Internet Protocols
Fall 2004

Lecture 11
Transport Layer
Andreas Terzis

Midterm Results

• CS349
  - Average: 54.3
  - Median: 47
  - Std-dev: 18.9

• CS449
  - Average: 44.3
  - Median: 44
  - Std-dev: 15.6
Outline

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

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

<table>
<thead>
<tr>
<th>Host A</th>
<th>Source port: x</th>
<th>Dest port: 23</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server B</td>
<td>Source port: 23</td>
<td>Dest port: x</td>
</tr>
</tbody>
</table>

Port use: simple telnet app

<table>
<thead>
<tr>
<th>Host C</th>
<th>Source IP: C</th>
<th>Dest IP: B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Server B</td>
<td>Source port: 2211</td>
<td>Dest port: 80</td>
</tr>
<tr>
<td>Source IP: C</td>
<td>Dest IP: B</td>
<td>Source port: 1180</td>
</tr>
</tbody>
</table>

Port use: Web server

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**
- Length of UDP segment (in bytes), including header

<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)

**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

<table>
<thead>
<tr>
<th>Source port #</th>
<th>Dest port #</th>
<th>Length</th>
<th>Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Data</td>
<td>message</td>
<td></td>
</tr>
</tbody>
</table>

Implementation (sender side)

- App calls `socket()` 
- App writes data in buffer
- App calls `Sendto(&buffer, bufferlen)`
- Data is copied to kernel
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,buferrlen)`
- Data is copied to application buffer

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

Data Link vs. Transport Reliable Connections

• Explicit Connection Establishment phase required
• Variable RTTs
• Delayed Packets
• Flow Control
• Congestion Control
TCP segment structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>sequence #</td>
<td>Sequence number</td>
</tr>
<tr>
<td>acknowledgement #</td>
<td>Acknowledgment number</td>
</tr>
<tr>
<td>rcvr window size</td>
<td>Receiver window size</td>
</tr>
<tr>
<td>ptr urgent data</td>
<td>Pointer to urgent data</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>Options</td>
<td>Variable-length options</td>
</tr>
</tbody>
</table>

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

User types 'C'

Seq=42, ACK=79, data = 'C'

host ACKs receipt of echoed 'C'

Seq=79, ACK=43, data = 'C'
host B ACKs receipt of 'C', echoes back 'C'

Seq=43, ACK=80

Host A

Host B

A simple telnet example
Timing Diagram

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)

Active Open
connect()

 SYN, SeqNum = x

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

 ACK, Ack = y + 1

Server

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

Host 1
- close
- FIN
- FIN ACK

Host 2
- close
- FIN
- FIN ACK

Timeout