CSCI-1680
Layering and Encapsulation

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, David Mazières, Phil Levis, John Jannotti
Administrivia

• HW0: Due TODAY by 11:59pm
• Snowcast out later today
  – Look for an announcement on Ed
• Milestone due by Monday, 9/19 by 11:59pm EDT
  – Small lab, Design questions about your server,
• Waitlist: I will admit another batch after class
Topics for Today

• Layering and Encapsulation
• Intro to IP, TCP, UDP
• Demo on sockets
Map of the Internet, 2021 (via BGP)
OPTE project
OPTE Internet map, 1997-2021:  https://youtu.be/DdaEl6oP6w
How do we make sense of all this?

• Very large number of computers
• Incredible variety of technologies
  – Each with very different constraints
• Lots of *multiplexing*
• 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
- Abstract data from the layer above inside data from the layer below  
  - “Encapsulation”
- Different implementations at one “layer” use same interface
- Allows independent evolution
An analogy

How to deliver a package?
Layers: the classical picture

- **Application** – what users see, e.g., web page via HTTP
- **Presentation** – crypto, conversion between representations
- **Session** – can tie together multiple streams (e.g., audio & video)
- **Transport** – creates a “pipe” to move data between applications
- **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

Service: abstraction provided to layer above
API: concrete way of using the service

Protocol: rules for communication within same layer

Layer N uses the services provided by N-1 to implement its protocol and provide its own services

Layer N+1
A more realistic picture

1. Physical
   Service: move bits to other node across link
   (Electrical engineering problem)

2. Link
   Service: move frames to other node across link
   (eg. Ethernet, Wifi, …)

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

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

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

Where do we handle, eg, security, reliability, fairness?
How/where to handle challenges?

• Can decide on how to distribute certain 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?

Get to decide where (and if) to pay the “cost” of certain features
IP as the “Narrow Waist”

- Applications built using IP, (and TCP, UDP, …)
- IP works over many types of networks
- “Hourglass” architecture of the Internet
  - If every network supports IP, applications run over many different networks (e.g., cellular network, Wifi, Satellite, …)
Network Layer: Internet Protocol (IP)

• Used by most computer networks today
  – Runs over a variety of physical networks: Ethernet, wireless, modem lines, etc.

• Every host has a unique 4-byte IP address (IPv4)
  – E.g., www.cs.brown.edu → 128.148.32.110
  – The network knows how to route a packet to any address
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

- **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)
Anatomy of a packet

- Frame: 452 bytes on wire (3616 bits), 452 bytes captured (3616 bits) on interface en0, id 0
- Transmission Control Protocol, Src Port: 52725, Dst Port: 80, Seq: 1, Ack: 1, Len: 386
- Hypertext Transfer Protocol

```
0000  f8  c2  88  c5  2c  a3  f0  18  98  15  8e  b8  08  00  45  02  
0010  01  b6  00  00  40  00  40  06  bb  92  ac  11  30  fc  80  94  
0020  20  0c  cd  f5  00  50  f1  b0  89  57  ae  46  0c  d9  80  18  
0030  08  02  b2  50  00  00  01  01  08  0a  36  da  1f  03  69  c9  
0040  85  22  47  45  54  20  2f  20  48  54  54  50  2f  31  2e  31  
0050  0d  0a  48  6f  73  74  3a  20  63  73  74  3a  20  77  71  72  6f  
0060  6e  65  74  20  74  68  65  6c  6c  6f  77  6f  72  6b  65  73  73  
0070  3a  20  4d  6f  62  65  20  6f  66  20  73  74  72  69  6e  67  75  73  
```

"GET / HTTP/1.1"
Host: cs.brown
User-Agent: Mozilla/5.0 (M
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”
Problem 1: many existing tools fail because of their specific assumptions and limitations. Operators still find the diagnosis surprisingly hard in practice. To provide a private network with the customer, the traffic is encapsulated with VXLAN, forwarded to the datacenter before reaching the destination server. Table 1 lists the corresponding network except the ingress and egress of ISP-Y-switch. The packet format of each numbered network segment is listed in Table 1.

Figure 1: The example scenario. We collect per-hop traces in our network (Y and ISP-Y-switch) and do not have the traces outside our network except the ingress and egress of ISP-Y-switch. The packet format of each numbered network segment is listed in Table 1.

<table>
<thead>
<tr>
<th>Number</th>
<th>Headers Added after Mirroring</th>
<th>Header Format</th>
<th>Mirrored Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ETHERNET IPv4 ERSVPN</td>
<td>ETHERNET IPv4</td>
<td>TCP</td>
</tr>
<tr>
<td>2</td>
<td>ETHERNET IPv4 ERSVPN</td>
<td>ETHERNET IPv4</td>
<td>TCP</td>
</tr>
<tr>
<td>3</td>
<td>ETHERNET IPv4 ERSVPN</td>
<td>ETHERNET IPv4</td>
<td>TCP</td>
</tr>
<tr>
<td>4</td>
<td>ETHERNET IPv4 GRE</td>
<td>IPv4 UDP</td>
<td>VXLAN Ethernet</td>
</tr>
<tr>
<td>5</td>
<td>ETHERNET IPv4 GRE</td>
<td>IPv4 UDP</td>
<td>VXLAN Ethernet</td>
</tr>
<tr>
<td>6</td>
<td>ETHERNET IPv4 GRE</td>
<td>IPv4 UDP</td>
<td>VXLAN Ethernet</td>
</tr>
<tr>
<td>7</td>
<td>ETHERNET IPv4 ERSVPN</td>
<td>IPv4 GRE</td>
<td>Ethernet IPv4</td>
</tr>
<tr>
<td>8</td>
<td>ETHERNET IPv4 GRE</td>
<td>IPv4 GRE</td>
<td>Ethernet IPv4</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, Windows, …)
  – Important to know and do once
  – Higher-level APIs build on them

• After basic setup, it’s a lot like working with files
Sockets: Communication Between Machines

- Network sockets are file descriptors too
- Datagram sockets (e.g. UDP): unreliable message delivery
  - Send atomic messages, which may be reordered or lost

- Stream sockets (TCP): bi-directional pipes
  - Stream of bytes written on one end, read on another
  - Reads may not return full amount requested, must re-read
System calls for using TCP

**Client**
- `socket` – make socket
- `bind*` – assign address
- `connect` – connect to listening socket

**Server**
- `socket` – make socket
- `bind` – assign address, port
- `listen` – listen for clients
- `accept` – accept connection

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

- 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)
  - 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
Dealing with Data

• Many messages are binary data sent with precise formats

• Data usually sent in Network byte order (Big Endian)
  – Remember to always convert!
  – In C, this is htons(), htonl(), ntohs(), ntohl()