Recall: TCP sender, receiver establish “connection” before exchanging data segments

• initialize TCP variables:
  – seq. #s
  – buffers, flow control info (e.g. RcvWindow)

• client: connection initiator
  
  Socket clientSocket = new Socket("hostname", "port number");

• server: contacted by client
  
  Socket connectionSocket = 

Three way handshake:

Step 1: client host sends TCP SYN segment to server
  specifies initial seq #
  no data

Step 2: server host receives SYN, replies with SYNACK segment
  server allocates buffers
  specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data 1
Closing a connection:

client closes socket: `clientSocket.close();`

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

**Step 2:** server
receives FIN, replies with ACK. Closes connection, sends FIN.
**Step 3:** client receives FIN, replies with ACK.
- Enters “timed wait” - will respond with ACK to received FINs

**Step 4:** server, receives ACK. Connection closed.

**Note:** with small modification, can handle simultaneous FINs.
TCP Connection Management (cont)

TCP client lifecycle

TCP server lifecycle

- CLOSED
- TIME_WAIT
- SYN_SENT
- ESTABLISHED
- FIN_WAIT_2
- FIN_WAIT_1

TCP client lifecycle:
- CLOSED
- TIME_WAIT
- SYN_SENT
- ESTABLISHED
- FIN_WAIT_2
- FIN_WAIT_1

TCP server lifecycle:
- CLOSED
- LISTEN
- SYN_RCVD
- ESTABLISHED
- CLOSE_WAIT
- LAST_ACK

- Server application creates a listen socket
- Receive SYN, send SYN & ACK
- Receive ACK, send nothing
- Receive FIN, send ACK
- Receive FIN, send nothing

- Client application initiates a TCP connection
- Send SYN
- Receive SYN & ACK, send ACK
- Send FIN
- Receive ACK
- Send nothing
Congestion:

• informally: “too many sources sending too much data too fast for network to handle”

• different from flow control!

• manifestations:
  – lost packets (buffer overflow at routers)
  – long delays (queueing in router buffers)

• a top-10 problem!
Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission

\[ \lambda_{\text{out}} \sim \frac{C}{2} \]

\[ \lambda_{\text{in}} \sim \frac{C}{2} \]

large delays when congested
maximum achievable throughput

\[ \lambda_{\text{in}} \sim \frac{C}{2} \]
Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet
Causes/costs of congestion: scenario 2

- always: $\lambda_{\text{in}} = \lambda_{\text{out}}$ (goodput)
- “perfect” retransmission only when loss: $\lambda'_{\text{in}} > \lambda_{\text{out}}$
- retransmission of delayed (not lost) packet makes larger (than perfect case) for same $\lambda_{\text{in}}$

“costs” of congestion:
- more work (retrans) for given “goodput”
- unneeded retransmissions: link carries multiple copies of pkt
Causes/costs of congestion: scenario 3

- four senders
- Multi-hop paths
- timeout/retransmit

Q: what happens as $\lambda_{in}$ increase?

Finite shared output link buffers

\( \lambda_{in} : \text{original data} \)
\( \lambda'_{in} : \text{original data, plus retransmitted data} \)

Host A

Host B

$\lambda_{out}$
Causes/costs of congestion: scenario 3

Another “cost” of congestion:

- when packet dropped, any “upstream transmission capacity used for that packet was wasted!
Two broad approaches towards congestion control:

End-end congestion control:
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:
- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate the sender should send
**TCP congestion control:** additive increase, multiplicative decrease

**Approach:** increase transmission rate (window size), probing for usable bandwidth, until loss occurs

- **additive increase:** increase $\text{CongWin}$ by 1 MSS every RTT until loss detected
- **multiplicative decrease:** cut $\text{CongWin}$ in half after loss

Saw tooth behavior: probing for bandwidth

![Diagram showing the saw tooth behavior of TCP congestion window size over time](chart.png)
TCP Congestion Control: details

• sender limits transmission: 
  \[ \text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin} \]

• Roughly, 
  \[ \text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec} \]

• \text{CongWin} is dynamic, function of perceived network congestion

How does sender perceive congestion? 
loss event = timeout or 3 duplicate acks
TCP sender reduces rate (\text{CongWin}) after loss event

three mechanisms:
  AIMD
  slow start
  conservative after timeout events
TCP Slow Start

- When connection begins, $\text{CongWin} = 1$ MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
    - initial rate = 20 kbps
- available bandwidth may be $>>$ MSS/RTT
  - desirable to quickly ramp up to respectable rate

When connection begins, increase rate exponentially fast until first loss event
• When connection begins, increase rate exponentially until first loss event:
  – double $\text{CongWin}$ every RTT
  – done by incrementing $\text{CongWin}$ for every ACK received

• **Summary:** initial rate is slow but ramps up exponentially fast
• After 3 dup ACKs:
  – $\text{CongWin}$ is cut in half
  – window then grows linearly

• But after timeout event:
  – $\text{CongWin}$ instead set to 1 MSS;
  – window then grows exponentially
  – to a threshold, then grows linearly

Philosophy:
- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a “more alarming” congestion scenario
Q: When should the exponential increase switch to linear?

A: When CongWin gets to 1/2 of its value before timeout.

**Implementation:**

Variable Threshold

At loss event, Threshold is set to 1/2 of CongWin just before loss event
Summary: TCP Congestion Control

- When \text{CongWin} is below \text{Threshold}, sender is in \text{slow-start} phase, window grows exponentially.

- When \text{CongWin} is above \text{Threshold}, sender is in \text{congestion-avoidance} phase, window grows linearly.

- When a \text{triple duplicate ACK} occurs, \text{Threshold} set to \text{CongWin}/2 and \text{CongWin} set to \text{Threshold}.

- When \text{timeout} occurs, \text{Threshold} set to \text{CongWin}/2 and \text{CongWin} is set to 1 MSS.
## TCP sender congestion control

<table>
<thead>
<tr>
<th>State</th>
<th>Event</th>
<th>TCP Sender Action</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow Start (SS)</td>
<td>ACK receipt for previously unacked data</td>
<td>CongWin = CongWin + MSS, If (CongWin &gt; Threshold) set state to “Congestion Avoidance”</td>
<td>Resulting in a doubling of CongWin every RTT</td>
</tr>
<tr>
<td>Congestion Avoidance (CA)</td>
<td>ACK receipt for previously unacked data</td>
<td>CongWin = CongWin + MSS * (MSS/CongWin)</td>
<td>Additive increase, resulting in increase of CongWin by 1 MSS every RTT</td>
</tr>
<tr>
<td>SS or CA</td>
<td>Loss event detected by triple duplicate ACK</td>
<td>Threshold = CongWin/2, CongWin = Threshold, Set state to “Congestion Avoidance”</td>
<td>Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.</td>
</tr>
<tr>
<td>SS or CA</td>
<td>Timeout</td>
<td>Threshold = CongWin/2, CongWin = 1 MSS, Set state to “Slow Start”</td>
<td>Enter slow start</td>
</tr>
<tr>
<td>SS or CA</td>
<td>Duplicate ACK</td>
<td>Increment duplicate ACK count for segment being acked</td>
<td>CongWin and Threshold not changed</td>
</tr>
</tbody>
</table>
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K
Why is TCP fair?

Two competing sessions:
- Additive increase gives slope of 1, as throughput increases
- Multiplicative decrease decreases throughput proportionally

Equal bandwidth share

Loss: decrease window by factor of 2
Congestion avoidance: additive increase

Connection 1 throughput vs. Connection 2 throughput
Fairness and UDP

- Multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

Fairness and parallel TCP connections

nothing prevents app from opening parallel connections between 2 hosts.

Web browsers do this

Example: link of rate R supporting 9 connections;
new app asks for 1 TCP, gets rate R/10
new app asks for 11 TCPs, gets R/2!