Transmission Control Protocol (TCP)
Introduction
TCP: Overview

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

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

- **pipelined:**
  - TCP congestion and flow control set window size

- **send & receive buffers**

- **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

- **source port #**
- **dest port #**
- **sequence number**
- **acknowledgement number**
- **rcvr window size**
- **checksum**
- **ptr urgent data**
- **Options (variable length)**
- **application data** (variable length)

**Fields and Options:**
- **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)

**Not Used Fields:**
- **len**
- **URP**
- **RSF**
- **not used**

**Counting:**
- # bytes rcvr willing to accept

**Notes:**
- Internet checksum as in UDP
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 implementation

simple telnet scenario
TCP: reliable data transfer

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

event: data received from application above
create, send segment

wait for event

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

event: ACK received, with ACK # y
ACK processing
TCP: reliable data transfer

Simplified TCP sender

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

Host A
Seq=92, 8 bytes data
ACK=100

Host B
X
Seq=92, 8 bytes data
ACK=100

Host A
Seq=100, 20 bytes data
ACK=100
Seq=92 timeout

Host B
Seq=92, 8 bytes data
ACK=120
Seq=92, 8 bytes data
ACK=120
Seq=100, 20 bytes data
ACK=100
ACK=120

lost ACK scenario

premature timeout, cumulative ACKs
TCP Flow Control

**sender:** won’t overrun receiver’s buffers by transmitting too much, too fast

- RcvBuffer = size or TCP Receive Buffer
- RcvWindow = amount of spare room in Buffer

**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

**receiver buffering**

**flow control**
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
TCP Round Trip Time and Timeout

EstimatedRTT = (1-x)*EstimatedRTT + x*SampleRTT

- Exponential weighted moving average
- Influence of given sample decreases exponentially fast
- Typical value of x: 0.125 (1/8)

Setting the timeout

- EstimatedRTT plus “safety margin”
- Large variation in EstimatedRTT -> larger safety margin

Timeout = EstimatedRTT + 4*Deviation

Deviation = (1-x)*Deviation + x|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
  
  ```java
  Socket clientSocket = new Socket("hostname","port number");
  ```

- **server**: contacted by client
  
  ```java
  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. #

- **Step 3**: client receives SYNACK control segment
  - ACKs received SYN
  - SYN=0, connection has been established.
  - Client data may be piggybacked
TCP Connection Setup

- Connection request (SYN=1, seq=client_isn)
- Connection granted (SYN=1, seq=server_isn, ack=client_isn+1)
- ACK (SYN=0, seq=client_isn+1, ack=server_isn+1)
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:
- CLOSED
  - wait 30 seconds
  - receive FIN, send ACK
- TIME_WAIT
  - receive FIN, send ACK
- FIN_WAIT_2
  - receive ACK, send nothing
- FIN_WAIT_1
  - send FIN
- ESTABLISHED
  - client application initiates close connection
- SYN_SENT
  - receive SYN & ACK, send ACK
- SYN_RCVD
  - receive ACK, send nothing

TCP server lifecycle:
- CLOSED
  - server application creates a listen socket
- LISTEN
  - receive SYN, send SYN & ACK
- CLOSE_WAIT
  - send FIN
- ESTABLISHED
  - client application initiates a TCP connection
- SYN_SENT
  - receive SYN & ACK, send ACK