TCP: Overview

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

TCP segment structure

- **Sequence #s:**
  - "number" of first byte in segment's data
- **ACKs:**
  - seq # of next byte expected from other side
  - cumulative ACK
- **Options:**
  - variable length
- **URG:** urgent data (generally not used)
- **ACK:** valid
- **PSH:** push data now (generally not used)
- **RST, SYN, FIN:** connection estab (setup, teardown commands)
- **Options:** (variable length)

TCP Round Trip Time and Timeout

- **Q:** how to set TCP timeout value?
  - longer than RTT
  - too short: premature timeout
  - 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"
TCP Round Trip Time and Timeout

EstimatedRTT = (1- \( \alpha \)) * EstimatedRTT + \( \alpha \) * SampleRTT

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: \( \alpha = 0.125 \)

Example RTT estimation:

```
+-------------------+-------------------+
|                   |                   |
|  time (seconds)   |  RTT (milliseconds)|
|                   |                   |
|  0                | 100               |
|  10               | 150               |
|  20               | 200               |
|  30               | 250               |
|  40               | 300               |
```

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
  - 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

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'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 reliable data transfer**

- TCP creates rdt 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
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
    NextSeqNum = NextSeqNum + length(data)
    pass segment to IP
    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
    }
}

TCP sender
(simplified)

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

TCP: retransmission scenarios

Host A
Seq=92
time
timeout
Host B

SendBase = 120
ACK=100
time
Cumulative ACK scenario

TCP retransmission scenarios (more)

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver

<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

Figure 3.37 Resending a segment after triple duplicate ACK
Fast retransmit algorithm:

- **event**: ACK received, with ACK field value of \( y \)
  - if \( y > \text{SendBase} \)
    - 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

- Receiver 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

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP Flow control: how it works

- Receiver advertises spare room by including value of RcvWindow in segments
- Sender limits unACKed data to RcvWindow
  - Guarantees receive buffer doesn't overflow
- Spare room in buffer
  - \( \text{RcvWindow} = \text{RcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead}) \)