Chapter 21

The Network Stack

In comparison with some other parts of OS design, networking has very little (if any) basis in formalism or algorithms – yet (some of us are working on this . . ).

Definition 21.1 (Network stack). The network stack is the component of the OS which handles all network I/O, including packet transmit and receive, multiplexing and demultiplexing, and other kinds of protocol processing.

Remarks:

- Not all network protocols are handled by the OS network stack: some (such as HTTP) will be handled as part of the application. The division of responsibility for a complete network communication channel between an application and the OS network stack is somewhat arbitrary, but typically happens above the transport layer (e.g. TCP).
- Within the OS network stack, protocol processing can (and does) occur in many different places:
  - The network hardware, such as the Network Interface Card (NIC).
  - The first-level interrupt handlers for the NIC.
  - The rest of the NIC driver bottom-half, e.g. the DPCs.
  - The driver top-half, and other kernel code invoked from user space with a system call.
  - Libraries linked with the application.

When a packet is received by a NIC, the data generally traverses all these layers in order.
- Similarly, when data is to be sent from a process over the network, all these layers can be involved in reverse.
- In addition to these components, there also
  - System daemons, and
  - Utility programs
  - which are used on longer timescales to maintain the state of the network stack, and form part of the control plane.

21.1 Network stack functions

The network stack’s purpose is to move data between user programs on different machines. Within a single end-system, getting this done involves converting between the user’s data and the set of network protocols being used to communicate it.

This can be broken down into various functions, all of which happen at multiple places in the stack:

Definition 21.2 (Multiplexing). Multiplexing is the process of sending packets from multiple connections at one layer in the protocol stack down a single connection at a lower later. Demultiplexing is the reverse process: taking packets received on a connection at one layer, and directing each one to the appropriate channel in an upper layer.

Definition 21.3 (Encapsulation). While multiplexing refers to sharing a single lower-level channel among multiple higher-level connections, encapsulation is the mechanism by which is this usually achieved: by wrapping a packet on one channel as the payload of a lower-layer connection, adding a header and/or trailer in the process. De-encapsulation is the reverse: interpreting the payload of one packet as a packet for a different, higher-layer protocol.

Remarks:

- Multiplexing and demultiplexing, and encapsulation and de-encapsulation, are close related but not the same thing. Multiplexing generally requires some kind encapsulation so as to be able to distinguish packets from different connections, but encapsulation may not imply multiplexing.
- Demultiplexing is frequently the most important factor in the performance of a workload over a particular network stack. Fully demultiplexing an Ethernet packet to an application socket may require 10 or 20 different stages of de-encapsulation, if done naively.

Definition 21.4 (Protocol state processing). The networking stack needs to do more than simply move packets between the network interface and main memory. In addition, it must maintain, and execute, state machines for some (though not all) network protocols. Such protocol state processing not only means that the network stack needs to maintain more state than simply the set of active connections, but it also may need to generate new packets if the state machine so requires.

Example 21.5. TCP is a good example of protocol state processing, and in UNIX (as in most other OSes) it is performed in the kernel to prevent abuse of the protocol by unprivileged applications. TCP state for a given TCP connection in the kernel, generally held in a data structure called the TCP Control Block, is much more than simply the state you are familiar with from the networking course. It includes the congestion and flow control windows, a buffer of all unacknowledged data it might need to resend, plus a collection of timers that trigger the state transitions that are not initiated by packets arriving.
Definition 21.6 (Buffering and data movement). The network stack also needs to move packet data through the system, both buffering (holding the data until it is ready to be consumed) and transferring it between different protocol processing elements.

Remarks:

- Copying packets between every protocol processing stage in the networking stack is clearly inefficient, and so implementations try to avoid copying wherever possible. However, encapsulation and de-encapsulation require care in data representation: adding a header to the front of a packet without copy any data is challenging.

In addition, most network stacks also perform routing and forwarding, which are discussed below.

21.2 Header space

Definition 21.7 (Header space). The header space is an abstract vector space which represents the set of all possible headers of a packet.

Remarks:

- This definition is a bit under-specified: the dimensions of the header space might be each bit in a packet header, or something more abstract (port numbers, source/dest addresses, etc.).

- Even so, any network packet occupies a point in header space.

- Each node in the protocol graph corresponds to a region of header space.

- The set of packets than can be received at a NIC potentially occupies the whole header space, but each demultiplexing step in the protocol stack reduces the sub-volume of the header space that a packet must lie in.

- Each network connection, such as a socket, corresponds to two sub-volumes of header space: which we might call the receive set and the transmit set. Any packet to be transmitted has to lie inside the transmit set when it leaves the NIC, and any packet that is received on the socket must lie in the receive set.

- Packet demultiplexing therefore is the operation of identifying the set of receive sets each received packet lies within, and delivering the packet payload to the socket corresponding to each of these receive sets.

21.3 Protocol graphs

Definition 21.8 (Protocol graph). The protocol graph of a network stack is a directed-graph representation of the forwarding and multiplexing rules at any point in time in the OS. Nodes in the protocol graph represent a protocol acting on a communications channel, and perform encapsulation or decapsulation, and possibly multiplexing or demultiplexing.

Example 21.9. Nodes in the protocol graph might include:

- Demultiplexing an LLC/SNAP packet based on whether the protocol field is IP, or AppleTalk, or . . .
- Demultiplexing UDP packets based on the destination IP address and port
- Demultiplexing TCP packets based on the four-tuple (destination address, destination port, source address, source port).
- Processing a single, bidirectional TCP connection
- Responding to ICMP echo requests
- Encapsulating messages sent via a UDP socket with the correct UDP header

And so on.

Remarks:

- If a node has multiple outgoing edges, it’s probably demultiplexing packets. If it has multiple inbound edges, it’s multiplexing.

- People have constructed network stacks explicitly this way, notably the x-kernel [HP88], but it’s also an appropriate model of any network stack in the abstract.

- One advantage of the protocol graph over a traditional layered model is that it handles multiple connections at the same layer.

- More importantly, it can be cyclic: for example, an IP packet might be received, demultiplexed into a TCP flow, decrypted using TLS, decapsulated via a tunneling protocol like PPTP, and result in a new set of IP packets to be demultiplexed.

21.4 Network I/O

Let’s work up the network stack, starting at the hardware.

We’ve already seen how a NIC operates at the lowest level in Chapter 10: packets that are received are copied into buffers supplied by the OS and enqueued onto a descriptor queue, and these filled buffers are then returned to the OS using the same, or possibly a different, queue.

Similarly, packets to be sent are enqueued on a descriptor queue, and the NIC notified. When a packet has been sent over the network, its buffer is returned
21.5. DATA MOVEMENT INSIDE THE NETWORK STACK

Data movement inside the network stack is of interest. Example 21.12 (BSD mbuf structures). To simplify somewhat, a Unix mbuf contains the following fields:

```c
struct mbuf {
    struct mbuf *m_next; /* next mbuf in chain */
    struct mbuf *m_nextpkt; /* next packet in list */
    char *m_data; /* location of data */
    int32_t m_len; /* amount of data in this mbuf */
    int32_t m_type; /* type of data in this mbuf */
    int32_t m_flags:24; /* flags; see below */
    char m_dat[0];
};
```

How is this used?

A single mbuf describes b.m_len bytes of data, starting in memory at address b.m_data. This data might be stored in the mbuf itself, in the array b.m_dat, or alternatively somewhere else in memory; the b.m_type field specifies which case this is.

An mbuf chain represents a single contiguous packet, using non-contiguous areas of memory. An mbuf chain is a singly-linked list of mbufs chained with the m_next field.

Moreover, packets themselves can be hooked together in lists or queues using the m_nextpkt field.

There are a whole host of utility functions in the BSD kernel for creating, filling, manipulating, duplicating, coalescing, freeing, and doing all kinds of other things to mbufs.

Remarks:

- There’s a reason for this level of complexity, and it’s to do with avoiding copying data, and the need for encapsulation. If a user program wants to send a packet, a packet descriptor is created to wrap the payload the user has supplied. To put a header on the packet, it’s easier (in the BSD case above) to add a new mbuf to the front of the list which holds the new header, rather than copying the entire packet to a bigger buffer with room for the new header.

- Similarly, on receive, it’s easier to strip the header from a packet when de-encapsulating it by simply bumping the m_data field.

- The main goal here is to avoid excessive copying of packet data in the kernel – it’s expensive and inefficient.

21.6 Protocol state processing

Protocol state processing generally also happens in the “bottom half” of the network stack, even though it is generally independent of the particular NIC device driver. To understand why, consider TCP.

Example 21.13 (TCP protocol state processing). The code implementing a single TCP connection (managing a single TCP control block) has to run in response to many different external events: the user sending a segment, or the network receiving data on the connection, but also timers expiring, or acknowledgments received by the network, etc. Many of these events don’t involve the user program at all (such as acknowledging data as soon as it is received).
21.7. TOP-HALF HANDLING

What’s more, these events also require TCP to send packets (such as acknowledgments or retransmissions) that the user will never see. These can be time-critical: TCP estimates the size of its windows based on measuring round-trip time, which includes the time taken for TCP code to execute when a packet is received.

For this reason, if TCP is to run in the “top half” (i.e. in code executed by the user program, or in the kernel but invoked by the user program via a syscall), it has to be scheduled to run at all kinds of events not scheduled by the user program. UNIX-like operating systems avoid this problem by running most of TCP in the bottom half, as a set of DPCs.

21.7 Top-half handling

Moving up the stack further, we come to the top half: that invoked by user programs. You’re already familiar with the common “sockets” interface: bind(), listen(), accept(), connect(), send(), recv(), etc.

Some protocol processing happens in the kernel directly as a result of top half invocations, but for the most part the top half is concerned with copying network payload data and metadata (if the user requests it) between queues of protocol descriptors in the kernel (attached to sockets) and user-space buffers.

21.8 Performance issues

So far, so good. But much time does the OS have to process a packet with a modern network?

Example 21.14 (10 Gb/s Ethernet performance). At full line rate for a single 10 Gb/s port, we receive about 1 GB (gigabyte) per second of packet data. This corresponds to about 700,000 full-size Ethernet frames per second, rather more if the frames are smaller.

At 2 GHz, this means the OS has to process a packet in under 3000 cycles. This includes IP and TCP checksums, TCP window calculations and flow control, and copying the packet to user space. Note that this is a 1500-byte packet, and we can copy about 4 bytes per cycle through CPU registers. That’s 400 cycles gone right there, and we need some left over for the user program to run.

It gets worse. An L3 cache miss (64-byte lines) is about 200 cycles, which means we can afford at most 10 or so cache misses per packet. However, on most machines, DMA transfers from the NIC mean that the processor cache is cold for the packet.

Furthermore, interrupt latency in a typical PC is 500 cycles, so we’re going to have trouble reacting fast enough if we take an interrupt on each packet. This is without considering the cost in latency for kernel entry and exit, hardware register access (often hundreds of cycles), context switch overhead, the cost of enqueueing and dispatching a DPC, etc.

We also have to send packets as well as receive them.

This example is a single-port 10 Gb/s Ethernet card, but vendors are selling dual-port cards that run at 200 Gb/s today.

Clearly, the network stack design we have described so far will be unable to handle this traffic.

21.9 Network hardware acceleration

The solution is to put more functionality into hardware. As with much of networking, there are few agreed-upon terms here but a huge variety of hardware technology features, so treat this list as a broad overview.

Definition 21.15 (Polling). Instead of the conventional interrupt-driven descriptor queues, a network receive queue can be served by a processor polling it continuously.

Remarks:
- Polling eliminates the overhead of interrupts, context switches, and even kernel entry and exit if there is a way to access the queues from user space.
- This is an old idea, but it has recently come into vogue through schemes like NetMap [Riz12] and Intel’s Data Plane Development Kit [dpd18]. There is now a direct polling interface to the network in Linux.
- This, of course, requires a dedicated processor core to spin forever back to the interrupt-driven model, and resort to polling at high-load – for example, microkernel-based drivers have done this for some time.
- Even polling, however, is insufficient to handle modern high-speed networks. For one thing, it is not clear how to scale this to multiple cores. If the network stack for a given NIC has to go through a single core, Amdahl’s Law will fundamentally limit the performance of networked programs.

Definition 21.16 (Multiple queues). Modern NICs support multiple send and receive queues per port. Received packets are demultiplexed in hardware based on a set of flow tables which determine which descriptor queue to put each packet onto. Similarly, multiple transmit queues are multiplexed onto the network physical port.

Remarks:
- The number of queues supported these days ranges from 2 (on cheap cards), to 64 (on fairly cheap cards), to several thousand (on fancy hardware).
- There are plenty of criteria for demultiplexing flows. A typical table maps IP 5-tuples in the packet (sometimes with “wildcard” entries) to the queues. Other pattern matching is possible, for example the Intel X559 has a “SYN filter”, which can redirect TCP connection setup packets to a different queue.
- On transmit, the card typically has a configurable scheduling policy for picking which non-empty transmit queue to pick a packet from next.
21.9. NETWORK HARDWARE ACCELERATION

Definition 21.17 (Flow steering). Sending received packets to the right receive queue is only part of the solution. **Flow steering** not only picks a receive queue based on the network flow (e.g., TCP connection) that the packet is part of, but can send an interrupt to a specific core that is waiting for that packet.

Remarks:
- Given the overhead of moving packet data from one core’s cache to another, it’s quite important in many cases that the first core to find out about a new packet is the one running the thread that is waiting to receive it.

Definition 21.18 (Receive-side scaling). **Receive-side scaling (RSS)** uses a deterministic hash function on the packet header to balance flows across receive queues and interrupts.

Remarks:
- Flow steering specifies for each flow (up to the size of the hardware table . . . ) where to send it. RSS doesn’t need a table, but attempts to balance lots of flows across lots of cores.
- The assumption here is that one core is just as good as any other at handling a flow, and as long as it’s the same core for all packets in a flow, cache lines will migrate to that core.
- RSS allows the network stack performance to scale with the number of cores (assuming it is written correctly . . . ) by removing the multiplexing function—which is a serialization point—from software.

Definition 21.19 (TCP Chimney Offload). **TCP chimney offload**, sometimes called **partial TCP offload**, allows the entire state machine for a TCP connection—once it has been established—to be pushed down the hardware, which will then handle timers, acknowledgments, retries, etc. and simply deliver in-order TCP segments to the kernel.

Remarks:
- Chimney offload is a limited case of a full TCP Offload Engine (TOE), which moves even more of the TCP stack onto the NIC.

Definition 21.20 (Remote DMA). **Remote Direct Memory Access** or **RDMA** is a completely different set of network protocols and hardware implementations. RDMA supports Ethernet-style descriptor rings for messages, but also supports (hence the name) so-called **one-sided operations** which allow main memory on a machine to be written and read directly over the network without involving the host CPU: the NIC receives packets requesting such an operation, executes it itself, and returns the results.

Definition 21.21 (Forwarding). **Packet forwarding** is the process of deciding, based on a packet and the interface on which the packet was received, which interface to send the packet out on.

Definition 21.22 (Routing). **Packet routing** is the process of calculating rules to determine how all possible packets are to be forwarded.

Remarks:
- Forwarding needs to be fast, since any latency adds to the end-to-end delay of the packet concerned. Consequently, it’s done at the lower levels of the stack, as close to the hardware as possible (unless the hardware itself performs forwarding).
- As in a hardware-based router, forwarding is performed according to a set of tables called the FIB.

Definition 21.23 (Forwarding information base). The **forwarding information base** or FIB in a router is the set of data structures which are traversed for each packet received to determine what actions to perform on it, such as sending it out on a port or putting it into a memory buffer.
Remarks:

- The FIB is the result of the routing calculation. Routing may be a purely local computation, but might involved a routing protocol which itself needs to send and receive network messages to exchange information with other routers.

- For this reason, routing in an OS stack usually happens in user space, either in a routing daemon (sometimes called routed) or by manual configuration utilities.

Example 21.24. Routing in Linux can be done manually using the `ip route` command - type `ip route show` to show the machine’s routing table. If you want to run a routing daemon to talk BGP, OSPF, or some other routing protocol, packages like `quagga` supply suitable daemons.

Remarks:

- Forwarding a packet found on a receive queue essentially involves reading its header to classify it, then using this information to transfer the packet to one or more transmit queues to be sent.

- This is essentially the same operation as demultiplexing a received packet to an appropriate application. For this reason, forwarding code is generally the same as the lower-level protocol demultiplexing code.

Bibliography

