Communication Networks
(0368-3030) / Fall 2013
The Blavatnik School of Computer Science, Tel-Aviv University

Allon Wagner
TCP Overview

Kurose & Ross, Chapter 3 (5th ed.)

Many slides adapted from:
J. Kurose & K. Ross \nComputer Networking: A Top Down Approach (5th ed.)
Addison-Wesley, April 2009.
Copyright 1996-2010, J.F Kurose and K.W. Ross, All Rights Reserved.
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) initiates sender, receiver state before data exchange

- **flow controlled:**
  - sender will not overwhelm receiver
TCP segment structure

- **Source port #**
- **Destination port #**
- **Sequence number**
- **Acknowledgement number**
- **Receive window**
- **Checksum**
- **Urgent data pointer**
- **Options** (variable length)
- **Application data** (variable length)

- **URG**: urgent data (generally not used)
- **ACK**: ACK # valid
- **PSH**: push data now (generally not used)
- **RST, SYN, FIN**: connection establishment (setup, teardown commands)
- **Internet checksum** (as in UDP)

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

Host A
- User types 'C'
- Seq=42, ACK=79, data = ‘C’
- host ACKs receipt of ‘C’, echoes back ‘C’

Host B
- Seq=79, ACK=43, data = ‘C’
- Seq=43, ACK=80
- simple telnet scenario
TCP Round Trip Time and Timeout

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
- Typical value: \(\alpha = 0.125\)
Example RTT estimation:

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

SampleRTT Estimated RTT
TCP Round Trip Time and Timeout

Setting the timeout

- EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT -> larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

  \[ \text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}| \]

  (typically, \( \beta = 0.25 \))

Then set timeout interval:

\[ \text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT} \]
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 host sends TCP SYN segment to server
  - specifies initial seq #
  - no data

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

- **Step 3**: client receives SYNAACK, replies with ACK segment, which may contain data
Three-way handshake

SYN = 1 (SEQ = x)

SYN = 1 ACK = 1
(SEQ = y, ACK = x+1)

ACK = 1
(SEQ = x+1, ACK = y+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

TCP server lifecycle

CLOSED

TIME_WAIT

SYN_SENT

FIN_WAIT_1

ESTABLISHED

client application initiates a TCP connection

send SYN

receive SYN & ACK send ACK

send FIN

receive ACK send nothing

TCP client lifecycle

TCP server lifecycle

wait 30 seconds

receive FIN send nothing

listen socket

receive SYN send SYN & ACK

receive ACK send nothing
TCP’s statechart

• On board
  ▫ Statechart appears in RFC 793
• Discussion of:
  ▫ TIME_WAIT state
   • Connection in TIME_WAIT state cannot move to the CLOSED state until it has waited for two times the maximum segment lifetime (MSL).
   • Why? We do not know whether the ack sent in response to the other side’s FIN was delivered. The other side might retransmit its FIN segment.
   • This second FIN might be delayed in the network. If the connection were allowed to move directly to the CLOSED state, then another pair of application processes could have opened the same connection (i.e., use the same port numbers).
   • The delayed FIN from the previous incarnation terminates the later incarnation of the same connection.
   • Because only a connection between the same endpoints can cause the confusion, only one endpoint needs to hold the state.
  ▫ Syn flood attacks
# 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 <em>duplicate ACK</em>, 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
Figure 3.37 Resending a segment after triple duplicate ACK
Fast retransmit algorithm:

- **Event**: ACK received, with ACK field value of $y$
  - If ($y > \text{SendBase}$) {
    - $\text{SendBase} = y$
    - 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 $y$
    }
  }

A duplicate ACK for already ACKed segment

Fast retransmit
**TCP Flow Control**

- receive side of TCP connection has a receive buffer:
  - flow control
    - 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 RcvWindow
  - guarantees receive buffer doesn't overflow

(suppose TCP receiver discards out-of-order segments)

- spare room in buffer
  - \( RcvWindow \)
  - \( RcvBuffer - \text{LastByteRcvd} - \text{LastByteRead} \)