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
- SampleRTT will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current SampleRTT

TCP Round Trip Time and Timeout

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

- Exponential weighted moving average
- influence of past sample decreases exponentially fast

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 sender events:

- **data rcvd from app:**
  - Create segment with seq #
  - seq # is byte-stream number of first data byte in segment
  - start timer if not already running (think of timer as for oldest unacked segment)
  - expiration interval: TimeOutInterval

- **timeout:**
  - retransmit segment that caused timeout
  - restart timer

- **Ack rcvd:**
  - If acknowledges previously unacked segments
    - update what is known to be acked
    - start timer if there are outstanding segments

TCP sender (simplified):

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)
  event: data received from application above
    create TCP segment with sequence number NextSeqNum
    if (timer currently not running)
      start timer
    pass segment to IP
    NextSeqNum = NextSeqNum + length(data)
  event: timer timeout
    retransmit not-yet-acknowledged segment with smallest sequence number
    start timer
  event: ACK received, with ACK field value of y
    if (y > SendBase) {
      SendBase = y
      if (there are currently not-yet-acknowledged segments)
        start timer
    }
} /* end of loop forever */

TCP retransmission scenarios

Example: SendBase-1: last cumulatively acked byte
- SendBase-1 = 71; y = 73, so the rcvr wants 73+; y > SendBase, so new data is acked

Comments:
- SendBase-1: last cumulatively acked byte
- Examples:
  - SendBase-1 = 71; y = 73, so the rcvr wants 73+; y > SendBase, so new data is acked
TCP retransmission scenarios (more)

- **Host A**
  - Seq = 92, 8 bytes data
  - ACK = 100
  - Loss
  - Timeout

- **Host B**
  - X
  - Seq = 100, 20 bytes data
  - ACK = 120

TCP ACK generation [RFC 1122, RFC 2581]

<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 #. All data up to expected seq # already ACKed</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 #. Gap detected</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>

Fast Retransmit

- Time-out period often relatively long:
  - Long delay before resending lost packet
- Detect lost segments via duplicate ACKs:
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - **Fast retransmit**: resend segment before timer expires

---
Fast retransmit algorithm:

- event: ACK received, with ACK field value of y
  - SendBase = y
  - if (there are currently not-yet-acknowledged segments)
    - start timer
  - else
    - increment count of dup ACKs received for y
    - if (count of dup ACKs received for y = 3)
      - resend segment with sequence number y

TCP Flow Control:
- receive side of TCP connection has a receive buffer:
  - 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 RcvrWindow, guarantees receive buffer doesn’t overflow
  - spare room in buffer
    - RcvrWindow = RcvBuffer - [LastByteRcvd - LastByteRead]
TCP Connection Management

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 = welcomeSocket.accept();

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

TCP Connection Management (cont.)

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.

Step 3: Client receives ACK, replies with FIN.

TCP Connection Management (cont.)

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

Principles of Congestion Control

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

- large delays when congested
- maximum achievable throughput
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 \( \lambda_{\text{in}}' \) larger
  (than perfect case) for same \( \lambda_{\text{out}} \)

"costs" of congestion:
- more work (retransmission) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt

Causes/costs of congestion: scenario 3
- four senders
- multihop paths
- timeout/retransmit

Q: what happens as \( \lambda_{\text{in}} \) and \( \lambda_{\text{in}}' \) increase?
Causes/costs of congestion: scenario 3

Another "cost" of congestion:
- when packet dropped, any upstream transmission capacity used for that packet was wasted.

Approaches towards congestion control

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 sender should send at

TCP Congestion Control

- end-end control (no network assistance)
- sender limits transmission:
  \[ \text{LastByteSent - LastByteAcked} \leq \text{CongWin} \]
- Roughly:
  \[ \text{rate} = \frac{\text{CongWin \times Bytes/sec}}{\text{RTT}} \]
- CongWin is dynamic, function of perceived network congestion

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

three mechanisms:
- AIMD
- slow start
- conservative after timeout events
TCP AIMD

- Multiplicative decrease: cut CongWin in half after loss event
- Additive increase: increase CongWin by 1 MSS every RTT in the absence of loss events: probing

TCP Slow Start

- When connection begins, 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

TCP Slow Start (more)

- When connection begins, increase rate exponentially fast until first loss event:
  - Double CongWin every RTT
  - Done by incrementing CongWin for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast
Refinement

- After 3 dup ACKs:
  - CongWin is cut in half
  - window then grows linearly
- But after timeout event:
  - 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 before 3 dup ACKs is "more alarming"

Refinement (more)

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 CongWin is below Threshold, sender in slow-start phase, window grows exponentially.
- When CongWin is above Threshold, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, Threshold set to CongWin/2 and CongWin set to Threshold.
- When timeout occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS.
TCP Fairness

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

Fairness (more)

- 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
- 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/21
**Delay modeling**

Q: How long does it take to receive an object from a Web server after sending a request?

Ignoring congestion, delay is influenced by:
- TCP connection establishment
- data transmission delay
- slow start

Notation, assumptions:
- Assume one link between client and server of rate $R$
- $S$: MSS (bits)
- $O$: object size (bits)
- no retransmissions (no loss, no corruption)

Window size:
- First assume: fixed congestion window, window = $W$ segments
- Then dynamic window, modeling slow start

---

**Fixed congestion window (1)**

First case:
$$\frac{WS}{R} > RTT + \frac{S}{R}$$

ACK for first segment in window returns before window’s worth of data sent

$$\text{delay} = 2RTT + \frac{O}{R}$$

---

**Fixed congestion window (2)**

Second case:
$$\frac{WS}{R} < RTT + \frac{S}{R}$$

wait for ACK after sending window’s worth of data sent

$$\text{delay} = 2RTT + \frac{O}{R} + (K-1)(\frac{S}{R} + RTT - WS/R)$$
TCP Delay Modeling: Slow Start (1)

Now suppose window grows according to slow start

Will show that the delay for one object is:

\[
\text{Latency} = 2\text{RTT} + \frac{Q}{R} + \frac{P}{R} \left( \frac{\text{RTT}}{R} - \frac{S}{R} \right) \left( 2^k - 1 \right) \frac{S}{R}
\]

where \( P \) is the number of times TCP idles at server:

\[ P = \min(Q, K - 1) \]

- where \( Q \) is the number of times the server idles if the object were of infinite size.
- and \( K \) is the number of windows that cover the object.

TCP Delay Modeling: Slow Start (2)

Delay components:
- \( 2\text{RTT} \) for connection estab and request
- \( Q/R \) to transmit object
- time server idles due to slow start

Server idles:
\[ P = \min(K - 1, Q) \]

Example:
- \( Q = 15 \) segments
- \( K = 4 \) windows
- \( Q = 2 \)
- \( P = \min(K, Q) = 2 \)

Server idles \( P = 2 \) times

TCP Delay Modeling (3)

\[
\text{delay} = \frac{Q}{R} 2\text{RTT} + \frac{\text{sum of } \text{time}}{R} + \frac{\text{sum of } \text{time}}{R} \left( \frac{\text{RTT}}{R} - \frac{S}{R} \right) \text{ for } \sum{\text{time}} \]

\[
= \frac{Q}{R} 2\text{RTT} + \frac{\text{sum of } \text{time}}{R} + \frac{\text{sum of } \text{time}}{R} \left( 2^k - 1 \right) \frac{S}{R}
\]

\[
= \frac{Q}{R} 2\text{RTT} + \frac{\text{sum of } \text{time}}{R} + \frac{\text{sum of } \text{time}}{R} \left( 2^k - 1 \right) \frac{S}{R}
\]
TCP Delay Modeling (4)

Recall $K = \text{number of windows that cover object}$

How do we calculate $K$?

$$
K = \min( k : 2S + 2S + \cdots + 2^kS \geq O )
= \min( k : 2S + 2S + \cdots + 2^kS \geq O/S )
= \min( k : 2^k - 1 \geq \frac{O}{S} )
= \min( k : k \geq \log_2(\frac{O}{S} + 1) )
= \left\lfloor \log_2(\frac{O}{S} + 1) \right\rfloor
$$

Calculation of $Q$, number of idles for infinite-size object, is similar.