CS 356: Computer Network Architectures

Lecture 18: End-to-end Protocols
Chapter 5.1, 5.2

Xiaowei Yang
xwy@cs.duke.edu

Transport protocols

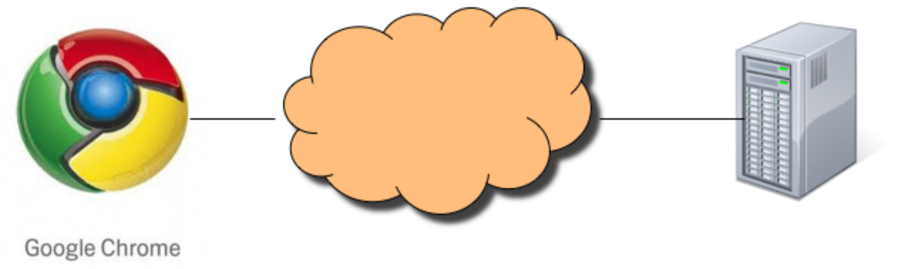
Overview

- UDP and TCP
- Lab 3

Before: How to deliver packet from one host to another

- Direct link
  - Encoding, framing, error detection, reliability
  - Multi-access control
- Multi-link network switching and forwarding
  - Datagrams, virtual circuit
  - Bridges, spanning tree algorithm
- Interconnecting multiple networks
  - IP addressing, forwarding, routing
    - ARP, distance vector, link state, path vector
  - NAT, DHCP, VPN, tunnels etc.

Transport layer design goals



- Goal: a process to process communication channel
  - Upper-layer: application
  - Lower-layer: network

Desirable features

- Reliable delivery
- In-order
- No duplication
- Arbitrarily large messages
- Multiple processes on the same host
- Connection setup
- Not to send faster than a receiver can receive
- Not to send faster than the network allows
- Security
- …

Network service model

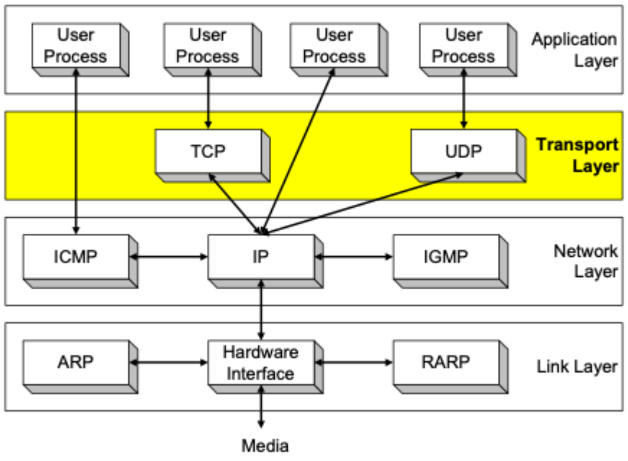
- Best-effort
  - May discard, reorder, duplicate messages
  - Links have MTU limits
  - Arbitrarily long latency

Design choices

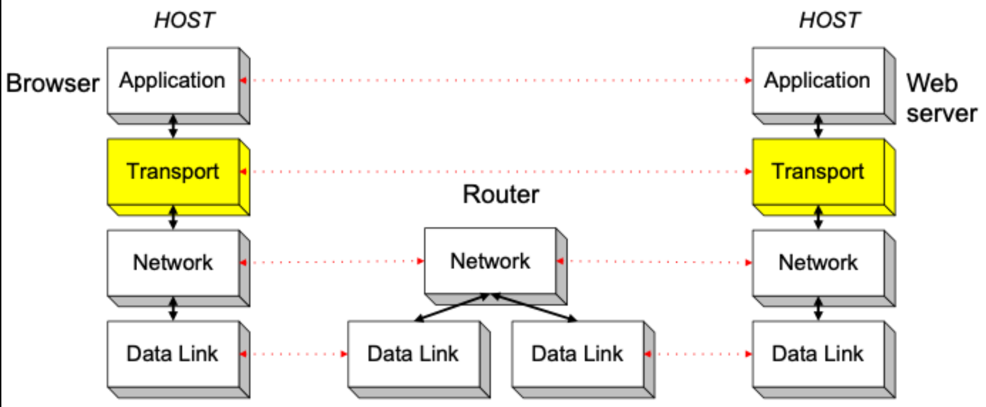
- How to achieve the desired process-to-process service model?
  - Let applications handle it
    - Develop a set of libraries
  - Enhance the network to provide the desirable features
    - Not considered a good idea
  - Place a service layer on top of IP to handle it
    - This is chosen by the Internet design

Big picture

- We move one layer up and look at the transport layer.



Transport layer protocols are end-to-end protocols



Transport Protocols in the Internet

The most commonly used transport protocols are UDP and TCP.

UDP - User Datagram Protocol

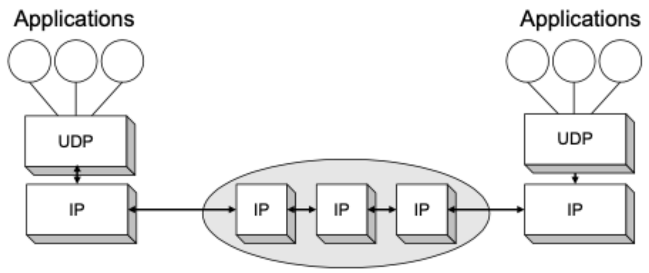
- datagram oriented
- unreliable, connectionless
- simple
- unicast and multicast
- useful only for few applications, e.g., multimedia applications
- used by many services
  - network management (SNMP), routing (RIP), naming (DNS), etc.

TCP - Transmission Control Protocol

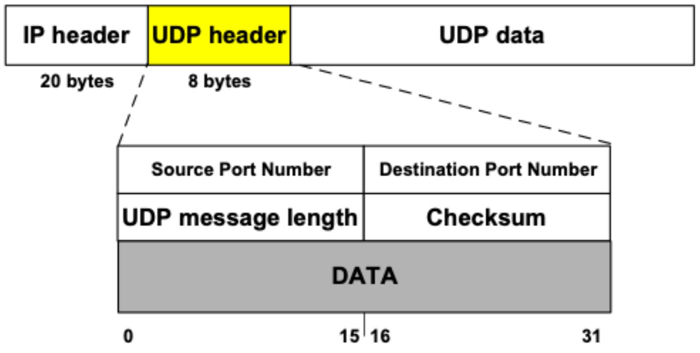
- byte stream oriented
- reliable, connection-oriented
- complex
- only unicast
- used for most Internet applications:
  - web (http), email (smtp), file transfer (ftp), terminal (telnet), etc.

UDP - User Datagram Protocol

- UDP supports unreliable transmissions of datagrams
  - Each output operation by a process produces exactly one UDP datagram
- The only thing that UDP adds is multiplexing and demultiplexing
  - Support multiple processes on the same host
- Protocol number: 17



UDP Format



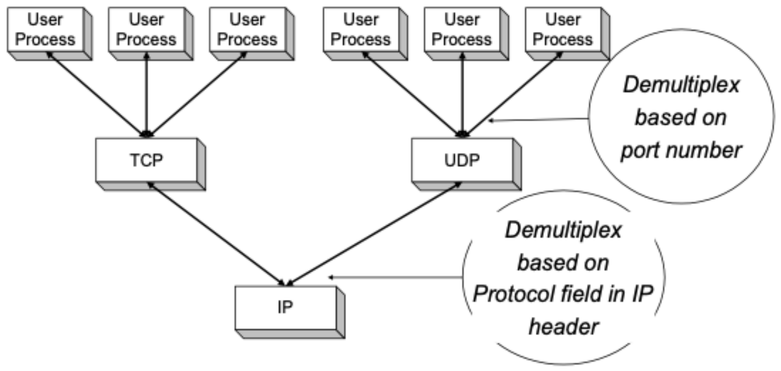
Port numbers (16-bit) identify sending and receiving applications (processes). Maximum port number is 2^{16}-1= 65,535

Message Length (16-bit) is at least 8 bytes (I.e., Data field can be empty) and at most 65,535

Checksum (16-bit) includes UDP header and data, and a pseudo-header (protocol number, IP source/dst) (optional IPv4, mandatory IPv6)

Port Numbers

- UDP (and TCP) use port numbers to identify applications
- A globally unique address at the transport layer (for both UDP and TCP) is a tuple **<IP address, port number>**
- There are 65,535 UDP ports per host.

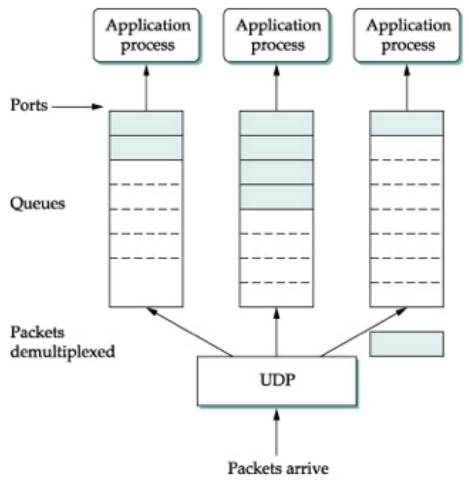

User Process
User Process
User Process
User Process
User Process
User Process
Demultiplex based on port number
TCP
UDP
Demultiplex based on Protocol field in IP header
IP

How to find out application ports

- Servers use well-known ports
  - DNS: 53
  - /etc/services
- A server learns a client’s port from its packets

Implementation

- A “port” is an abstraction
- Implementation may differ from OS to OS
- Ex: port implemented using a message queue
  - Packets discarded when queues are full



Applications

- Domain Name Service
- Streaming applications
  - Real-time Transport protocol (RTP), RTCP
  - Transport on transport
- DHCP
- Traceroute
- Simple Network Management Protocol (SNMP)

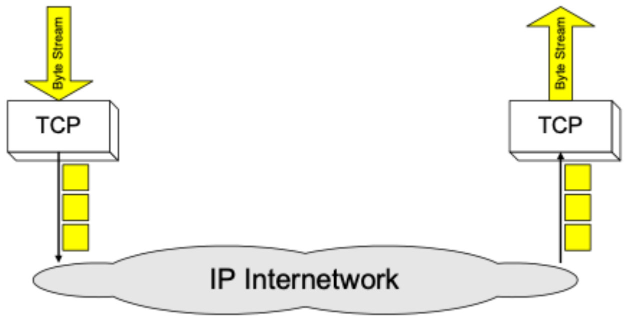
Transport Control Protocol (TCP)

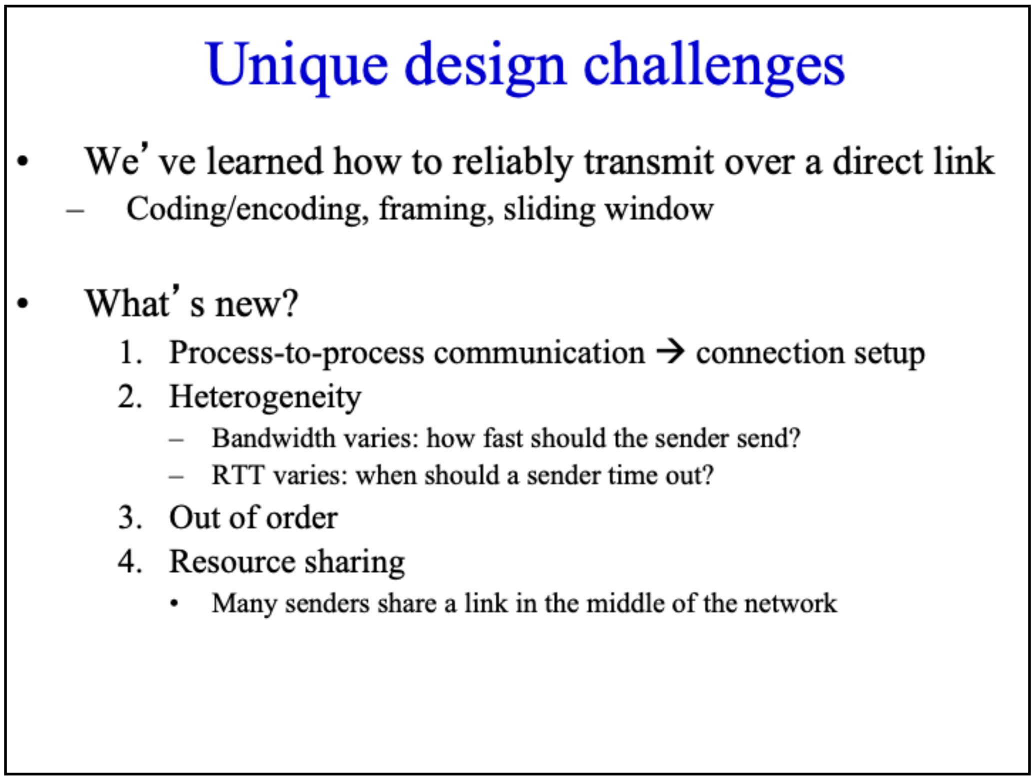
-- perhaps the most widely used protocol

Overview

TCP = Transmission Control Protocol

- Connection-oriented protocol
- Provides a reliable unicast end-to-end byte stream over an unreliable internetwork.





Unique design challenges

- We' ve learned how to reliably transmit over a direct link
  - Coding/encoding, framing, sliding window
- What' s new?
  1. Process-to-process communication → connection setup
  2. Heterogeneity
     - Bandwidth varies: how fast should the sender send?
     - RTT varies: when should a sender time out?
  3. Out of order
  4. Resource sharing
     - Many senders share a link in the middle of the network

A logic connection between two processes

- Need an explicit connection setup/teardown scheme to make sure both ends are ready

Greater heterogeneity

- RTT may range from 1ms to 100ms
  - How to set retransmission timeout values?
- Link bandwidth varies
  - How fast should a sender sends?
  - How to set window size?

Message re-order

- Not possible over a physical link
- A packet may be arbitrarily delayed
  - Use a Maximum Segment Lifetime to discard old messages

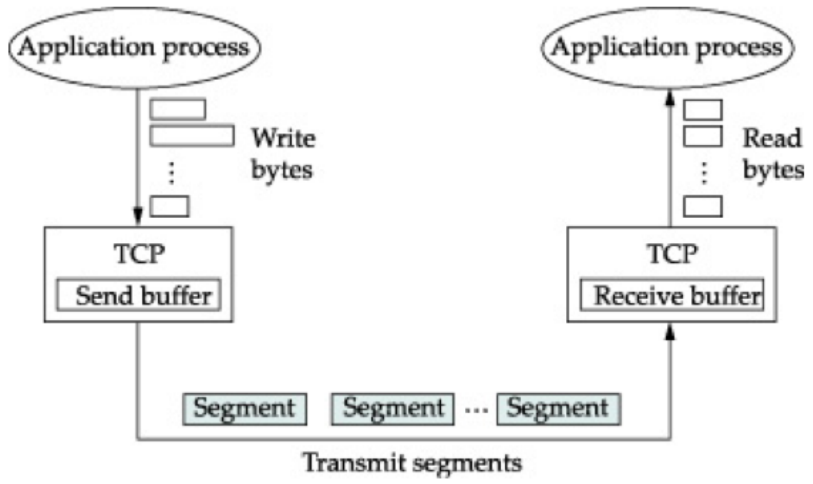
A strawman design

- Hop-by-hop reliable transmission
- A bad idea
  - Can’t ensure end-to-end reliability
  - The end-to-end argument: a function should not be provided at the lower levels of a system unless it can be completely and correctly implemented at that level

TCP features

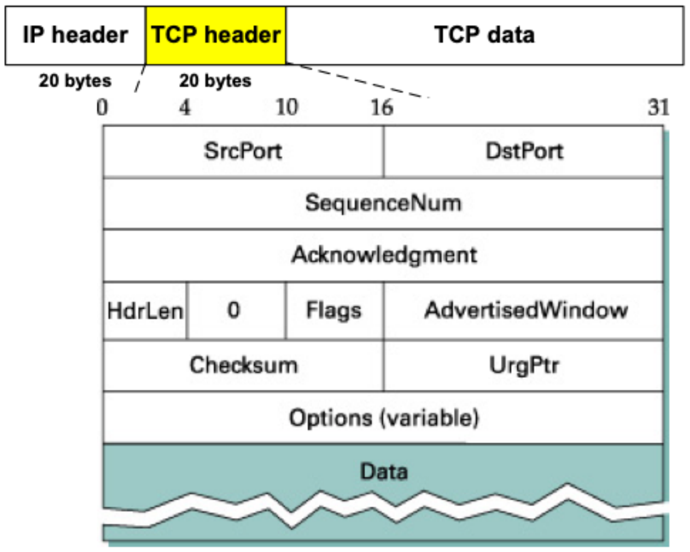
- Connection-oriented
- Reliable, in-order byte stream service
- Fully duplex
- Flow control: not to overrun a receiver
- Congestion control: not to congest the network

TCP manages a byte stream

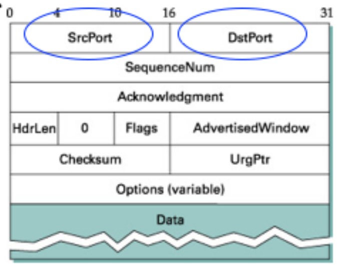


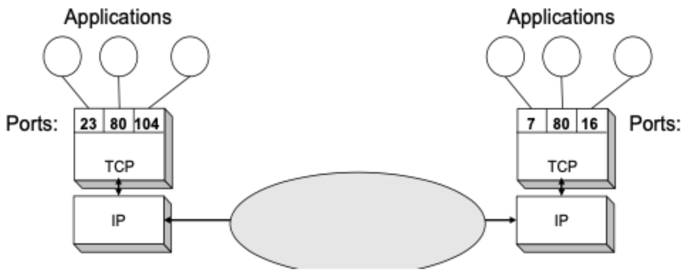
TCP Segment format

TCP segments have a 20 byte header with >= 0 bytes of data.



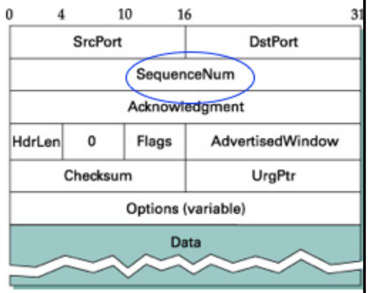
- **Port Number:** multiplexing/demultiplexing
  - A port number identifies the endpoint of a connection.
  - A pair `<IP address, port number>` identifies one endpoint of a connection.
  - Two pairs `<client IP address, client port number>` and `<server IP address, server port number>` identify a TCP connection.



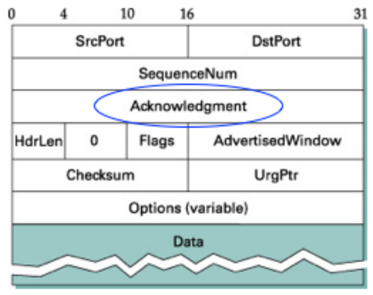


- **Sequence Number (SeqNo):**
  - Sequence number is 32 bits long.
  - So the range of SeqNo is

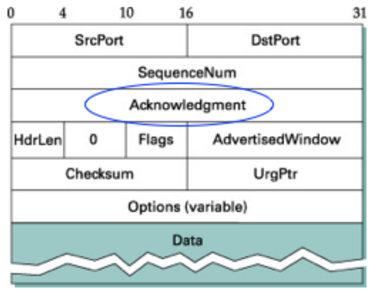
    $0 <= \text{SeqNo} <= 2^{32} - 1 \approx 4.3$ Gbyte
  - The sequence number in a segment identifies the first byte in the segment
  - Initial Sequence Number (ISN) of a connection is set during connection establishment



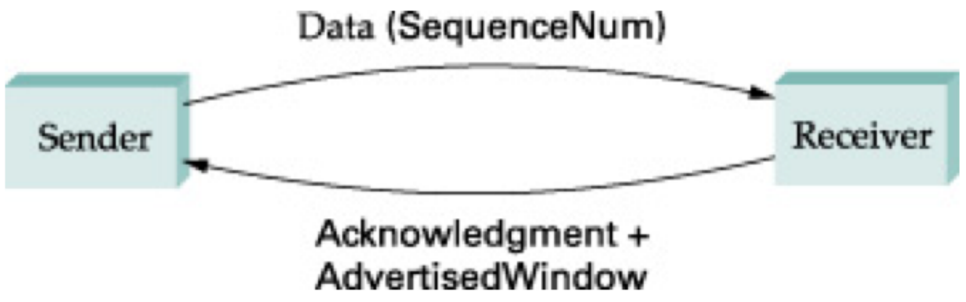
- **Acknowledgement Number (AckNo):**
  - Acknowledgements are piggybacked
  - The AckNo contains the next SeqNo that a host is expecting
  - ACK is cumulative



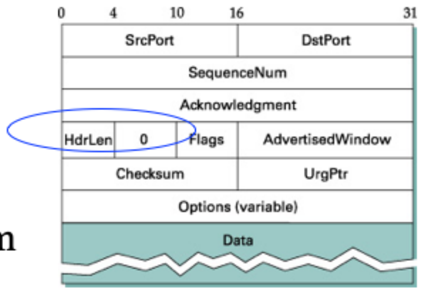
- **AdvertisedWindow:**
  - Used to implement flow control
  - Each side of the connection advertises the window size
  - Window size is the maximum number of bytes that a receiver can accept
  - Maximum window size is 2^{16}-1= 65535 bytes
  - Problematic for high-speed links



A simplified TCP process



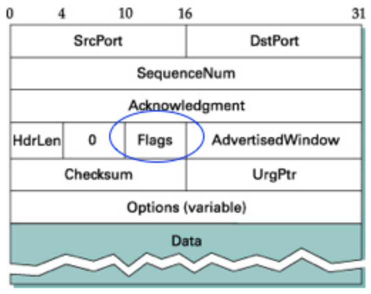
- **Header Length (4bits):**
  - Length of header in 32-bit words
  - Note that TCP header has variable length (with minimum 20 bytes)
  - Question: what's the maximum header length?



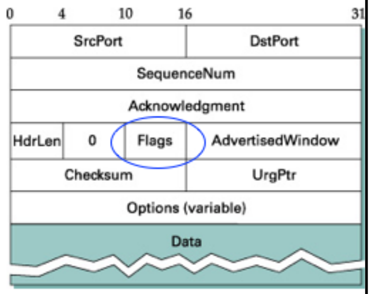
- Reserved: 6 bits
  - Must be zero

- **Flag bits: (from left to right)**
  - **URG: Urgent pointer is valid (not encouraged to use)**
    - If the bit is set, the following bytes contain an urgent message in the range:

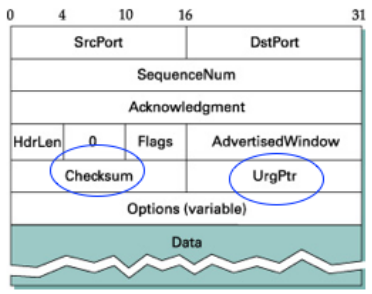
      SeqNo <= urgent message < SeqNo+urgent pointer
  - **ACK: Acknowledgement Number is valid**
    - Segment contains a valid ACK
  - **PSH: PUSH Flag**
    - Notification from sender to the receiver that the receiver should pass all data that it has to the application.
    - Normally set by a sender when the sender's buffer is empty



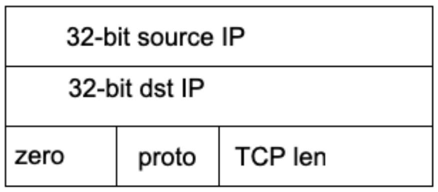
- **Flag bits:**
  - **RST: Reset the connection**
    - The flag causes the receiver to reset the connection
    - Receiver of a RST terminates the connection and indicates higher layer application about the reset
    - (Real life usage: ISP uses RST to block P2P traffic)
  - **SYN: Synchronize sequence numbers**
    - Sent in the first packet when initiating a connection
  - **FIN: Sender is finished with sending**
    - Used for closing a connection
    - Both sides of a connection must send a **FIN**



- **TCP Checksum:**
  - TCP checksum covers over both TCP header **and** TCP data, and a pseudo-header (see next slide)
- **Urgent Pointer:**
  - Only valid if **URG** flag is set



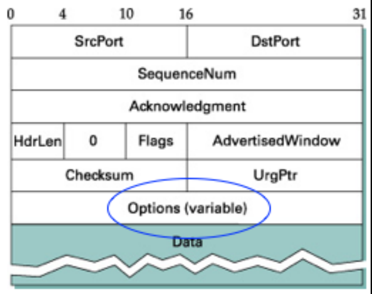
Pseudo-header



- Make sure IP does not make a mistake and delivers a wrong packet to the TCP module
- TCP length
  - The length of the TCP segment, including both header and data. Note that this is not a specific field in the TCP header; it is computed.
- If TCP length is odd, one pad byte of zero will be added to the end for a 16-bit checksum computation

TCP header fields

- **Options**: (Type, length, value)



TCP header fields

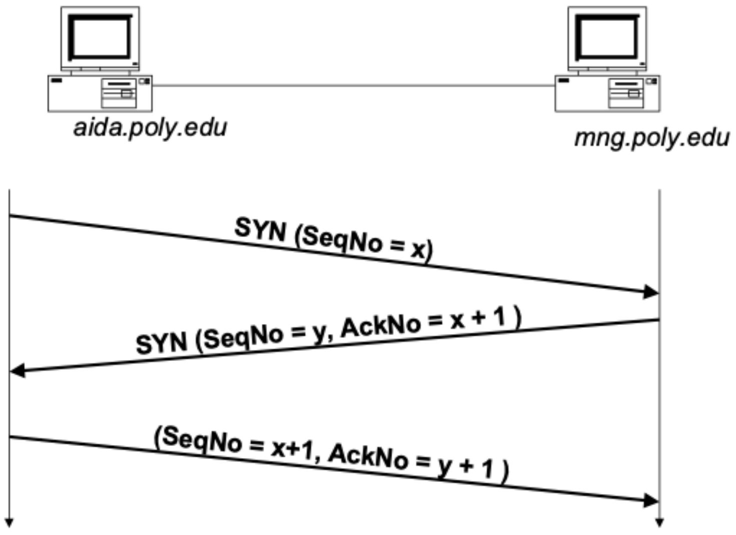
- **Options**:
  - **NOP** is used to pad TCP header to multiples of 4 bytes
  - **Maximum Segment Size**
  - **Window Scale Options**
    - » Increases the TCP window from 16 to 32 bits, i.e., the window size is interpreted differently
    - » This option can only be used in the SYN segment (first segment) during connection establishment time
  - **Timestamp Option**
    - » Can be used for roundtrip measurements

Connection Management in TCP

- **Opening a TCP Connection**
- **Closing a TCP Connection**
- **State Diagram**

TCP Connection Establishment

- TCP uses a **three-way handshake** to open a connection:


aida.poly.edu
mng.poly.edu
SYN (SeqNo = x)
SYN (SeqNo = y, AckNo = x + 1)
(SeqNo = x+1, AckNo = y + 1)

(1) ACTIVE OPEN: Client sends a segment with

SYN bit set *

port number of client

initial sequence number (ISN) of client

(2) PASSIVE OPEN: Server responds with a segment with

SYN bit set *

initial sequence number of server

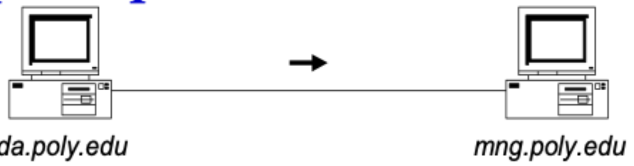
ACK for ISN of client

(3) Client acknowledges by sending a segment with:

ACK ISN of server (* counts as one byte)

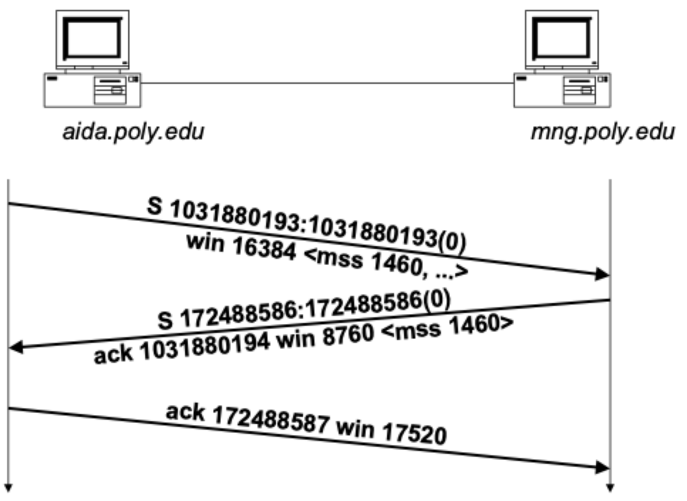
A Closer Look with tcpdump/wireshark

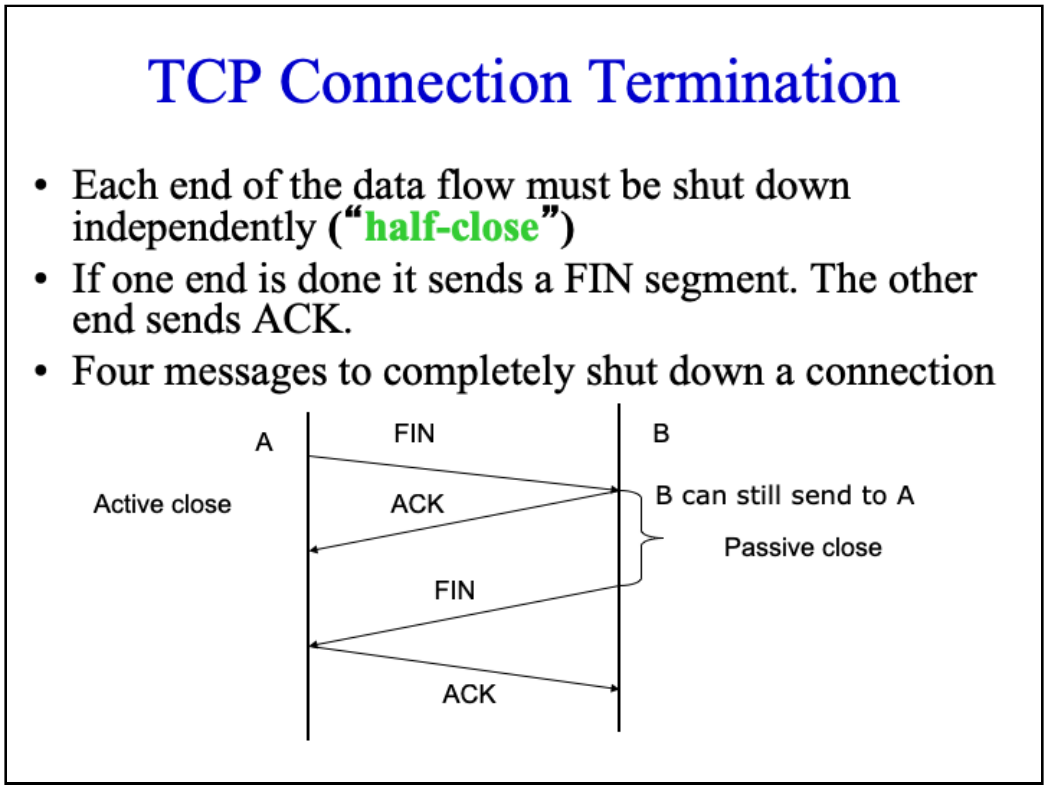
aida issues
an "telnet mng"



```
1   aida.poly.edu.1121 > mng.poly.edu.telnet: S 1031880193:1031880193(0)
                                win 16384 <mss 1460,nop,wscale 0,nop,nop,timestamp>
2   mng.poly.edu.telnet > aida.poly.edu.1121: S 172488586:172488586(0)
                                ack 1031880194 win 8760 <mss 1460>
3   aida.poly.edu.1121 > mng.poly.edu.telnet: . ack 172488587 win 17520
4   aida.poly.edu.1121 > mng.poly.edu.telnet: P 1031880194:1031880218(24)
                                ack 172488587 win 17520
5   mng.poly.edu.telnet > aida.poly.edu.1121: P 172488587:172488590(3)
                                ack 1031880218 win 8736
6   aida.poly.edu.1121 > mng.poly.edu.telnet: P 1031880218:1031880221(3)
                                ack 172488590 win 17520
```

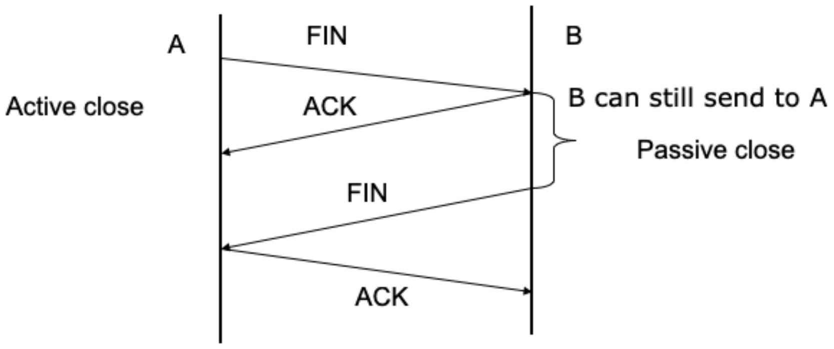
Three-Way Handshake


aida.poly.edu
mng.poly.edu
S 1031880193:1031880193(0)
win 16384 <mss 1460, ...>
S 172488586:172488586(0)
ack 1031880194 win 8760 <mss 1460>
ack 172488587 win 17520



TCP Connection Termination

- Each end of the data flow must be shut down independently ("**half-close**")
- If one end is done it sends a FIN segment. The other end sends ACK.
- Four messages to completely shut down a connection



Four steps involved:

(1) X sends a FIN to Y **(active close)**

(2) Y ACKs the FIN,

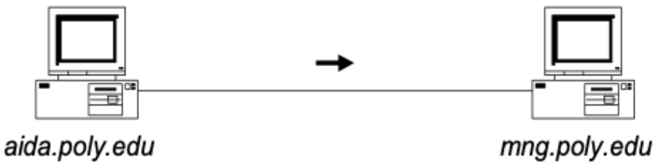
(at this time: Y can still send data to X)

(3) and Y sends a FIN to X **(passive close)**

(4) X ACKs the FIN.

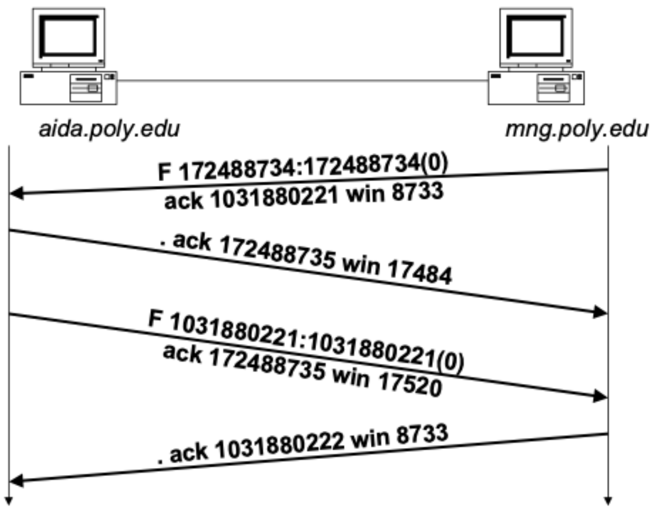
Connection termination with tcpdump/wireshark

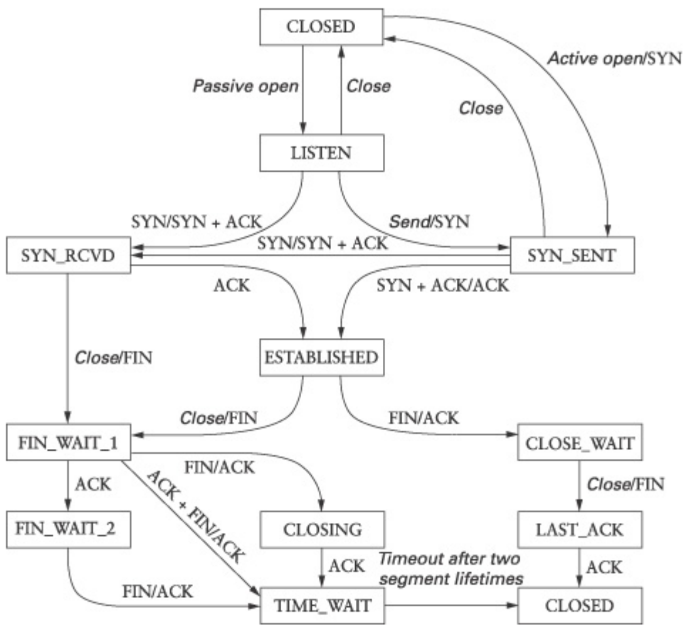
aida issues
an "telnet mng"



```
1  mng.poly.edu.telnet > aida.poly.edu.1121: F 172488734:172488734(0)
                                             ack 1031880221 win 8733
2  aida.poly.edu.1121 > mng.poly.edu.telnet: . ack 172488735 win 17484
3  aida.poly.edu.1121 > mng.poly.edu.telnet: F 1031880221:1031880221(0)
                                             ack 172488735 win 17520
4  mng.poly.edu.telnet > aida.poly.edu.1121: . ack 1031880222 win 8733
```

TCP Connection Termination





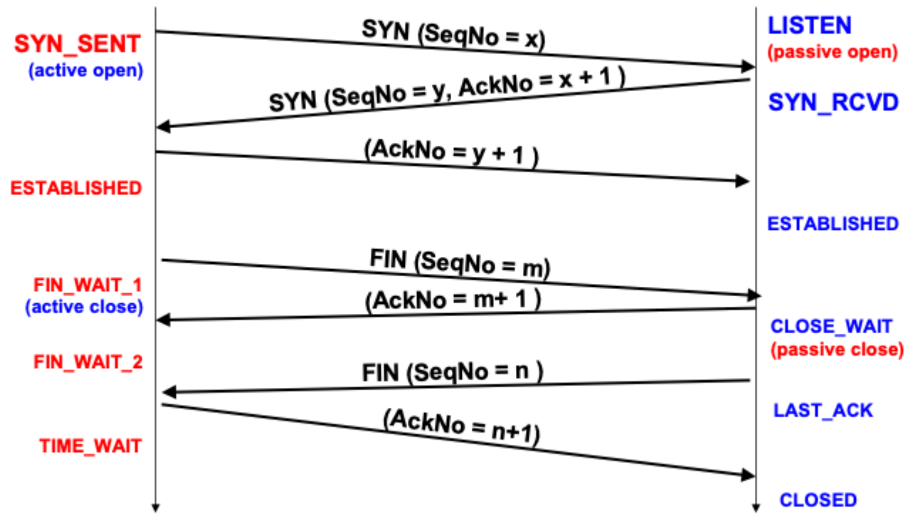
- Two events trigger transitions:
  - Packet arrival
  - Application operations
- Half-close end may still receive data
- Time_Wait
  - Must wait because ACK may be lost
  - The other end may retransmit FIN

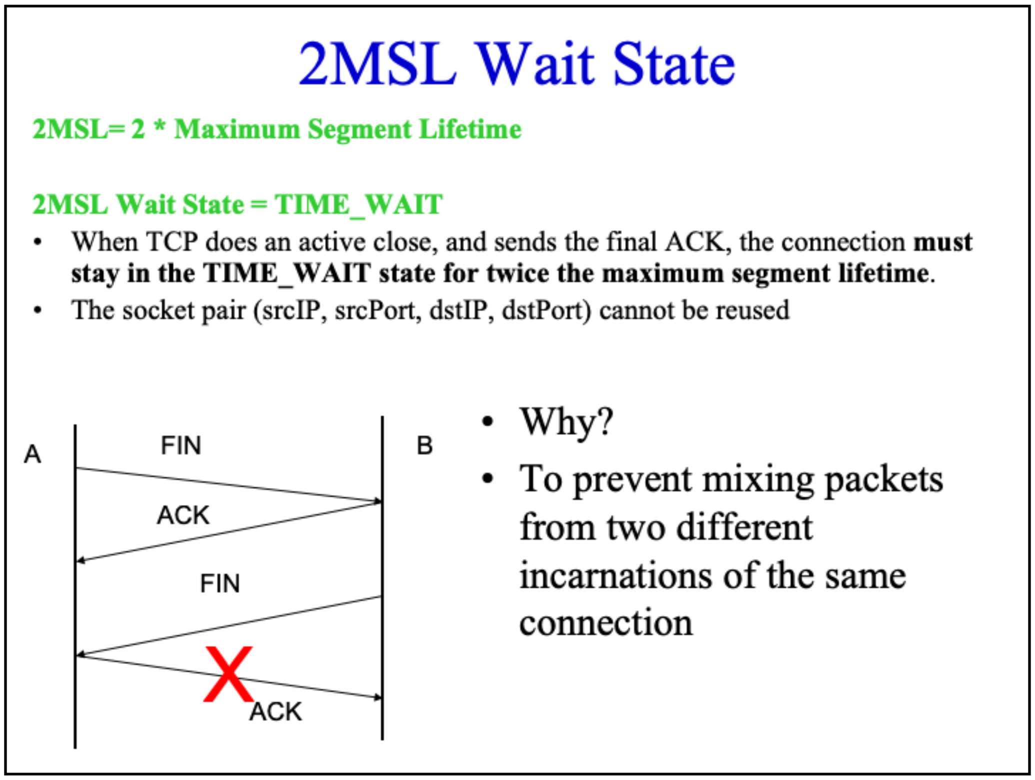
1. Each square denotes a state. Each arc is a transition labeled with the event that triggers the transition.
2. Half close: one end may close and the other end can still send

Connection establishment/tear down

- Active/passive open
- Active/passive close, simultaneous close

TCP States in "Normal" Connection Lifetime



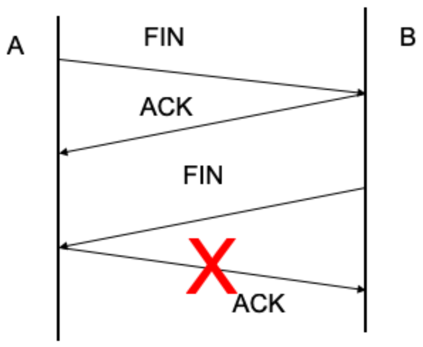


2MSL Wait State

2MSL= 2 * Maximum Segment Lifetime

2MSL Wait State = TIME_WAIT

- When TCP does an active close, and sends the final ACK, the connection **must stay in the TIME_WAIT state for twice the maximum segment lifetime**.
- The socket pair (srcIP, srcPort, dstIP, dstPort) cannot be reused



- Why?
- To prevent mixing packets from two different incarnations of the same connection

TCP is given a chance to resent the final ACK. (Server will timeout after sending the FIN segment and resend the FIN)

Without waiting, FIN may close a wrong connection

The MSL is set to 2 minutes or 1 minute or 30 seconds.

Resetting Connections

- Resetting connections is done by setting the RST flag
- **When is the RST flag set?**
  - Connection request arrives and no server process is waiting on the destination port
  - Abort a connection causes the receiver to throw away buffered data
  - Receiver does not acknowledge the RST segment
  - Abused in practice to block applications

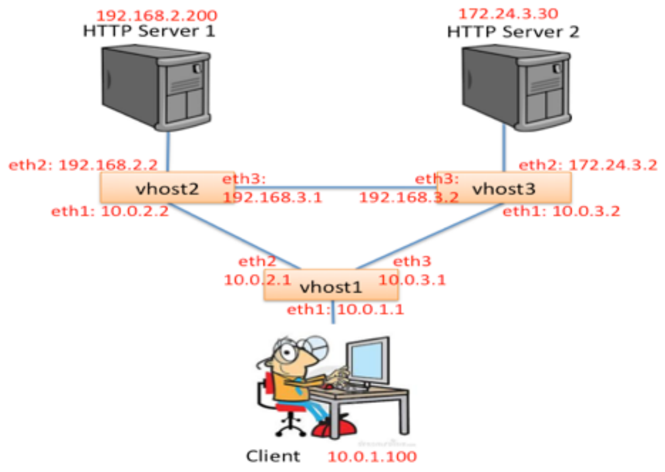
Summary

- UDP
  - Datagram oriented service
- TCP
  - Segment format
  - Connection establish and termination
- Next: continue on TCP

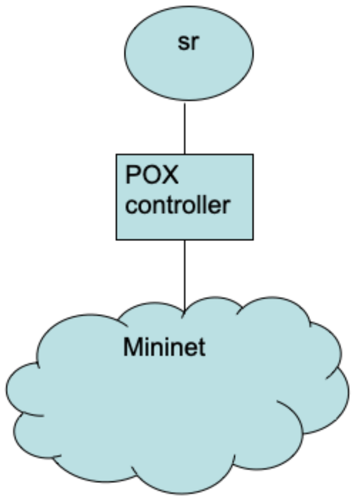
Lab3 - Routing Information Protocol

COMPSCI 356

Topology


192.168.2.200
HTTP Server 1
172.24.3.30
HTTP Server 2
eth2: 192.168.2.2
vhost2
eth3:
192.168.3.1
eth3:
192.168.3.2
vhost3
eth2: 172.24.3.2
eth1: 10.0.2.2
eth1: 10.0.3.2
eth2
10.0.2.1
vhost1
eth3
10.0.3.1
eth1: 10.0.1.1
Client
10.0.1.100

Architecture



Overview

Your task is to implement RIP within your router so that your router will be able to do the following:

1. Build the correct forwarding tables on the assignment topology

2. Detect when routers join/or leave the topology and correct the forwarding tables correctly

Actions required

- Download
  - The latest code skeleton
  - Test cases
- Start working on it now

What’s new

```
struct sr_if
{
    char name[sr_IFACE_NAMELEN];
    unsigned char addr[ETHER_ADDR_LEN];
    uint32_t ip;
    uint32_t speed;
    uint32_t mask;
    uint32_t status; /* 0 - interface down; 1 - interface up*/
    struct sr_if* next;
};
```

Call function “uint32_t sr_obtain_interface_status(struct sr_instance*, const char*)” to obtain the status of an interface

What’s new

How to obtain the interfaces’ status?

- Call the function uint32_t sr_obtain_interface_status(struct sr_instance* sr, const char* name)
- Example: uint32_t status = sr_obtain_interface_status(sr, ‘eth1’)

    status == 0 means the eth1 is down

    status == 1 means the eth1 is up.

- After obtaining the status, you should also change the interface’s status by yourself:

```
sr_if* interface = sr->if_list;
if (strcmp(interface->name, ‘eth1’) == 0)
        interface->status = status.
```

What’s new

```
struct sr_rt
{
    struct in_addr dest;
    struct in_addr gw;
    struct in_addr mask;
    char interface[sr_IFACE_NAMELEN];
    uint32_t metric;
    time_t updated_time;
    struct sr_rt* next;
};
```

What’s new

```
struct sr_rip_pkt {
    uint8_t command;
    uint8_t version; /* version = 2, RIPv2 */
    uint16_t unused;
    struct entry{
        uint16_t afi; /* Address Family Identifier */
        uint16_t tag; /*Route Tag */
        uint32_t address; /* IP Address */
        uint32_t mask; /* Subnet Mask */
        uint32_t next_hop; /* Next Hop */
        uint32_t metric; /* Metric */
        } entries[MAX_NUM_ENTRIES]; # MAX_NUM_ENTRIES = 25
    } __attribute__ ((packed)) ;
typedef struct sr_rip_pkt sr_rip_pkt_t;
```

What's new

```
struct sr_udp_hdr {
    uint16_t port_src, port_dst; /* source and dest port_number */
    uint16_t udp_len;     /* total length */
    uint16_t udp_sum;     /* checksum */
} __attribute__ ((packed)) ;
typedef struct sr_udp_hdr sr_udp_hdr_t;
```

Functions you need to implement

1. void *sr_rip_timeout(void *sr_ptr)
2. void send_rip_request(struct sr_instance *sr);
3. void send_rip_update(struct sr_instance *sr);
4. void update_route_table(struct sr_instance *sr, sr_ip_hdr_t* ip_packet, sr_rip_pkt_t* rip_packet, char* iface);

All of these functions need to be implemented in sr_rt.c

Implement details

1. This assignment uses RIP version 2. All the RIP request and RIP response packets are sent using broadcast.
2. RIP uses UDP as its transport protocol, and is assigned the reserved port number 520.
3. When your code starts, it will automatically call the function send_rip_request.
4. When your router receives a RIP request packet, you should reply a RIP response packet.
5. The function send_rip_update sends RIP response packets. You should enable the split horizon here to alleviate the count-to-infinity problem.

Implement details

6. The function sr_rip_timeout is called every 5 seconds, to send the RIP response packets periodically. It should also check the routing table and remove expired route entry. If a route entry is not updated in 20 seconds, we will think it is expired.

7. The function update_route_table will be called after receiving a RIP response packet. You should enable triggered updates here. When the routing table changes, the router will send a RIP response immediately.

Suggested implementation plan

1. Get familiar with UDP header and RIPv2 Packets
2. Modify your sr_handlepacket function to add a mutex lock before you update your routing table
   - pthread_mutex_lock(&(sr->rt_lock)
   - pthread_mutex_unlock(&(sr->rt_lock)
3. Implement the send_rip_request function
4. Implement the send_rip_response function
5. Test these two functions using Wireshark
6. Implement the update_route_table function
7. Implement the sr_rip_timeout function
8. Test, Test and Test.