TCP: Overview

- **point-to-point:**
  - one sender, one receiver

- **reliable, in-order byte steam:**
  - no "message boundaries"

- **full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size

- **connection-oriented:**
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange

- **flow controlled:**
  - sender will not overwhelm receiver

---

TCP segment structure

- URG: urgent data (generally not used)
- ACK: ACK #
  - valid
- PSH: push data now (generally not used)
- RST, SYN, FIN: connection estab (setup, teardown commands)
- Internet checksum (as in UDP)
- Options (variable length)
- application data (variable length)
- source port #
- dest port #
- sequence number
- acknowledgement number
- rcvr window size
- ptr urgent data
- checksum
- head
- not used
- TCP

---
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
- cumulative ACK

Q: how receiver handles out-of-order segments
- A: TCP spec doesn’t say, - up to implementor

TCP: reliable data transfer

**event:** data received from application above
- create, send segment

**event:** timer timeout for segment with seq # y
- retransmit segment

**event:** ACK received, with ACK # y
- ACK processing

simplified sender, assuming
- one way data transfer
- no flow, congestion control
TCP: reliable data transfer

Simplified TCP sender

<table>
<thead>
<tr>
<th>Event</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-order segment arrival, no gaps, everything else already ACKed</td>
<td>delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>in-order segment arrival, no gaps, one delayed ACK pending</td>
<td>immediately send single cumulative ACK</td>
</tr>
<tr>
<td>out-of-order segment 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 gap</td>
<td>immediate ACK if segment starts at lower end of gap</td>
</tr>
</tbody>
</table>
TCP: retransmission scenarios

TCP Flow Control

**Flow control**
- **sender** won't overrun receiver's buffers by transmitting too much, too fast

(receiver: explicitly informs sender of (dynamically changing) amount of free buffer space)
- **RcvWindow** field in TCP segment

(sender: keeps the amount of transmitted, unACKed data less than most recently received **RcvWindow**)

**RcvBuffer**: size of TCP Receive Buffer

**RcvWindow**: amount of spare room in Buffer

**RcvBuffer** = size of TCP Receive Buffer

**RcvWindow** = amount of spare room in Buffer

data from IP

TCP data in buffer

application process

receiver buffering
**TCP Round Trip Time and Timeout**

**Q: how to set TCP timeout value?**
- longer than RTT
  - note: RTT will vary
- 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, cumulatively ACKed segments
- **SampleRTT** will vary, want estimated RTT "smoother"
  - use several recent measurements, not just current **SampleRTT**

\[
\text{EstimatedRTT} = (1-x) \times \text{EstimatedRTT} + x \times \text{SampleRTT}
\]
- Exponential weighted moving average
- influence of given sample decreases exponentially fast
- typical value of \( x \): 0.1

**Setting the timeout**
- **EstimatedRTT** plus "safety margin"
- large variation in **EstimatedRTT** → larger safety margin

\[
\text{Timeout} = \text{EstimatedRTT} + 4 \times \text{Deviation}
\]

\[
\text{Deviation} = (1-x) \times \text{Deviation} + x \times |\text{SampleRTT} - \text{EstimatedRTT}|
\]
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 end system sends TCP SYN control segment to server
  - specifies initial seq #

Step 2: server end system receives SYN, replies with SYNACK control segment
  - ACKs received SYN
  - allocates buffers
  - specifies server->receiver initial seq #

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. Closes connection, sends 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**

---

3: Transport Layer 3b-13

3: Transport Layer 3b-14
**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

![Diagram of network with nodes and data traffic]

always: $\lambda_{in} = \lambda_{out}$ (goodput)

"perfect" retransmission only when loss: $\lambda_{in}' > \lambda_{out}$

retransmission of delayed (not lost) packet makes $\lambda_{in}'$ larger (than perfect case) for same $\lambda_{out}$

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

3: Transport Layer 3b-17
Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

Q: what happens as $\lambda_{\text{in}}$ and $\lambda'_{\text{in}}$ increase?

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

**Case study: ATM ABR congestion control**

**ABR: available bit rate:**
- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender throttled to minimum guaranteed rate

**RM (resource management) cells:**
- sent by sender, interspersed with data cells
- bits in RM cell set by switches ("network-assisted")
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- RM cells returned to sender by receiver, with bits intact
Case study: ATM ABR congestion control

- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender's send rate thus minimum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

TCP Congestion Control

- end-end control (no network assistance)
- transmission rate limited by congestion window size, Congwin, over segments:

\[ \text{throughput} = \frac{w \times \text{MSS}}{\text{RTT}} \text{ Bytes/sec} \]
TCP congestion control:

- “probing” for usable bandwidth:
  - ideally: transmit as fast as possible (Congwin as large as possible) without loss
  - increase Congwin until loss (congestion)
  - loss: decrease Congwin, then begin probing (increasing) again

- two “phases”
  - slow start
  - congestion avoidance

- important variables:
  - Congwin
  - threshold: defines threshold between two slow start phase, congestion control phase

TCP Slowstart

Slowstart algorithm:

initialize: Congwin = 1
for (each segment ACKed) Congwin++
until (loss event OR CongWin > threshold)

- exponential increase (per RTT) in window size (not so slow!)
- loss event: timeout (Tahoe TCP) and/or or three duplicate ACKs (Reno TCP)
TCP Congestion Avoidance

/* slowstart is over */
/* Congwin > threshold */
Until (loss event) {
    every w segments ACKed:
        Congwin++
}
threshold = Congwin/2
Congwin = 1
perform slowstart

1: TCP Reno skips slowstart (fast recovery) after three duplicate ACKs

AIMD

TCP congestion avoidance:
- **AIMD:** additive increase, multiplicative decrease
  - increase window by 1 per RTT
  - decrease window by factor of 2 on loss event

TCP Fairness

Fairness goal: if N TCP sessions share same bottleneck link, each should get 1/N of link capacity
Why is TCP fair?

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

TCP latency modeling

Q: How long does it take to receive an object from a Web server after sending a request?
- TCP connection establishment
- Data transfer delay

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

Two cases to consider:
- $WS/R > RTT + S/R$: ACK for first segment in window returns before window's worth of data sent
- $WS/R < RTT + S/R$: wait for ACK after sending window's worth of data sent
TCP Latency Modeling

Case 1: latency = 2RTT + O/R

Case 2: latency = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]

K := O/WS

TCP Latency Modeling: Slow Start

- Now suppose window grows according to slow start.
- Will show that the latency of one object of size O is:

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

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

\[ P = \min\{Q, K-1\} \]

- where \( Q \) is the number of times the server would stall if the object were of infinite size.
- and \( K \) is the number of windows that cover the object.
**TCP Latency Modeling: Slow Start (cont.)**

Example:

- O/S = 15 segments
- K = 4 windows
- Q = 2
- P = min(K-1,Q) = 2

Server stalls P=2 times.

\[
\frac{S}{R} + RTT = \text{time from when server startstosend segment until server receives acknowledgment}
\]

\[
2^{k-1} \frac{S}{R} = \text{time to transmit the kth window}
\]

\[
\left[ \frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right] = \text{stall time after the kth window}
\]

\[
\text{latency} = \frac{O}{R} + 2RTT + \sum_{p=1}^{P} \text{stallTime}_p
\]

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

\[
= \frac{O}{R} + 2RTT + P[RTT + \frac{S}{R}] - (2^p - 1) \frac{S}{R}
\]
Chapter 3: Summary

- principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP

Next:
- leaving the network “edge” (application transport layer)
- into the network “core”