualisation
- Userspace networking via netmap
Memory flow in hardware

Key idea: follow the memory

- Historically, memory copying in stack avoided due to CPU cost
- Today, memory copying in stack avoided due to cache footprint
- Recent Intel CPUs push and pull DMA via the LLC (“DDIO”)
- NB: if we differentiate ‘send’ and ‘transmit’, is this a good idea?
Memory flow in software

- Socket API implies one 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

- mbuf chains represent in-flight packets, streams, etc.
  - Also a unit of *work* allocation throughout the stack
  - mbufs reference an in-**mbuf** or external buffer (e.g., VM page)
  - Bi-modal packet size distribution; e.g., TCP ACKs vs. data
  - Common operations: prepend, append, truncate at front or end
- Similar abstractions in other OSes – e.g., skbuff in Linux
Local send/receive paths in the network stack

Network-stack design
Network-stack design

Forwarding path in the network stack

- IP layer
  - ip_forward()
  - ip_input()
  - ip_output()
- Link layer
  - ether_input()
  - ether_output()
- Device driver
  - em_intr()
  - em_start()
  - em_entr()

NIC
Network-stack design

Work dispatch: input path

- Deferred dispatch - *ithread* -> *netisr thread* -> *user thread*
- Now: direct dispatch - *ithread* -> *user thread*
  - Pros: reduced latency, better cache locality, drop overload early
  - Cons: reduced parallelism and work placement opportunities
Network-stack design

## Work dispatch: output path

<table>
<thead>
<tr>
<th>Userspace</th>
<th>Kernel</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application</strong></td>
<td><strong>Socket</strong></td>
<td><strong>Device</strong></td>
</tr>
</tbody>
</table>
| Data stream from application | Kernel copies in data to mbufs + clusters | CPU
| 256 | 2k, 4k, 9k, 16k | ▶ Checksum calculation, segmentation, ...
| ▶ Fewer deferred dispatch opportunities implemented | TCP segmentation, header encapsulation, checksum | ▶ Gradual shift of work from software to hardware
| MSS | MSS | ▶ But no fundamental changes to output path |
| IP | Ethernet frame encapsulation, checksum | |
| Link layer + driver | | |
| Ethernet frame encapsulation, insert in descriptor ring |

Dr Robert N. M. Watson

L41 - Lecture 5: The Network Stack (1) 27 April 2015 13 / 19
Network-stack design

Work dispatch: TOE input path

- Full TCP offload: kernel provides socket buffers and resource allocation
- Remainder, including state, retransmits, reassembly, etc, in NIC
  - But: Two network stacks? Less flexible/updateable structure?
- Better with an explicit SW architecture – e.g., Microsoft Chimney?
netmap: a novel framework for fast packet I/O

Luigi Rizzo, USENIX ATC 2012 (best paper).

- Map NIC buffers directly into user process memory
- Zero copy to application
- Userspace network stack can be specialized to task (e.g., packet forwarding)
- System calls initiate DMA, block for NIC events
- Packets can be reinjected into normal stack
- Ships in FreeBSD, patch available for Linux
Network Stack Specialization for Performance
Ilias Marinos, Robert N.M. Watson, Mark Handley, SIGCCOMM 2014.

- 30 years since current network-stack architecture principles developed
- Massive changes in compilers, architecture, micro-architecture, memory, buses, NICs
  - Optimising compilers
  - Cache-centered CPUs
  - Multiprocessing, NUMA
  - DMA, multiqueue
  - 10 Gigabit/s Ethernet
- Revisit fundamentals through clean-slate stack
Next time: Socket buffers and TCP

- The socket buffer abstraction
- A (very) little about the TCP implementation
- The TCP state machine
- TCP flow and congestion control
- The final two labs
Lab 1 - I/O - Buffer size vs. throughput

```
./io-static -r -b <bytes> -t 16777216 /data/iofile
```

- unmodified benchmark
- bzero() buffer at start

Graph showing median KBytes/s bandwidth reading 16M file vs. bytes-buffer-size argument to read(2). The graph includes data points for different buffer sizes ranging from 20000 to 180000 bytes, with a peak around 100000 bytes. The graph also indicates the impact of different cache levels, such as 32x4K TLB entries, 32K L1, and 256K L2, on throughput.
Lab 1 - I/O - Static/dynamic linking vs. throughput

Static vs. Dynamic Binary Performance by File Size
Fixed Buffer Size (2M)

Median Execution Time (us)

Ratio of Medians

Ratio falls below 1.05 at roughly 8MB file, and 1.01 at roughly 32M
Ratio of 1.0: dynamic = static

Dynamic/Static

File Size in Bytes