Outline

- Transport Layer Functions
  - De-multiplexing
  - Reliability
  - Flow & Congestion Control
- UDP
  - UDP Checksum
  - IP Fragmentation
- TCP
  - Connection Establishment and Termination
  - Sliding Window
  - ACK Strategy
  - Nagle’s algorithm
  - Timeout estimation
  - Flow Control
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

peregrine.cs.jhu.edu

Andreas

O.S.

Data

Header

magneto.isi.jhu.edu

Sam

O.S.

Data

Header

Network Layer

Link Layer

Transport Layer

Application 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
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

![UDP header format diagram](image)
Implementation (sender side)

- App calls `socket()`
- App writes data in `buffer`
- App calls `sendto(&buffer, buflen)`
- 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, buflen)
- 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
# TCP segment structure

<table>
<thead>
<tr>
<th>32 bits</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>sequence number</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>acknowledgement number</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>head len</th>
<th>not used</th>
<th>UAP</th>
<th>R</th>
<th>S</th>
<th>F</th>
<th>rcvr window size</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>checksum</th>
<th>ptr urgent data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Options (variable length)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>application data</th>
</tr>
</thead>
</table>

(content continues)
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’

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

A simple telnet example
Timing Diagram

3-way handshake

1. SYN k
2. SYN n; ACK k+1
3. DATA k+1; ACK n+1
4. ACK k+n+1

Data exchange

1. ACK k+n+1
2. FIN
3. FIN ACK

Half close

1. FIN
2. FIN ACK
3. \(\frac{1}{2}\) close
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

connect()  listen()
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
TCP Connection Management

TCP client lifecycle

TCP server lifecycle

wait 2 min

CLOSED

SYN_SENT

LAST_ACK

ESTABLISHED

CLOSED

LISTEN

FIN_WAIT_2

FIN_WAIT_1

receive ACK send nothing

send FIN

client application initiates close conn

send SYN

receive SYN & ACK send ACK

clients application initiates a TCP connec

send nothing

receive ACK

receive FIN send ACK

receive SYN send SYN & ACK

receive ACK send nothing

server application creates a listen socket
TCP state-transition diagram

A

- I-finished(M) → FIN(M)
- timer on FIN → ACK (M+1)
- Wait for B to finish
- ack(N+1) → FIN (N)
- ACK(N+1) → wait for 2MSL before deleting the conn state
- I-finished

B

- CLOSED
- LISTEN
- SYN_SENT
- SYN_RCVD
- ESTABLISHED
- FIN_WAIT_1
- FIN_WAIT_2
- CLOSING
- TIME_WAIT
- CLOSE_WAIT
- LAST_ACK
- CLOSED

- Active open/SYN
- Passive open
- Close

- FIN/ACK
- SYN/SYN + ACK
- SYN + ACK/ACK
- FIN/ACK
- SYN/SYN + ACK
- ACK
- ACK
- ACK
- FIN/ACK
- ACK
- FIN/ACK
- ACK
- FIN/ACK
- ACK
- FIN/ACK
- ACK
- FIN/ACK
- ACK

CS 344/Spring 2007
Connection Reset

• RST Segments
  - When connection request arrives to non-existing server, OS sends back RST signal
  - Can be used to do **abortive release** of established connection
    • Receiver throws away queued data
• **Sending side**
  - \( \text{LastByteAacked} \leq \text{LastByteSent} \)
  - \( \text{LastByteSent} \leq \text{LastByteWritten} \)
  - buffer bytes between \( \text{LastByteAacked} \) and \( \text{LastByteWritten} \)

• **Receiving side**
  - \( \text{LastByteRead} < \text{NextByteExpected} \)
  - \( \text{NextByteExpected} \leq \text{LastByteRcvd} + 1 \)
  - buffer bytes between \( \text{NextByteRead} \) and \( \text{LastByteRcvd} \)
Protection Against Wrap Around

• 32-bit SequenceNum

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Time Until Wrap Around</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (1.5 Mbps)</td>
<td>6.4 hours</td>
</tr>
<tr>
<td>Ethernet (10 Mbps)</td>
<td>57 minutes</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>13 minutes</td>
</tr>
<tr>
<td>FDDI (100 Mbps)</td>
<td>6 minutes</td>
</tr>
<tr>
<td>STS-3 (155 Mbps)</td>
<td>4 minutes</td>
</tr>
<tr>
<td>STS-12 (622 Mbps)</td>
<td>55 seconds</td>
</tr>
<tr>
<td>STS-24 (1.2 Gbps)</td>
<td>28 seconds</td>
</tr>
</tbody>
</table>
Keeping the Pipe Full

- **16-bit AdvertisedWindow**

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Delay x Bandwidth Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (1.5 Mbps)</td>
<td>18KB</td>
</tr>
<tr>
<td>Ethernet (10 Mbps)</td>
<td>122KB</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>549KB</td>
</tr>
<tr>
<td>FDDI (100 Mbps)</td>
<td>1.2MB</td>
</tr>
<tr>
<td>STS-3 (155 Mbps)</td>
<td>1.8MB</td>
</tr>
<tr>
<td>STS-12 (622 Mbps)</td>
<td>7.4MB</td>
</tr>
<tr>
<td>STS-24 (1.2 Gbps)</td>
<td>14.8MB</td>
</tr>
</tbody>
</table>

assuming 100ms RTT
Silly Window Syndrome

- How aggressively does sender exploit open window?

- Receiver-side solutions
  - after advertising zero window, wait for space equal to a maximum segment size (MSS)
  - delayed acknowledgements
Nagle’s Algorithm

• How long does sender delay sending data?
  – too long: hurts interactive applications
  – too short: poor network utilization
  – strategies: timer-based vs self-clocking

• When application generates additional data
  – if fills a max segment (and window open): send it
  – else
    – if there is unack’ed data in transit: buffer it until ACK arrives
    – else: send it
# TCP Recvr: when to send ACK?

<table>
<thead>
<tr>
<th>Event</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-order segment arrival, no gaps, everything earlier already ACKed</td>
<td>delayed ACK: wait up to 200ms, If nothing arrived, send ACK</td>
</tr>
<tr>
<td>in-order segment arrival, no gaps, one delayed ACK pending</td>
<td>immediately send one cumulative ACK</td>
</tr>
<tr>
<td>out-of-order arrival: higher-than-expect seq. #, gap detected</td>
<td>send duplicate ACK, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>arrival of segment that partially or completely fills a gap</td>
<td>immediate ACK if segment starts at the lower end of gap</td>
</tr>
</tbody>
</table>
TCP Flow Control

- receive side of TCP connection has a receive buffer:

  - flow control: sender won’t overflow receiver’s buffer by transmitting too much, too fast

- app process may be slow at reading from buffer

  - speed-matching service: matching the send rate to the receiving app’s drain rate
TCP Flow control: how it works

(Suppose TCP receiver discards out-of-order segments)

• spare room in buffer
  = RcvWindow
  = RcvBuffer - [LastByteRcvd - LastByteRead]
Flow Control (cont.)

• If the receiver indicated size of zero in last ACK how can the server send more data?
  - Sender periodically sends probes
  - Receiver sends *window update* ACK
Setting Timers

• The sender needs to set retransmission timers in order to know when to retransmit a packet the may have been lost

• How long to set the timer for?
  - **Too short**: may retransmit before data or ACK has arrived, creating duplicates
  - **Too long**: if a packet is lost, will take a long time to recover (inefficient)
Timing Illustration

Timeout too long $\Rightarrow$ inefficiency

Timeout too short $\Rightarrow$ duplicate packets
Adaptive Timers

- The amount of time the sender should wait is about the round-trip time (RTT) between the sender and receiver
  - For link-layer networks (LANs), this value is essentially known
  - For multi-hop WANS, rarely known
- Must work in both environments, so protocol should adapt to the path behavior
- Measure successive ack delays $T(n)$
  Set timeout = average + 4 deviations
RTT measurement and RTO

\[
\text{SRTT} = (1-\alpha) \times \text{SRTT} + \alpha \times \text{SampleRTT}
\]
\[
\text{rttvar} = \text{rttvar} + \beta \times (|\text{diff}| - \text{rttvar})
\]
Where \( \text{diff} = \text{SampleRTT} - \text{SRTT} \)

Assume \( \text{SRTT} = 500\text{msec} \), \( \text{rttvar} = 120 \), \( \alpha = 1/8 \), \( \beta = 1/4 \):
\[
\text{diff} = \text{SampleRTT} - \text{SRTT} = 80\text{ms}
\]
\[
\text{SRTT} = \text{SRTT} + \alpha \times \text{diff} = 510\text{ms}
\]
\[
\text{rttvar} = \text{rttvar} + \beta \times (|\text{diff}| - \text{rttvar}) = 110
\]
\[
\text{RTO} = \text{SRTT} + 4 \times \text{rttvar} = 510 + 440 = 950
\]
How to measure RTT in case of retransmission?

- Karn’s algorithm
  - On retx, don’t update estimated RTT (and double RTO)