Outline

- Transport Layer Functions
  - De-multiplexing
  - Reliability
  - Flow & Congestion Control

- UDP
  - UDP Checksum
  - IP Fragmentation

- TCP
  - Connection Establishment and Termination
Motivation

- IP provides a weak, but efficient service model (*best-effort*)
  - Packets can be delayed, dropped, reordered, duplicated
  - Packets have limited size (why?)

- IP packets are addressed to a host
  - How to decide which application gets which packets?

- How should hosts send into the network?
  - Too fast is bad; too slow is not efficient
Review of the transport layer
Transport Layer Functions

- De-multiplexing
  - Deliver packets to/from different applications on the same host
- Reliability
- Flow Control
- Congestion Control
Multiplexing/demultiplexing

**Multiplexing**
data segments from multiple app processes is sent to lower layer for transmission

**Demultiplexing**
delivering received data segments to corresponding upper layer protocols/apps

Diagram:
- Sender
- Receiver
- Application data
- Transport header
- Segments
- Some other host
Ports

• Need to decide which application gets which packets
• Solution: map each socket to a port
• Client must know server’s port
• Separate 16-bit port address space for UDP and TCP
  – (src_IP, src_port, dst_IP, dst_port) uniquely identifies TCP connection
• Well known ports (0-1023): everyone agrees which services run on these ports
  – e.g., ssh:22, http:80
  – on UNIX, must be root to gain access to these ports (why?)
• Ephemeral ports (most 1024-65535): given to clients
Multiplexing/demultiplexing: examples

host A

host C

Server B

Web client

Web clients

Source IP: C
Dest IP: B
sour port: 1180
dest. port: 80

Source IP: C
Dest IP: B
sour port: 2211
dest. port: 80

Source IP: A
Dest IP: B
sour port: 1180
dest. port: 80

Web server B

port use: simple telnet app

port use: Web server

Source IP: A
Dest IP: B
sour port: 1180
dest. port: 80

Source IP: C
Dest IP: B
sour port: 1180
dest. port: 80

CS 349/Fall05
UDP Characteristics

• UDP is a connectionless datagram service.
  – There is no connection establishment: packets may show up at any time.

• UDP packets are self-contained.

• UDP is unreliable:
  – No acknowledgements to indicate delivery of data.
  – Contains no mechanism to detect missing or mis-sequenced packets.
  – No mechanism for automatic retransmission.
  – No mechanism for flow control, and so can over-run the receiver.
UDP Header

UDP format

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>checksum</td>
</tr>
</tbody>
</table>

Length of UDP segment (in bytes), including header
UDP checksum

**Goal:** detect bit errors (e.g., flipped bits) in transmitted segment

- **Sender:**
  - treat data in the segment as sequence of 16-bit integers
  - checksum: addition (1’s complement sum) of segment contents
  - puts checksum value into UDP checksum field

- **Receiver:**
  - compute checksum of received segment
  - check if computed checksum equals checksum field value:
    - NO - error detected
    - YES - no error detected
UDP Checksum Calculation

- **UDP header**
  - Length: # of bytes (including both header & data)
  - checksum: computed over
    - the *pseudo header*, and
    - UDP header and data.
    - if the field is 0, no checksum

- **pseudo header**: UDP's self-protection against mis-delivered IP packets

### UDP header format

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>checksum</td>
</tr>
</tbody>
</table>

Application data (message)

<table>
<thead>
<tr>
<th>source IP address</th>
<th>destination IP address</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>protocol</td>
</tr>
<tr>
<td></td>
<td>UDP length</td>
</tr>
</tbody>
</table>
Implementation (sender side)

- App calls `socket()`
- App writes data in `buffer`
- App calls `sendto(&buffer, buflen)`
- Data is copied to `kernel`

![Diagram showing the process of data transmission from application to Ethernet]
Implementation (recv. side)

- App calls socket()
- App calls bind()
- Data is received from net and stored in kernel buffer
  - What happens if buffer gets full?
- App calls recvfrom(&buffer,buf
ferlen)
- Data is copied to application buffer
Fragmentation

- UDP application can send buffer up to 8000 bytes
- Most links have MTU size ~ 1500 bytes
  - UDP segment is fragmented to multiple IP datagram fragments
  - Fragments reassembled at receiver
  - Expensive process
Path MTU Discovery

• Alternative:
  - Discover max path MTU on the sender side
  - How:
    • Send IP datagrams with DF flag set
    • Routers reply with ICMP Unreachable Error (Frag Required)
      - Contains the MTU of next hop
    • Continue process with the smaller MTU
TCP

• Transmission Control Protocol
• Reliable, in-order, and at most once delivery
• Messages can be of arbitrary length
• Provides multiplexing/demultiplexing to IP
• Provides congestion control and avoidance
• Application examples: file transfer, chat, web
TCP Service

1) Open connection

2) Reliable byte stream transfer from (IPa, TCP Port1) to (IPb, TCP Port2)
   • Indication if connection fails: Reset

3) Close connection
TCP segment structure

- **URG**: urgent data (generally not used)
- **ACK**: ACK # (field valid)
- **PSH**: push data now (generally not used)
- **RST, SYN, FIN**: connection estab (setup, teardown commands)
- **Checksum**: (as in UDP)
- **Checksum count**: counting by bytes of data

### TCP Segment Fields

- **Source Port #**
- **Destination Port #**
- **Sequence Number**
- **Acknowledgement Number**
- **Receiver Window Size**
- **Checksum**
- **Pointer Urgent Data**
- **Options (variable length)**

### Options Flags

- **URG**: urgent data
- **ACK**: ACK # (field valid)
- **PSH**: push data now
- **RST, SYN, FIN**: connection estab (setup, teardown commands)

### Application Data

- **Variable length**
TCP’s seq. #s and ACK #s

**Seq. #:**
- The number of first byte in segment’s data

**ACK #:**
- seq # of next byte expected from other side
- cumulative ACK

A simple telnet example

User types ‘C’

Host A

Seq=42, ACK=79, data = ‘C’

host B ACKs receipt of ‘C’, echoes back ‘C’

Seq=79, ACK=43, data = ‘C’

host ACKs receipt of echoed ‘C’

Seq=43, ACK=80

host B ACKs receipt of ‘C’, echoes back ‘C’

CS 349/Fall05
Timing Diagram

3-way handshake

SYN k

SYN n; ACK k+1

DATA k+1; ACK n+1

ACK k+n+1

data exchange

FIN

FIN ACK

FIN

FIN ACK

½ close

½ close

Open connect.

Transfer

Close connect.
Open Connection: 3-Way Handshaking

- Goal: agree on a set of parameters: the start sequence number for each side
  - Starting sequence numbers are random.

Client (initiator) <-> Server

Active Open
- connect()

SYN, SeqNum = x

SYN and ACK, SeqNum = y and Ack = x + 1

ACK, Ack = y + 1

Passive Open
- listen()
- accept()
- allocate buffer space
3-Way Handshaking (cont’d)

• Three-way handshake adds 1 RTT delay
• Why?
  – Congestion control: SYN (40 byte) acts as cheap probe
  – Protects against delayed packets from other connection (would confuse receiver)
Close Connection (Two-Army Problem)

- Goal: both sides agree to close the connection
- Two-army problem:
  - “Two blue armies need to simultaneously attack the white army to win; otherwise they will be defeated. The blue army can communicate only across the area controlled by the white army which can intercept the messengers.”
- What is the solution?
Close Connection

- 4-ways tear down connection

- Avoid reincarnation
- Can retransmit FIN ACK if it is lost