Transport Layer

- Transport Layer Services
  - connection-oriented vs. connectionless
  - multiplexing and demultiplexing
- UDP: Connectionless Unreliable Service
- TCP: Connection-Oriented Reliable Service
  - connection management: set-up and tear down
  - reliable data transfer protocols
  - flow and congestion control

Readings: Chapter 5 (5.1, 5.2)

Transport Protocols

- Lowest level end-to-end protocol,
  - Header generated by sender is interpreted only by the destination
  - Routers view transport header as part of the payload

Transport Services and Protocols

- provide logical communication between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - recv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP

Transport Layer Services

- Underlying best-effort network
  - drops messages
  - re-orders messages
  - delivers duplicate copies of a given message
  - delivers messages after an arbitrarily long delay
- Common end-to-end services
  - guarantee message delivery
  - deliver messages in the same order they are sent
  - deliver at most one copy of each message
  - allow the receiver to flow control the sender
  - support multiple application processes on each host

Transport vs. Application and Network Layer

- application layer: application processes and message exchange
- network layer: logical communication between hosts
- transport layer: logical communication support for app processes
  - relies on, enhances, network layer services

Household analogy: 12 kids sending letters to 12 kids
- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service

End to End Issues

- Transport services built on top of (potentially) unreliable network service
  - packets can be corrupted or lost
  - Packets can be delayed or arrive "out of order"
- Do we detect and/or recover errors for apps?
  - Error Control & Reliable Data Transfer
- Do we provide "in-order" delivery of packets?
  - Connection Management & Reliable Data Transfer
- Potentially different capacity at destination, and potentially different network capacity
  - Flow and Congestion Control
Internet Transport Protocols

TCP service:
- connection-oriented: setup required between client, server
- reliable transport between sender and receiver
- flow control: sender won’t overwhelm receiver
- congestion control: throttle sender when network overloaded

Both provide logical communication between app processes running on different hosts!

UDP service:
- unreliable data transfer between sender and receiver
- does not provide: connection setup, reliability, flow control, congestion control

UDP: User Datagram Protocol [RFC 768]
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

UDP Checksum

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

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

Receiver:
- computes checksum of received segment
- check if computed checksum equals checksum field value:
  - NO – error detected
  - YES – no error detected. But maybe error nonetheless? More later …
Checksum: Example

arrange data segment in sequences of 16-bit words

\[
\begin{align*}
010011001100110 & \quad 110101010101010 \\
000111000011111 & \quad 010101001010100
\end{align*}
\]

sum: 0100110011001101

checksum(1’s complement): 1011010100110100

verify by adding:

\[
\begin{align*}
1111111111111111 & \quad 0110011001100110 \\
1101010101010101 & \quad 0000111100001111
\end{align*}
\]

TCP Overview

- Connection-oriented
- Full duplex
- Byte-stream: app writes bytes, TCP sends segments, app reads bytes
- Flow control: keep sender from overrunning receiver
- Congestion control: keep sender from overrunning network

Functionality Split

- Network provides best-effort delivery
- End-systems implement many functions
  - Reliability
  - In-order delivery
  - Demultiplexing
  - Message boundaries
  - Connection abstraction
  - Flow Control
  - Congestion control
  - ...

High-Level TCP Characteristics

- Protocol implemented entirely at the ends
  - Fate sharing
- Protocol has evolved over time and will continue to do so
  - Nearly impossible to change the header
  - Use options to add information to the header
  - Change processing at endpoints
  - Backward compatibility is what makes it TCP

Evolution of TCP

1975: Three-way handshake (Raymond Tomlinson, SIGCOMM 75)
1976: Nagle’s algorithm to reduce overhead and predict congestion collapse
1983: BSD 4.2 supports TCP/IP
1985: Congestion collapse observed
1986: Van Jacobson’s algorithms for TCP congestion control (implemented in 4.3BSD Tahoe)
1989: 4.3BSD Reno supports TCP/ip
1990: 4.3BSD Reno has improved TCP/ip
1991: Nagle’s algorithm to reduce overhead and predict congestion collapse
1993: Fast retransmit
1994: ECN
1995: SACK TCP
1996: TCP Vegas
1997: TCP Tahoe
1998: TCP Vegas
1999: SACK TCP
2000: TCP Vegas

TCP Through the 1990s

1993: Fast retransmit
1994: ECN
1996: SACK TCP
1998: TCP Vegas
TCP Segment Header Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>16-bit source port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>16-bit destination port number</td>
</tr>
<tr>
<td>sequence number</td>
<td>32-bit sequence number</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>32-bit acknowledgement number</td>
</tr>
<tr>
<td>rcvr window size</td>
<td>16-bit receiver window size</td>
</tr>
<tr>
<td>ptr urgent data</td>
<td>16-bit pointer to urgent data</td>
</tr>
<tr>
<td>options (variable length)</td>
<td>Options for additional TCP fields (variable length)</td>
</tr>
<tr>
<td>application data</td>
<td>Variable-length application data</td>
</tr>
</tbody>
</table>

TCP Segment Format (cont)

- Each connection identified with 4-tuple:
  - (SrcPort, SrcIPAddr, DstPort, DstIPAddr)
- Sliding window + flow control
  - acknowledgment, SequenceNum, AdvertisedWindow
- Flags
  - SYN, FIN, ACK, RESET, PUSH, URG
- Checksum
  - pseudo header (src & dst IP addresses) + TCP header + data

TCP Connection Set Up

TCP sender, receiver establish “connection” before exchanging data segments
- initialize TCP variables:
  - seq #
  - buffers, flow control info
- client: end host that initiates connection
- server: end host contacted by client

TCP 3-Way Hand-Shake

**Step 1:** client sends TCP SYN control segment to server
- specifies initial seq #

**Step 2:** server receives SYN, replies with SYN+ACK control segment
- ACKs received SYN
- specifies server -> receiver initial seq #

**Step 3:** client receives SYN+ACK, replies with ACK segment (which may contain 1st data segment)

TCP Connection Setup Example

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Proto</th>
<th>SrcPort-DstPort [Flags]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13:37:4375</td>
<td>76.13.155.114</td>
<td>128.101.35.150</td>
<td>TCP</td>
<td>1144 &gt; 22 [SYN]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq=78244755 Len=0 MSS=1260</td>
</tr>
<tr>
<td>2</td>
<td>13:38:750</td>
<td>128.101.35.150</td>
<td>76.13.155.114</td>
<td>TCP</td>
<td>22 &gt; 1414 [SYN, ACK]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq=778406756 Ack=78244755 Win=322000 Len=0 MSS=1440</td>
</tr>
<tr>
<td>3</td>
<td>13:38:750</td>
<td>76.13.155.114</td>
<td>128.101.35.150</td>
<td>TCP</td>
<td>1414 &gt; 22 [ACK]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq=37248552784 Ack=778406756 Win=6384 Len=0</td>
</tr>
<tr>
<td>4</td>
<td>13:38:750</td>
<td>76.13.155.114</td>
<td>128.101.35.150</td>
<td>TCP</td>
<td>80 &gt; 1567 [PSH, ACK]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq=48733972 Ack=87248552784 Win=17640 Len=0</td>
</tr>
<tr>
<td>5</td>
<td>13:38:750</td>
<td>76.13.155.114</td>
<td>128.101.35.150</td>
<td>TCP</td>
<td>80 &gt; 1567 [ACK]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq=87248552784 Ack=37248552784 Win=15200 Len=0 MSS=1440</td>
</tr>
</tbody>
</table>

TCP Connection Setup Example

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Proto</th>
<th>SrcPort-DstPort [Flags]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13:66:11233</td>
<td>76.13.155.114</td>
<td>128.101.35.204</td>
<td>TCP</td>
<td>1567 &gt; 80 [SYN]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq=37248552784 Len=0 MSS=1260</td>
</tr>
<tr>
<td>2</td>
<td>13:89:625</td>
<td>128.101.35.204</td>
<td>76.13.155.114</td>
<td>TCP</td>
<td>80 &gt; 1567 [SYN, ACK]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq=48733972 Ack=37248552784 Win=25200 Len=0 MSS=1440</td>
</tr>
<tr>
<td>3</td>
<td>13:89:625</td>
<td>76.13.155.114</td>
<td>128.101.35.204</td>
<td>TCP</td>
<td>1567 &gt; 80 [ACK]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq=37248552784 Ack=48733972 Win=17640 Len=0 MSS=1440</td>
</tr>
<tr>
<td>4</td>
<td>13:89:625</td>
<td>76.13.155.114</td>
<td>128.101.35.204</td>
<td>TCP</td>
<td>80 &gt; 1567 [PSH, ACK]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq=48733972 Ack=37248552784 Win=15200 Len=0 MSS=1440</td>
</tr>
<tr>
<td>5</td>
<td>13:89:625</td>
<td>128.101.35.204</td>
<td>76.13.155.114</td>
<td>TCP</td>
<td>80 &gt; 1567 [ACK]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq=48733972 Ack=37248552784 Win=15200 Len=0 MSS=1440</td>
</tr>
</tbody>
</table>
Connection Setup Error Scenarios

- Lost (control) packets
  - What happen if SYN lost? client vs server actions
  - What happen if SYN+ACK lost? client vs server actions
  - What happen if ACK lost? client vs server actions
- Duplicate (control) packets
  - What does server do if duplicate SYN received?
  - What does client do if duplicate SYN+ACK received?
  - What does server do if duplicate ACK received?

Connection Setup Error Scenarios (cont'd)

- Importance of (unique) initial seq. no.?
  - When receiving SYN, how does server know it's a new connection request?
  - When receiving SYN+ACK, how does client know it's a legitimate, i.e., a response to its SYN request?
- Dealing with old duplicate packets from old connections (or from malicious users)
  - If not careful: "TCP Hijacking"
  - How to choose unique initial seq. no.?
    - randomly choose a number (and add to last syn# used)
- Other security concern:
  - "SYN Flood" -- denial-of-service attack

Detecting Half-Open Connections

<table>
<thead>
<tr>
<th>TCP A</th>
<th>TCP B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. (CRASH)</td>
<td>(send 300, receive 100)</td>
</tr>
<tr>
<td>2. CLOSED</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>3. SYN-SENT (\Rightarrow) (&lt;SEQ=400&gt;&lt;CTL=SYN&gt;)</td>
<td>(??)</td>
</tr>
<tr>
<td>4. (??) (\Leftarrow) (&lt;SEQ=300&gt;&lt;ACK=100&gt;&lt;CTL=ACK&gt;) (\Leftarrow) ESTABLISHED</td>
<td>(Abort!!)</td>
</tr>
<tr>
<td>5. SYN-SENT (\Rightarrow) (&lt;SEQ=100&gt;&lt;CTL=_RST&gt;)</td>
<td>CLOSED</td>
</tr>
<tr>
<td>6. SYN-SENT</td>
<td>SYN-SENT</td>
</tr>
<tr>
<td>7. SYN-SENT (\Rightarrow) (&lt;SEQ=400&gt;&lt;CTL=SYN&gt;)</td>
<td>CLOSED</td>
</tr>
</tbody>
</table>

TCP State Diagram: Connection Setup

TCP: Closing Connection

Remember TCP duplex connection:
Client wants to close connection:

**Step 1:** client end system sends TCP FIN control segment to server

**Step 2:** server receives FIN, replies with ACK, half closed

**Step 3:** client receives ACK, half closed, wait for server to close

**Step 4:** server finishes sending data, also ready to close

**Step 5:** client receives FIN, replies with ACK, connection fully closed

**Step 6:** server receives ACK, connection fully closed

Well Done!

Problem Solved?
**TCP: Closing Connection (revised)**

**Two Army Problem!**

**Step 5:** client receives FIN, replies with ACK.
- Enters "timed wait" - will respond with ACK to received FINs

**Step 6:** server receives ACK, connection fully closed

**Step 7:** client timer expires, connection fully closed

---

**TCP Connection Tear-Down Example**

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Proto</th>
<th>SrcPort&gt;DstPort [Flags]</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>35.156</td>
<td>70.13.155.114</td>
<td>128.101.35.150</td>
<td>TCP</td>
<td>1414 &gt; 22 [PSH,ACK]</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Seq=758246388</td>
<td>Ack=3778411633</td>
<td>Win=15920 Len=32</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>35.156</td>
<td>70.13.155.114</td>
<td>128.101.35.150</td>
<td>TCP</td>
<td>1414 &gt; 22 [FIN,ACK]</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Seq=758246420</td>
<td>Ack=3778411633</td>
<td>Win=15920 Len=0</td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>35.437</td>
<td>128.101.35.150</td>
<td>70.13.155.114</td>
<td>TCP</td>
<td>22 &gt; 1414 [ACK]</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Seq=7778411633</td>
<td>Ack=758246420</td>
<td>Win=25200 Len=0</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>35.453</td>
<td>128.101.35.150</td>
<td>70.13.155.114</td>
<td>TCP</td>
<td>22 &gt; 1414 [FIN,ACK]</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Seq=7778411633</td>
<td>Ack=758246421</td>
<td>Win=25200 Len=0</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>35.453</td>
<td>128.101.35.150</td>
<td>70.13.155.114</td>
<td>TCP</td>
<td>22 &gt; 1414 [FIN,ACK]</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Seq=7778411633</td>
<td>Ack=758246421</td>
<td>Win=25200 Len=0</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>35.453</td>
<td>128.101.35.150</td>
<td>70.13.155.114</td>
<td>TCP</td>
<td>1414 &gt; 22 [ACK]</td>
</tr>
</tbody>
</table>

---

**State Diagram: Connection Tear-down**

**TCP Connection Management FSM**

**Reliability and Error Recovery**

- **ARQ vs. FEC**
  - Automatic retransmission request
  - Forward error correction
- **General ARQ Algorithms**
  - **Stop & Wait**
    - Perform issue: low utilization when delay-bw product large
  - **Sliding Window Protocols**
    - Go-Back-N
    - Selective Repeat
    - Key design issue: window size vs. size of seq. no. space
Error Recovery: Stop and Wait

- **ARQ**
  - Receiver sends acknowledgement (ACK) when it receives packet
  - Sender waits for ACK and timeouts if it does not arrive within some time period
- Simplest ARQ protocol
- Send a packet, stop and wait until ACK arrives

Recovering from Error

- Time
- Packet
- ACK
- Timeout

Sender Receiver

Problems with Stop and Wait

- How to recognize a duplicate
- Performance
  - Can only send one packet per round trip

How to Recognize Resends?

- Use sequence numbers
  - Both packets and acks
- Sequence # in packet is finite → How big should it be?
  - For stop and wait?
  - One bit - won’t send seq #1 until received ACK for seq #0

Problem with Stop & Wait Protocol

- Can’t keep the pipe full
  - Utilization is low when bandwidth-delay product (R x RTT) is large!

Stop & Wait: Performance Analysis

Example:
1 Gbps connection, 15 ms end-end prop. delay, data segment size: 1 KB = 8Kb

\[
T_{\text{server}} = \frac{L}{R} \quad \text{(packet length in bits)}
\]

\[
= 8 \times 10^{-4} \text{ s} = 0.008 \text{ ms}
\]

\[
U_{\text{server}} = \frac{L}{L + L/R} \quad \frac{L}{RTT + L/R} = \frac{L}{RTT \times R + L} = \frac{0.008}{30.008} = 0.00027
\]

- Utilization, fraction of time server busy sending
- 1KB data segment every 30 msec (round trip time)

Moral of story:
- Network protocol limits use of physical resources
How to Keep the Pipe Full?

- Send multiple packets without waiting for first to be acked
  - Number of pkts in flight = window
- Reliable, unordered delivery
  - Several parallel stop & wait
  - Send new packet after each ack
  - Sender keeps list of unack'ed packets; resends after timeout
  - Receiver same as stop & wait
- How large a window is needed?
  - Suppose 10Mbps link, 4ms delay, 500byte pkts
  - Round trip delay * bandwidth = capacity of pipe

Pipelining: Increased Utilization

- First packet bit transmitted, $t = 0$
- Last bit transmitted, $t = L / R$
- ACK arrives, send next packet, $t = RTT + L / R$

\[ U_{sender} = \frac{L}{R} \left( \frac{3}{RTT + L / R} \right) \]

<table>
<thead>
<tr>
<th>$U_{sender}$</th>
<th>0.0008</th>
</tr>
</thead>
</table>

Increase utilization by a factor of 3!

Sliding Window

- Reliable, ordered delivery
- Receiver has to hold onto a packet until all prior packets have arrived
  - Why might this be difficult for just parallel stop & wait?
  - Sender must prevent buffer overflow at receiver
- Circular buffer at sender and receiver
  - Packets in transit = buffer size
  - Advance when sender and receiver agree packets at beginning have been received

Sender/Receiver State

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max ACK received</td>
<td>Next seqnum</td>
</tr>
<tr>
<td>Sent &amp; Acked</td>
<td>Acceptable Packet</td>
</tr>
<tr>
<td>Not Usable</td>
<td>Not Usable</td>
</tr>
<tr>
<td>Not Sent</td>
<td>Not Sent</td>
</tr>
<tr>
<td>Not Acked</td>
<td>Not Acked</td>
</tr>
</tbody>
</table>

Window Sliding - Common Case

- On reception of new ACK (i.e. ACK for something that was not acked earlier)
  - Increase sequence of max ACK received
  - Send next packet
- On reception of new in-order data packet (next expected)
  - Hand packet to application
  - Send cumulative ACK - acknowledges reception of all packets up to sequence number
  - Increase sequence of max acceptable packet
Loss Recovery

- Go-Back-N recovery
  - Set timer upon transmission of each packet
  - Cumulative ACK
  - Retransmit all unacknowledged packets
  - No receiver buffering, out-of-order packets are discarded

- Selective Repeat
  - Sender keeps a timer for each packet
  - Selective ACK
  - Receiver must buffer all out-of-order packets
  - When timeout, retransmit only one packet

- Performance during loss recovery
  - No longer have an entire window in transit
  - Can have much more clever loss recovery

Selective Repeat

- Receiver individually acknowledges all correctly received pkts
  - Buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender only resends packets for which ACK not received
  - Sender timer for each unACKed packet
- Sender window
  - N consecutive seq #s
  - Again limits seq #s of sent, unACKed packets

Sequence Numbers

- How large does size of sequence number space need to be?
  - Must be able to detect wrap-around
  - Depends on sender/receiver window size
- E.g.
  - size of seq. no. space ≥ 8, send winrecv win?
  - If pkts 0..6 are sent successfully and all acks lost
    - Receiver expects 7..12, sender retransmits old 0..6!!
  - size of sequence no. space must be ≥ send window + recv window

Sequence Numbers in TCP

- TCP regards data as a "byte-stream"
  - each byte in byte stream is numbered.
    - 32 bit value, wraps around
    - initial values selected at start-up time
- TCP breaks up byte stream in packets.
  - Packet size is limited to the Maximum Segment Size (MSS)
- Each packet has a sequence number
  - seq. no. of 1st byte indicates where it fits in the byte stream
- TCP connection is duplex
  - data in each direction has its own sequence numbers

<table>
<thead>
<tr>
<th>packet 8</th>
<th>packet 9</th>
<th>packet 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>13450</td>
<td>14950</td>
<td>16050</td>
</tr>
<tr>
<td>17550</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Go-Back-N in Action

Selectively Repeat: Sender, Receiver Windows

Sequence Numbers in TCP

<table>
<thead>
<tr>
<th>packet 8</th>
<th>packet 9</th>
<th>packet 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>13450</td>
<td>14950</td>
<td>16050</td>
</tr>
<tr>
<td>17550</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TCP Seq. #'s and ACKs

**Seq. #'s:**
- byte stream "number" of first byte in segment's data

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

---

TCP Reliable Data Transfer

- TCP creates reliable data transfer service on top of IP's unreliable service
- Pipelined segments
- Cumulative ACKs
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

---

TCP = Go-Back-N Variant

- Sliding window with cumulative acks
  - Receiver can only return a single "ack" sequence number to the sender.
  - Acknowledges all bytes with a lower sequence number
  - Starting point for retransmission
  - Duplicate acks sent when out-of-order packet received
- But: sender only retransmits a single packet.
  - Reason?
    - Only one that it knows is lost
    - Network is congested? shouldn’t overload it
- Error control is based on byte sequences, not packets.
  - Retransmitted packet can be different from the original lost packet
    - Why?

---

TCP ACK generation ([RFC 1122, RFC 2581](#))

**Event at Receiver**

<table>
<thead>
<tr>
<th>Event at Receiver</th>
<th>TCP Receiver Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of in-order segment with expected seq #</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>Arrival of in-order segment with expected seq #. One other segment has ACK pending</td>
<td>Immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>Arrival of out-of-order segment higher-than-expect seq. #</td>
<td>Immediately send duplicate ACK, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>Arrival of segment that partially or completely fills gap</td>
<td>Immediate send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>

---

TCP Flow Control

- receiver side of TCP connection has a receive 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
- **app process may be slow at reading from buffer**
TCP Flow Control: How It Works

- Rcvr advertises spare room by including value of RcvWindow in segments
- Sender limits unACKed data to RcvWindow - guarantees receive buffer doesn't overflow

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

- spare room in buffer
  - RcvWindow (dynamically changes)
  - RcvBuffer - [LastByteRcvd - LastByteRead]

TCP Segment Structure

- source port #
- dest port #
- sequence number
- acknowledgement number
- checksum
- receiver window
- sender window
- options (variable length)
- data (variable length)
- urgent pointer
- data
- padding

TCP Round Trip Time and Timeout

Q: How to set TCP timeout value?
- longer than RTT - but RTT varies
- too short - premature timeout
- unnecessary retransmissions
- too long - slow reaction to segment loss

Q: How to estimate RTT?
- SampleRTT: measured time from segment transmission until ACK receipt
  - ignore retransmissions, why?
- SampleRTT will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current SampleRTT

Triggering Transmission

- How does TCP decide to transmit a segment?
  - MSS (Maximum segment size)
    - set to size of largest segment TCP can send without local IP fragmentation (MTU of directly connected)
    - Sending process explicitly asked to do (Push to flush)
    - Firing timer
  - Silly Window Syndrome
    - Flow control needs to be maintained
    - Sender can transmit full segment (MSS) when Acked by receiver

Triggering Transmission (cont’d)

- Receiver may delay ACKs, but how long?
  - Ultimate solution lies with sender:
    - When does the TCP sender decide to transmit a segment?

  Nagle's Algorithm:
    - Waiting too long hurt interactive applications (Telnet)
      - Without waiting, risk of sending a bunch of tiny packets (silly window syndrome)
    - Wait till timer expires:
      - self clocking: As long as TCP has any data in flight, sender receives an ACK which can be used to trigger transmission
      - If no data in flight, immediately send the segment (setting TCP_NoDelAY option)

Silly Window Syndrome (cont’d)

- Window currently closed from receiver
  - ACK opens MSS/2 bytes
  - Should sender transmit MSS/2?
    - Original TCP implementation silent
    - Early implementation of TCP decided to go ahead
      - Sender can not know when the window will open for full MSS
    - If sender is aggressive, sending available window size
      - Results silly window syndrome
      - Small segment size remains indefinitely
      - Hence a problem when either sender transmits a small segment or receiver opens window a small amount
Round-trip Time Estimation

- Importance of accurate RTT estimators:
  - Low RTT estimate → unneeded retractions
  - High RTT estimate → poor throughput
- RTT estimator must adapt to change in RTT
  - But not too fast, or too slow!
- Spurious timeouts
  - "Conservation of packets" principle – never more than a window worth of packets in flight

Adaptive Retransmission (Original Algorithm)

- Measure `SampleRTT` for each segment/ACK pair
- Compute weighted running average of RTT
  - \( \text{EstRTT} = \alpha \times \text{EstimatedRTT} + (1-\alpha) \times \text{SampleRTT} \)
  - \( \alpha \) between 0.8 and 0.9 (to smooth `EstimatedRTT`)
  - Small \( \alpha \) indicates temp. fluctuation, a large value more stable, may not be quick to adapt to real changes
- Set timeout based on `EstRTT`
  - \( \text{TimeOut} = 2 \times \text{EstRTT} \)

Retransmission Ambiguity

- ACK is for Original transmission but was for retransmission => Sample RTT is too large
- ACK is for retransmission but was for original => Sample RTT too small

Karn/Partridge Algorithm

- Solution:
  - Do not sample RTT when retransmitting
  - Double timeout for each retransmission
  - Next timeout to be twice the last timeout, rather than basing it on the last `EstimatedRTT`
- Karn and Partridge proposal is exponential backoff
  - Congestion is most likely cause of lost segments
  - TCP sources should not react too aggressively to a timeout
  - More timeouts mean more cautious the source should become (congestion problem)

Jacobson/ Karels Algorithm

- Original computation for RTT did not take the variance of sample RTTs into account
  - If variation among samples is small, Estimated RTT can be better used without increasing the estimate twice
  - A large variance in the samples mean Time out values should not be too tightly coupled to the Estimated RTT
- New Calculations for average RTT
  - \( \text{Diff} = \text{SampleRTT} - \text{EstRTT} \)
  - \( \text{DevRTT} = \text{DevRTT} + (1/\beta) \times \text{Dev} \)
  - \( \text{Dev} = \text{Dev} + \phi \times (|\text{Diff}| - \text{Dev}) \)
  - where \( \phi \) is a fraction between 0 and 1
- Consider variance when setting timeout value
  - \( \text{TimeOut} = \mu \times \text{EstRTT} + \phi \times \text{DevRTT} \)
  - where \( \mu = 1 \) and \( \phi = 4 \)

TCP Round Trip Time Estimation

\[ \text{EstimatedRTT} = (1-\alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT} \]

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: \( \alpha = 0.125 \)

Setting the timeout interval

- Estimated RTT plus "safety margin"
  - Large variation in `EstimatedRTT` -> larger safety margin
- "safety margin": accommodate variations in `EstimatedRTT`
  - \( \text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}| \)
  - (typically, \( \beta = 0.25 \))
- \( \text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT} \)
Example RTT Estimation:

![RTT Data](image-url)

**Timestamp Extension**

- Used to improve timeout mechanism by more accurate measurement of RTT
- When sending a packet, insert current time into option
  - 4 bytes for time, 4 bytes for echo a received timestamp
- Receiver echoes timestamp in ACK
  - Actually will echo whatever is in timestamp
- Removes retransmission ambiguity
  - Can get RTT sample on any packet

**Timer Granularity**

- Many TCP implementations set RTO (Retransmission Timeout) in multiples of 200, 500, 1000ms
- Why?
  - Avoid spurious timeouts - RTTs can vary quickly due to cross traffic
  - Make timers interrupts efficient
- What happens for the first couple of packets?
  - Pick a very conservative value (seconds)

**Important Lessons**

- TCP state diagram → setup/teardown
- TCP timeout calculation → how is RTT estimated
- Modern TCP loss recovery
  - Why are timeouts bad?
  - How to avoid them? → e.g. fast retransmit