**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**

- **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

**RFCs:** 793, 1122, 1323, 2018, 2581

---

**TCP segment structure**

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

---

**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

---

**TCP: reliable data transfer**

- **sendbase** = initial_sequence_number
- **nextseqnum** = initial_sequence_number
- **loop (forever) {**
  - **event:** data received from application above
  - **create TCP segment with sequence number nextseqnum**
  - **start timer for segment nextseqnum**
  - **pass segment to IP**
  - **nextseqnum = nextseqnum + length(data)**
  - **event:** timer timeout for segment with sequence number y
    - **retransmit segment with sequence number y**
    - **compute new timeout interval for segment y**
    - **restart timer for sequence number y**
  - **event:** ACK received, with ACK field value of y
    - if (y > sendbase) {
      - /* cumulative ACK of all data up to y */
      - cancel all timers for segments with sequence numbers < y
      - **sendbase = y**
    } else {
      - /* a duplicate ACK for already ACKed segment */
      - **TCP fast retransmit**
      - **increment number of duplicate ACKs received for y**
      - **if (number of duplicate ACKs received for y == 3) {**
        - /* TCP fast retransmit */
        - **retransmit segment with sequence number y**
        - **restart timer for segment y**
      }**
    }
  **} /* 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

TCP Flow Control

**Flow control**

- sender won't overrun receiver's buffers by transmitting too much, too fast

\[\text{RcvBuffer} = \text{size of TCP Receive Buffer}\]
\[\text{RcvWindow} = \text{amount of spare room in Buffer}\]

**sender** keeps the amount of transmitted, unACKed data less than most recently received **RcvWindow**

**receiver** explicitly informs sender of (dynamically changing) amount of free buffer space

- **RcvWindow** field in TCP segment

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

\[
\text{EstimatedRTT} = (1-x) \times \text{EstimatedRTT} + x \times \text{SampleRTT}
\]

- Exponential weighted moving average
- Influence of given sample decreases exponentially fast
- Typical value of \(x\): 0.1

#### Setting the timeout

- \(\text{EstimatedRTT}\) plus "safety margin"
- Large variation in \(\text{EstimatedRTT}\) -> larger safety margin

\[
\text{Timeout} = \text{EstimatedRTT} + 4 \times \text{Deviation}
\]

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

### 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
- Server: contacted by client

```java
Socket clientSocket = new Socket("hostname", port number);
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:

```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.

#### Closing a connection (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

client application initiates a TCP connection
receive SYN
send SYN

client application initiates close connection
send FIN
receive FIN
send ACK

TCP client
TCP server

3: Transport Layer  3b-13