ECE 435 – Network Engineering
Lecture 9

Vince Weaver

https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

14 February 2023

Announcements

- HW#4 was posted (e-mail, DNS)

HW#2 – Programming Notes

- Watch warnings, though I might be running newer version of gcc
- Don't use string operations on binary files
- Writing out to a socket
- If no file specified, index.html If no index.html send a 404 error
- ctime prints own linefeed
- If you report HTTP 1.1, don't close connection after file, there might be more requests and you might get

"connection reset"

- Be sure to check for unexpected errors – what if huge URL is sent?
- Traditionally the biggest problem (if the browser refuses to display) is the wrong Content-length:
  If you send less data than you say you will, it will wait forever for it, or else give a "connection reset" if you close the connection.
- Be sure you read everything the browser is sending (Either big enough buffer, or repeat in loop reading it all). If you send a response before it is done sending it

can confuse things. How can you hold an arbitrary size header? malloc()? Do you want to?

- Be sure to drop the leading / in the file part of a URL
- Many crashed if I requested the `README` file. Have to handle unexpected input from user. (in this case, no file extension)
- If you use firefox you'll see it might also request favico.ico? Why? What should you return (assuming the file doesn't exist?) 404.
- A pain to write in C. But... what language are most webservers written in? Apache=C, nginx =C, lighttpd

$= C$

HW#2 – Questions

- browser
  - Error 404 – not found
  - Error 418 – RFC 2324 coffee protocol (I'm a teapot)
  - Error 451 – Unavailable For Legal Reasons / Ray Bradbury
- http header
  - nginx/1.16.1
  - Isn't actually a website, just redirect to the https site

HW#2 – Something Cool

- I do appreciate the pages you made, even if I didn't comment specifically in the grades.

The Transport Layer

	OSI	TCP/IP
7	Application	Application
6	Presentation	
5	Session	
4	Transport	Transport
3	Network	Internet
2	Data Link	Host-to-network
1	Physical	Host-to-network

The Transport Layer

- Responsible for reliable point-to-point data transport independent of whatever lies beneath.
- Provide process-to-process connectivity, and per-segment error control and per-flow reliability, as well as rate control
- Can be more reliable than underlying network
- Most common interface "socket" API from homeworks.
- Network layer dumps raw bytes onto computer, Transport layer figures out what application gets them

Some Transport Layer Protocols

- TCP (Transmission Control Protocol)
  - connection oriented / stateful / per-flow reliability and rate control
- UDP (User Datagram Protocol)
  - stateless / connectionless
- SCTP (stream control transmission protocol)
  - messages like UDP, reliable like TCP
- QUIC
  - running reliable connection over UDP

The Transport Layer

- Terminology: application = process, data-transfer-unit is a segment, traffic is a flow
- addressing – each process needs a unique ID. For internet, this is the "port" number (16-bit)
- Rate control
  - Flow control – between source and destination
  - Congestion control – between source and network
    None in link layer because only one hop?

Can be done by sender or network

- Real time requirements – things like video and audio need extra info such as timestamp, loss rate, etc. So hard to do with raw TCP/UDP

Unreliable, Connectionless – UDP

- User Datagram Protocol (RFC 768)
- Just an 8-byte header tacked onto the data packet
- No reliability, no rate control, stateless
  If you want these things you have to add at higher layer
- Error control optional
- Why none of those things? All add overhead.
  - Used when want packets to get through quickly.
  - Don't care about re-transmits, better for real-time (VOIP, streaming?)

- ◦ Easy to implement, for low-level stuff like bootp/dhcp
- ◦ Good for broadcasting

- Provides process-to-process communication and per-segment error control
- Can send UDP packets to a destination without having to set up a connection first

UDP Header

2 bytes	2 bytes
Source Port	Destination Port
Packet Length	Checksum

- 16-bits: source port (optional, says where it is coming from in case need to respond, 0 if unused)
- 16-bits: destination port
- 16-bits length (in bytes, includes the header)
  min: 8, max: 65,515 (less than 64k to fit in 64k IP packet)
- 16-bits checksum (optional, 0 if unused, see below)

- data follows

Port Number Review

- 16-bit, so 64k of them
- Can map to any you want, but there are certain well-known ones. Look in `/etc/services`
  For example. WWW is 80/tcp. DNS is 53/udp
- Most OSes, ports <1024 require root (why?)
- 1024 ... 49151 are registered IANA ports
- 49152 ... 65535 are ephemeral ports, dynamic for use by any service

Uniquely identifying flow

- TCP/UDP on IPv4 represented by 104 bit socket pair 5-tuple
  - Source/destination addr
  - Source/destination port
  - protocol ID (TCP or UDP)
- IPv6 has a specific field for this

UDP checksum

- Find info on this in RFC768 and RFC1071
- If set to zero, ignored
- Receiver drops packet if invalid checksum
  Does not request resend, does not notify sender

UDP checksum Algorithm

- 1s complement of sum all 16-bit words in header and payload
  padded with 0s to be multiple of 16-bits
- Also added to the checksum is the pseudo-header (Layering Violation)
  Enables receiver to catch problems (delivered to wrong machine) – why could this be a problem?

  ◦ IPV4: a 96-bit pseudo header: source IP (2*16), dest

IP (2*16), protocol (padded to 16), length

- ◦ IPv6: 128-bit src IP, 128-bit dest IP, 32-bit UDP len, 24-bit 0, 8-bit next/type (17 UDP)

- What happens if checksum is 0? Conflict with disable checksum? Entered as 0xffff, which in ones complement is -0
- Checksum considered mandatory on IPv6 because IPv6 header not checksummed
- Why would you ever leave checksum out? Takes time to compute, might care about latency over errors [video?]

UDP example

```
0x0000:  8875 563d 2a80 0030 18ab 1c39 86dd 6002  .uV=*..0...9..‘.
0x0010:  2618 0031 1140 2610 0048 0100 08da 0230  &..1.@&..H.....0
0x0020:  18ff feab 1c39 2001 4860 4860 0000 0000  .....9..H‘H‘....
0x0030:  0000 0000 8844
UDP starts at 0x36:
                        e239 0035 0031 9c0e 8657  .....D.9.5.1...W
0x0040:  0120 0001 0000 0000 0001 0377 7777 0465  ...........www.e
0x0050:  7370 6e03 636f 6d00 0001 0001 0000 2910  spn.com.......).
0x0060:  0000 0000 0000 00
```

- What is source port? What is destination port? Size?
- How can you tell what high-level protocol it is?
  Can you make an educated guess from the ports?

UDP checksum example (from prev slide)

- 16-bit sum of "virtual header" (two IPv6 addresses, protocol (0x0011) and length of udp packet/header (0x0031)) is 0x29f8c
- 16-bit sum of UDP header leaving off checksum is 0xe29f
- 16-bit sum of UDP data is 0x2e1c0
- Add them get 0x6 63eb
- It's a 16-bit sum, so add 0x6 + 0x63eb = 0x63f1 ones complement is 0x9c0e, which matches the UDP checksum field

UDP and the Operating System

- User binds to UDP socket
- OS sets up queue
- Network stack decodes packet at lower level, notes that it is UDP
- Runs checksum, drops it if invalid
- Finds port, looks to see if any processes listening on that port
- If so, adds to queue
- If not, sends an ICMP "port unreachable" error message

- All UDP messages to that port, no matter who sends them, end up in the same queue.

Writing UDP sockets code

- Use SOCK_DGRAM rather than SOCK_STREAM
- Can skip the listen/accept state, as no connection is there. Just receive the packets as they come in.
- Can't read then write, as no connection. For the server to write back to the client it needs to use recvfrom() which also provides ip/port
- To send a packet use sendto()

UDP Socket – Client code

```
// setup socket
socket_fd = socket(AF_INET, SOCK_DGRAM, 0);

// get server address/port
server=gethostbyname(DEFAULT_HOSTNAME);
memset(&server_addr,0,sizeof(server_addr));
server_addr.sin_family=AF_INET;
memcpy(server->h_addr,&server_addr.sin_addr.s_addr,server->h_length);
server_addr.sin_port=htons(port);

sendto(socket_fd,buffer,strlen(buffer),0,
    (struct sockaddr *)&server_addr, server_len);
```

UDP Socket – Server code

```
// setup socket
socket_fd = socket(AF_INET, SOCK_DGRAM, 0);

// wait for incoming connection
bind(socket_fd, (struct sockaddr *) &server_addr, sizeof(server_addr));

// read data from socket, including client_addr info
recvfrom(socket_fd,buffer,(BUFFER_SIZE-1),0,
    (struct sockaddr *) &client_addr, &client_len);

// send reply
sendto(socket_fd,buffer,strlen(buffer),0);
    (struct sockaddr *)&client_addr, client_len);
```

Common UDP Services

- Obsolete: echo/discard/users/daytime/quote/chargen
- Nameserver
- bootp/tftp
- ntp (network time protocol)
- snmp

UDP real-time

- Real-Time Protocol (RFC1889)
- On top of UDP, multiplexes
- data streams
- timestamps