**Computer Networks**

**Connection-Oriented Transport: TCP**

Based on Computer Networking, 4th Edition by Kurose and Ross

---

**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

---

![Diagram of TCP connection](image)
TCP segment structure

<table>
<thead>
<tr>
<th>Source port #</th>
<th>Dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>Head</td>
<td>Len</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urg data pointer</td>
</tr>
<tr>
<td>Receive window</td>
<td></td>
</tr>
<tr>
<td>Options (variable length)</td>
<td></td>
</tr>
<tr>
<td>Application data</td>
<td>(variable length)</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)
- # bytes rcvr willing to accept
- Seq. # bytes
- Counting by bytes of data (not segments!)

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'
Host B: Host ACKs receipt of 'C', echoes back 'C'

Simple telnet scenario

Time

Stan Kurkovsky
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
    EstimatedRTT = (1- α)EstimatedRTT + α*SampleRTT
  - Exponential weighted moving average
    - influence of past sample decreases exponentially fast
    - typical value: α = 0.125

Setting the timeout
- EstimatedRTT plus "safety margin"
  - large variation in EstimatedRTT → larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:
  - DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
    (typically, β = 0.25)

Then set timeout interval:
- TimeoutInterval = EstimatedRTT + 4*DevRTT
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
TCP retransmission scenarios

**Lost ACK scenario**
- Host A: Seq=92, 8 bytes data
- Host B: ACK=100
- Loss
- SendBase = 100
- Cumulative ACK scenario
- Host A: Seq=92, 8 bytes data
- Host B: Seq=100, 20 bytes data
- SendBase = 100
- ACK=100
- ACK=120
- Premature timeout

**Cumulative ACK scenario**
- Host A: Seq=92, 8 bytes data
- Host B: Seq=100, 20 bytes data
- SendBase = 100
- ACK=100
- ACK=120
- Cumulative ACK scenario
- Host A: Seq=92, 8 bytes data
- Host B: Seq=100, 20 bytes data
- SendBase = 120
- ACK=120
- Premature timeout
### TCP ACK generation

<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
**TCP Flow Control**

- sender won't overflow receiver's buffer by transmitting too much, too fast
- receive side of TCP connection has a receive buffer:
  
  ![Diagram of TCP Flow Control]

- app process may be slow at reading from buffer
- speed-matching service: matching the send rate to the receiving app's drain rate

---

**TCP Flow control: how it works**

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

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

![Diagram of TCP Flow Control: how it works]
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 SYNACK segment
- server allocates buffers
- specifies server initial seq. #

**Step 3:** client receives SYNACK, replies with ACK segment, which may contain data

---

TCP Connection Management (cont.)

**Closing a connection:**

client closes socket:
```java
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.