CSCI-1680
Layering and Encapsulation

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, David Mazières, Phil Levis, John Jannotti
• HW0: Due TODAY by 11:59pm
• Snowcast is out!
  – We will talk about sockets today
• Snowcast milestone due by Friday, 11:59pm
  – Partial server/client: we will grade your latest commit before deadline
  – Design questions about server: submit on gradescope
• Waitlist: I will admit another batch after class
Topics for Today

• Layering and Encapsulation
• Intro to IP, TCP, UDP
• Primer on sockets
Traceroute map of the Internet, ~5 million edges, circa 2003. opte.org
OPTE Internet map, 1997-2021:  https://youtu.be/DdaElt6oP6w
How do we make sense of all this?

- Very large number of computers
- Incredible variety of technologies
  - Each with very different constraints
- No single administrative entity
- Evolving demands, protocols, applications
  - Each with very different requirements!
Layering

Separation of concerns
• Break problem into separate parts
• Solve each one independently
• Tie together through common interfaces: abstraction
• Encapsulate data from the layer above inside data from the layer below
• Allows independent evolution

Protocol layering

<table>
<thead>
<tr>
<th>Application</th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Link Layer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An analogy

• How to deliver a letter?
Layers

• Application – what the users sees, e.g., HTTP
• Presentation – crypto, conversion between representations
• Session – can tie together multiple streams (e.g., audio & video)
• Transport – demultiplexes, provides reliability, flow and congestion control
• Network – sends packets across entire net., using routing
• Data Link – sends frames, handles media access across links
• Physical – sends individual bits
OSI Reference Model

One or more nodes within the network

- **Application Protocol (L7)**
  - Application
  - Presentation
  - Session
  - Transport
  - Network
  - Data link
  - Physical

- **Transport Protocol (L4)**
  - Application
  - Presentation
  - Session
  - Transport
  - Network
  - Data link
  - Physical

- **Network Protocol (L3)**
  - Application
  - Presentation
  - Session
  - Transport
  - Network
  - Data link
  - Physical

- **Link-Layer Protocol (L2)**
  - Application
  - Presentation
  - Session
  - Transport
  - Network
  - Data link
  - Physical

One or more nodes in the network
Layers, Services, Protocols

- **Layer N**
  - Protocol: rules for communication within same layer
  - Service: abstraction provided to layer above
  - API: concrete way of using the service
  - Layer N uses the services provided by N-1 to implement its protocol and provide its own services
A more realistic picture

**Application**
Service: user-facing application. (eg. HTTP, SSH, …) Application-defined messages

**Transport**
Service: multiplexing applications
Reliable byte stream to other node (TCP),
Unreliable datagram (UDP)

**Network**
Service: move packets to any other node in the network
IP: Unreliable, best-effort service model

**Link**
Service: move frames to other node across link. (eg. Ethernet, Wifi, …)
May add reliability, medium access control

**Physical**
Service: move bits to other node across link
(Electrical engineering problem)
How/where to handle challenges?

• Can decide on how to distribute problems
  – What services at which layer?
  – What to leave out?
  – More on this later (End-to-end principle)

• Example: reliability
  – IP offers pretty crappy service, even on top of reliable links… why?
  – TCP: offers reliable, in-order, no-duplicates service. Why would you want UDP?

Other challenges: fairness, security, accountability…
IP as the Narrow Waist

- Many applications protocols on top of UDP & TCP
- IP works over many types of networks
- This is the “Hourglass” architecture of the Internet.
  - If every network supports IP, applications run over many different networks (e.g., cellular network)
Network Layer: Internet Protocol (IP)

- Used by most computer networks today
  - Runs over a variety of physical networks, can connect Ethernet, wireless, modem lines, etc.

- Every host has a unique 4-byte IP address (IPv4)
  - \textit{E.g.,} www.cs.brown.edu \(\rightarrow\) 128.148.32.110
  - The network knows how to route a packet to any address

- Need more to build something like the Web
  - Practical naming (DNS, \textit{eg.} google.com)
  - Application software (browsers, servers, etc.)
  - Demultiplexing within a host: which packets are for web browser, Skype, or the mail program?
Demultiplexing within a host

• Talking from host to host is great, but we want abstraction of inter-process communication
• Solution: encapsulate another protocol within IP
Transport: UDP and TCP

UDP and TCP: most popular protocols atop IP
- Both use 16-bit port number & 32-bit IP address
- Applications bind a port & receive traffic on that port

• UDP – User (unreliable) Datagram Protocol
  - Send packets to a port (… and not much else)
  - Sent packets may be dropped, reordered, even duplicated (but has corruption protection)

• TCP – Transmission Control Protocol
  - Provides illusion of reliable ‘pipe’ or ‘stream’ between two processes anywhere on the network
  - Handles congestion and flow control
Uses of TCP

• Most applications use TCP
  – Easier to program (reliability is convenient)
  – Automatically avoids congestion (don’t need to worry about taking down the network)

• Servers typically listen on well-known ports:
  – SSH: 22
  – SMTP (email): 25
  – HTTP (web): 80, 443
Uses of UDP

In general, when you have concerns other than a reliable “stream” of packets:

• When latency is critical (late messages don’t matter)
• When messages fit in a single packet
• When you want to build your own (un)reliable protocol!

Examples

• DNS (port 53)
• Streaming multimedia/gaming (sometimes)
A note on layering

Strict layering not required

- TCP/UDP “cheat” to detect certain errors in IP-level information like address
- Overall, allows evolution, experimentation
One more thing…

• Layering defines interfaces well
  – What if I get an Ethernet frame, and send it as the payload of an IP packet across the world?

• Layering can be recursive
  – Each layer agnostic to payload!

• Many examples
  – **Tunnels**: e.g., VXLAN is ETH over UDP (over IP over ETH again…)
  – Our IP assignment: IP on top of UDP “links”
Table 1: The packet formats in the example scenario. Different switch models may add different headers before sending out the mirrored packets, which further complicates the captured packet formats.

<table>
<thead>
<tr>
<th>Header Format</th>
<th>Mirrored Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETHERNET IPV4 ERSSPAN ETHERNET</td>
<td>IPV4 TCP</td>
</tr>
<tr>
<td>ETHERNET IPV4 ERSSPAN ETHERNET 802.1Q IPV4 TCP</td>
<td>IPV4 TCP</td>
</tr>
<tr>
<td>ETHERNET IPV4 ERSSPAN IPV4 UDP VXLAN ETHERNET</td>
<td>IPV4 TCP</td>
</tr>
<tr>
<td>ETHERNET IPV4 ERSSPAN IPV4 UDP VXLAN ETHERNET</td>
<td>IPV4 TCP</td>
</tr>
<tr>
<td>ETHERNET IPV4 GRE IPV4 UDP VXLAN ETHERNET</td>
<td>IPV4 TCP</td>
</tr>
<tr>
<td>ETHERNET IPV4 GRE IPV4 UDP VXLAN ETHERNET</td>
<td>IPV4 TCP</td>
</tr>
<tr>
<td>ETHERNET IPV4 ERSSPAN IPV4 GRE ETHERNET</td>
<td>IPV4 TCP</td>
</tr>
<tr>
<td>ETHERNET IPV4 GRE IPV4 GRE ETHERNET</td>
<td>IPV4 TCP</td>
</tr>
</tbody>
</table>

* This is just an example, do not worry about the details, or the specific protocols!

From: Yu et al., A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces, NSDI 2019
How do we use these protocols?
Using TCP/IP

• How can applications use the network?
• Sockets API.
  – Originally from BSD, widely implemented (*BSD, Linux, Mac OS X, Windows, …)
  – Important do know and do once
  – Higher-level APIs build on them
• After basic setup, much like files
Network sockets are file descriptors too

Datagram sockets: unreliable message delivery
  - With IP, gives you UDP
  - Send atomic messages, which may be reordered or lost
  - Special system calls to read/write: send/recv

Stream sockets: bi-directional pipes
  - With IP, gives you TCP
  - Bytes written on one end read on another
  - Reads may not return full amount requested, must re-read
# System calls for using TCP

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket</td>
<td>socket – make socket</td>
</tr>
<tr>
<td>bind* – assign address</td>
<td>bind – assign address, port</td>
</tr>
<tr>
<td>connect – connect to listening socket</td>
<td>listen – listen for clients</td>
</tr>
<tr>
<td>accept – accept connection</td>
<td></td>
</tr>
</tbody>
</table>

- This call to bind is optional, connect can choose address & port.
Socket Naming

• Recall how TCP & UDP name communication endpoints
  – IP address specifies host (128.148.32.110)
  – 16-bit port number demultiplexes within host
  – Well-known services listen on standard ports (e.g. ssh – 22, http – 80, mail – 25, see /etc/services for list)
  – Clients connect from arbitrary ports to well known ports

• A connection is named by 5 components
  – Protocol, local IP, local port, remote IP, remote port
  – TCP requires connected sockets, but not UDP
Dealing with Address Types

- All values in network byte order (Big Endian)
  - htonl(), htons(): host to network, 32 and 16 bits
  - ntohl(), ntohs(): network to host, 32 and 16 bits
  - Remember to always convert!

- All address types begin with family
  - sa_family in sockaddr tells you actual type

- Not all addresses are the same size
  - e.g., struct sockaddr_in6 is typically 28 bytes, yet generic struct sockaddr is only 16 bytes
  - So most calls require passing around socket length
  - New sockaddr_storage is big enough
Client Skeleton (IPv4 only)

```c
struct sockaddr_in {
    short sin_family; /* = AF_INET */
    u_short sin_port; /* = htons (PORT) */
    struct in_addr sin_addr;
    char sin_zero[8];
} sin;

int s = socket (AF_INET, SOCK_STREAM, 0);
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (13); /* daytime port */
sin.sin_addr.s_addr = htonl (IP_ADDRESS);
connect (s, (sockaddr *) &sin, sizeof (sin));
while ((n = read (s, buf, sizeof (buf))) > 0)
    write (1, buf, n);
```
int s = socket (AF_INET, SOCK_STREAM, 0);
struct sockaddr_in sin;
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (9999);
sin.sin_addr.s_addr = htonl (INADDR_ANY);
bind (s, (struct sockaddr *) &sin, sizeof (sin));
listen (s, 5);

for (;;) {
    socklen_t len = sizeof (sin);
    int cfd = accept (s, (struct sockaddr *) &sin, &len);
    /* cfd is new connection; you never read/write s */
    do_something_with (cfd);
    close (cfd);
}
Using UDP

- Call socket with SOCK_DGRAM, bind as before
- New calls for sending/receiving individual packets
  - `sendto(int s, const void *msg, int len, int flags, const struct sockaddr *to, socklen_t tolen);`
  - `recvfrom(int s, void *buf, int len, int flags, struct sockaddr *from, socklen_t *fromlen);`
  - Must send/get peer address with each packet
- Example: `udpecho.c`
- Can use UDP in connected mode (Why?)
  - `connect` assigns remote address
  - `send/receive` syscalls, like `sendto/recvfrom` w/o last two arguments
Uses of UDP Connected Sockets

- Kernel demultiplexes packets based on port
  - Can have different processes getting UDP packets from different peers

- Feedback based on ICMP messages (future lecture)
  - Say no process has bound UDP port you sent packet to
  - Server sends port unreachable message, but you will only receive it when using connected socket
Serving Multiple Clients

• A server may block when talking to a client
  – Read or write of a socket connected to a slow client can block
  – Server may be busy with CPU
  – Server might be blocked waiting for disk I/O

• Concurrency through multiple processes
  – Accept, fork, close in parent; child services request

• Advantages of one process per client
  – Don’t block on slow clients
  – May use multiple cores
  – Can keep disk queues full for disk-heavy workloads
Threads

• One process per client has disadvantages:
  – High overhead – fork + exit ~100\(\mu\)sec
  – Hard to share state across clients
  – Maximum number of processes limited

• Can use threads for concurrency
  – Data races and deadlocks make programming tricky
  – Must allocate one stack per request
  – Many thread implementations block on some I/O or have heavy thread-switch overhead

Rough equivalents to `fork()`, `waitpid()`, `exit()`, `kill()`, plus locking primitives.
Non-blocking I/O

• `fcntl` sets `O_NONBLOCK` flag on descriptor
  ```c
  int n;
  if ((n = fcntl(s, F_GETFL)) >= 0)
      fcntl(s, F_SETFL, n|O_NONBLOCK);
  ```

• Non-blocking semantics of system calls:
  – read immediately returns -1 with errno EAGAIN if no data
  – write may not write all data, or may return EAGAIN
  – connect may fail with EINPROGRESS (or may succeed, or may fail with a real error like ECONNREFUSED)
  – accept may fail with EAGAIN or EWOULDBLOCK if no connections present to be accepted
How do you know when to read/write?

Entire program runs in an event loop.

struct timeval {
    long   tv_sec;  /* seconds */
    long   tv_usec; /* and microseconds */
};

int select (int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);

FD_SET(fd, &fdset);
FD_CLR(fd, &fdset);
FD_ISSET(fd, &fdset);
FD_ZERO(&fdset);

• Entire program runs in an event loop.
Event-driven servers

• Quite different from processes/threads
  – Race conditions, deadlocks rare
  – Often more efficient

• But…
  – Unusual programming model
  – Sometimes difficult to avoid blocking
  – Scaling to more CPUs is more complex
Extra content for later
What to do when multiple flows must share a link?
Analogy

• You are running a restaurant. Do you take free reservations? Do you allow walk-ins?
Fixed Allocations

• Some notion of partitioning
  – Frequency, channels, time, …
Fixed Allocation

• **Guaranteed allocation**
  – Great for users, predictable!
• **Low space overhead**
  – Data needs no metadata (e.g., destination, owner)
• **Easy to reason about**

• **Overload: all or nothing**
  – No graceful degradation
• **Waste: allocate for peak, waste for less than peak**
• **Set up time**
  – E.g., set up or change schedule
Statistical Multiplexing

- Break information in finite chunks: packets
- Each packet forwarded independently
  - Must add metadata to each packet
- Properties
  - High utilization (if there is demand)
  - Very flexible
  - Can be unfair
  - Can have unpredictable delays (queues)
Switching

• How to communicate over multiple hops?
• Circuit switching vs Packet switching
  – Circuits reserve capacity along the entire path
  – Packets are switched independently
Circuit Switching
(Fixed Allocation over Multiple Hops)

- Guaranteed allocation
  - Great for users, predictable!
- Low space overhead
  - Data needs no metadata (e.g., destination, owner)
- Easy to reason about

- Overload: all or nothing
  - No graceful degradation
- Waste: allocate for peak, waste for less than peak
- Set up time
  - E.g., set up or change schedule, establish circuit along path
- Failures: must re-establish connection
  - For any failures along path
Each packet forwarded independently
  - Must add metadata to each packet

Properties
  - High utilization (if there is demand)
  - Very flexible
  - Can be unfair
  - Can have unpredictable delays (queues)
  - Different paths for each packet
A Taxonomy of networks

- Communication Network
  - Switched Communication Network
    - Circuit-Switched Communication Network
    - Packet-Switched Communication Network
  - Broadcast Communication Network
  - Point-to-point network
  - Datagram Network
  - Virtual Circuit Network

A hybrid of circuits and packets; headers include a “circuit identifier” established during a setup phase.