**TCP: Overview**

RFCs: 793, 1122, 1323, 2018, 2581

- **point-to-point:**
  - one sender, one receiver
- **reliable, in-order byte stream:**
  - no "message boundaries"
- **pipelined:**
  - TCP congestion and flow control set window size
- **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**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>16</td>
</tr>
<tr>
<td>dest port #</td>
<td>16</td>
</tr>
<tr>
<td>sequence number</td>
<td>32</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>32</td>
</tr>
<tr>
<td>Urgent</td>
<td>1</td>
</tr>
<tr>
<td>Ack</td>
<td>1</td>
</tr>
<tr>
<td>PSH</td>
<td>1</td>
</tr>
<tr>
<td>RST, SYN, FIN</td>
<td>1</td>
</tr>
<tr>
<td>connection estab</td>
<td>1</td>
</tr>
<tr>
<td>(setup, teardown commands)</td>
<td></td>
</tr>
<tr>
<td>Internet checksum</td>
<td>16</td>
</tr>
<tr>
<td>(as in UDP)</td>
<td></td>
</tr>
<tr>
<td>Options (variable length)</td>
<td></td>
</tr>
<tr>
<td>application data</td>
<td></td>
</tr>
<tr>
<td>(variable length)</td>
<td></td>
</tr>
</tbody>
</table>

- **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)
  - **counting by bytes of data (not segments!):**
  - **# bytes:** rcvr willing to accept
  - **counting by bytes of data:** (not segments!)
  - **# bytes:** rcvr willing to accept
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: what receiver does w/ out-of-order segments
- TCP spec doesn't say, - up to TCP implementor

TCP: reliable data transfer

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

3: Transport Layer 3b-3
3: Transport Layer 3b-4
TCP: reliable data transfer

Simplified TCP sender

00  sendbase = initial_sequence number
01  nextseqnum = initial_sequence number
02  loop (forever) {
03      switch(event)
04          event: data received from application above
05              create TCP segment with sequence number nextseqnum
06              start timer for segment nextseqnum
07              pass segment to IP
08              nextseqnum = nextseqnum + length(data)
09          event: timer timeout for segment with sequence number y
10              retransmit segment with sequence number y
11              compute new timeout interval for segment y
12              restart timer for sequence number y
13          event: ACK received, with ACK field value of y
14              if (y > sendbase) {
15                  /* cumulative ACK of all data up to y */
16                      cancel all timers for segments with sequence numbers < y
17                      sendbase = y
18              }
19              else {
20                  /* a duplicate ACK for already ACKed segment */
21                      increment number of duplicate ACKs received for y
22                      if (number of duplicate ACKs received for y == 3) {
23                          /* TCP fast retransmit */
24                          resend segment with sequence number y
25                          restart timer for segment y
26                      }
27                  }
28          } /* end of loop forever */

TCP ACK generation [RFC 1122, RFC 2581]

<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 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>everything else already ACKed</td>
<td></td>
</tr>
<tr>
<td>in-order segment arrival, no gaps,</td>
<td>immediately send single cumulative ACK</td>
</tr>
<tr>
<td>one delayed ACK pending</td>
<td></td>
</tr>
<tr>
<td>out-of-order segment arrival</td>
<td>send duplicate ACK, indicating seq. # of next expected (missing) byte</td>
</tr>
<tr>
<td>higher-than-expect seq. #</td>
<td></td>
</tr>
<tr>
<td>gap detected</td>
<td></td>
</tr>
<tr>
<td>arrival of segment that</td>
<td>immediate ACK if segment starts at lower end of gap</td>
</tr>
<tr>
<td>partially or completely fills gap</td>
<td></td>
</tr>
</tbody>
</table>

3: Transport Layer  3b-5

3: Transport Layer  3b-6
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
    - rcvr window size field in TCP segment

- receiver buffering

- sender: amount of transmitted, unACKed data less than most recently-received rcvr window size
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
  - or timestamp pkts
- SampleRTT will vary, want estimated RTT “smoother”
  - use several recent measurements, not just current SampleRTT

EstimatedRTT = (1-x)*EstimatedRTT + x*SampleRTT
- Exponential weighted moving average
- influence of given sample decreases exponentially fast
- typical value of x: 0.1

Setting the timeout
- RTT plus “safety margin”
- large variation in EstimatedRTT -> larger safety margin

Timeout = EstimatedRTT + 4*Deviation

Deviation = (1-x)*Deviation + x*abs(SampleRTT-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 #

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 showing causes of congestion]

3: Transport Layer 3b-17

---

Causes/costs of congestion: scenario 2

- always: \( \lambda_{in} = \lambda_{out} \) (goodput), but can have \( \lambda'_{in} > \lambda_{out} \)
- "perfect" when only retransmit lost pkts
- retransmission of delayed (not lost) packet makes \( \lambda'_{in} \) larger (than perfect case) for same \( \lambda_{out} \)

![Graphs showing different scenarios]

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

3: Transport Layer 3b-18
Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

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

3: Transport Layer 3b-19

Causes/costs of congestion: scenario 3

Another “cost” of congestion:
- when packet dropped, any “upstream transmission capacity used for that packet was wasted!”

3: Transport Layer 3b-20
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
- EFCl bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCl set, CI bit set in returned RM cell

TCP Congestion Control

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

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

- w segments, each with MSS bytes sent in one RTT:
- Congestion Window never larger than rcvr-advertised window
**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 slow start phase and 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 ( Tahoe )

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

TCP Reno

- Most of today ’ s TCPs are Reno
- Same behavior as Tahoe on timeout
- On Triple-duplicate ACK:
  - Tahoe does nothing ( no window change )
  - Reno:
    - Threshold = congwin / 2
    - congwin = congwin / 2
**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 throughput increases
- multiplicative decrease decreases throughput proportionally
Effects of TCP latencies

Q: client latency from WWW server to receipt?
   □ TCP connection establishment
   □ data transfer delay

Notation, assumptions:
   □ Assume: fixed congestion window, W, giving throughput of R bps
   □ 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 before window’s worth of data sent
   □ WS/R < RTT + S/R: wait for ACK after sending window’s worth of data sent

3: Transport Layer 3b-31
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"