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

Andreas

O.S.

Data
Header

Sam

O.S.

Data
Header

Application Layer

Transport Layer

Network Layer

Link 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**
  - Application data
  - Transport header
  - Segment

- **Receiver**
  - Some other host
  - Application, transport, network

---

CS 344/Spring 2008
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
  - \((\text{src\_IP}, \text{src\_port}, \text{dst\_IP}, \text{dst\_port})\) 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 (User Datagram Protocol) is a transport-layer protocol that offers unreliable, connectionless service. Its header contains the following fields:

- **source port #** (16 bits): The port number of the source application.
- **dest port #** (16 bits): The port number of the destination application.
- **length** (16 bits): The length of the UDP segment (in bytes), including the header.
- **checksum** (16 bits): A 16-bit checksum to verify the integrity of the data.
- **Application data** (variable length): The actual data (message) sent by the application.

The total length of the UDP segment is 32 bits, as shown in the figure.
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
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,buf,ferlen)`
- 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>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 number</td>
<td>Sequence number of bytes sent so far</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement of bytes received</td>
</tr>
<tr>
<td>rcvr window size</td>
<td>Receiver's window size</td>
</tr>
<tr>
<td>ptr urgent data</td>
<td>Pointer to urgent data</td>
</tr>
<tr>
<td>Options (variable length)</td>
<td>Options (variable length)</td>
</tr>
<tr>
<td>URG</td>
<td>Urgent data (generally not used)</td>
</tr>
<tr>
<td>ACK</td>
<td>ACK # field valid</td>
</tr>
<tr>
<td>PSH</td>
<td>Push data now (generally not used)</td>
</tr>
<tr>
<td>RST, SYN, FIN</td>
<td>Connection establishment/teardown commands</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>application data</td>
<td>Application data (variable length)</td>
</tr>
</tbody>
</table>

TCP segment structure diagram:

- **URG**: urgent data (generally not used)
- **ACK**: ACK # field valid
- **PSH**: push data now (generally not used)
- **RST, SYN, FIN**: connection establishment/teardown commands
- **checksum**: Checksum (as in UDP)
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**

Host A

User types ‘C’

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

CS 344/Spring 2008
3-way handshake:

- SYN k
- SYN n; ACK k+1
- DATA k+1; ACK n+1

Data exchange:

- ACK k+n+1

½ close:

- FIN
- FIN ACK
- FIN
- FIN ACK
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

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
TCP state-transition diagram

A

I-finished(M)  FIN(M)

FIN (N)  ACK (M+1)

ack(N+1)  wait for 2MSL before deleting the conn state

Wait for B to finish

I-finished

B

CLOSED

SYN_RCVD

SYN_SENT

LISTEN

ESTABLISHED

FIN_WAIT_1

FIN_WAIT_2

CLOSING

TIME_WAIT

CLOSE_WAIT

LAST_ACK

CLOSED

Active open γ SYN

Passive open

Close

Close

Send/SYN

SYN/SYN + ACK

SYN + ACK

FIN/ACK

Timeout after two segment lifetimes

CS 344/Spring 2008
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
Sliding Window Revisited

- **Sending side**
  - $\text{LastByteAced} \leq \text{LastByteSent}$
  - $\text{LastByteSent} \leq \text{LastByteWritten}$
  - buffer bytes between $\text{LastByteAced}$ and $\text{LastByteWritten}$

- **Receiving side**
  - $\text{LastByteRead} < \text{NextByteExpected}$
  - $\text{NextByteExpected} \leq \text{LastByteRcvd} + 1$
  - buffer bytes between $\text{LastByteRead}$ 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 **Advertised Window**

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

- receive side of TCP connection has a receive buffer:
  - RcvWindow
  - RcvBuffer
  - spare room
  - TCP data in buffer
  - data from IP
  - application process

- app process may be slow at reading from buffer

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

- 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]
• 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 → inefficiency

Timeout too short → 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

CS 344/Spring 2008
RTT measurement and RTO

Assume SRTT = 500msec, rttvar = 120, \( \alpha = \frac{1}{8} \), \( \beta = \frac{1}{4} \):

- diff = SampleRTT - SRTT = 80ms
- SRTT = SRTT + \( \alpha \times \text{diff} \) = 510ms
- rttvar = rttvar + \( \beta \times (|\text{diff}| - \text{rttvar}) \) = 110
- RTO = SRTT + 4 \times rttvar = 510 + 440 = 950

SRTT = (1-\( \alpha \)) \times \text{SRTT} + \( \alpha \) \times \text{SampleRTT}

rttvar = rttvar + \( \beta \) \times (|\text{diff}| - \text{rttvar})

Where \( \text{diff} = \text{SampleRTT} - \text{SRTT} \)
How to measure RTT in case of retransmission?

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