Transmission Control Protocol (TCP)

Srinidhi Varadarajan
TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- **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**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>head</td>
<td>Head field</td>
</tr>
<tr>
<td>len</td>
<td>Length field</td>
</tr>
<tr>
<td>UAP</td>
<td>Urgent data present</td>
</tr>
<tr>
<td>RSF</td>
<td>Reset flag</td>
</tr>
<tr>
<td>rcvr window size</td>
<td>Receiver window size</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>ptr urgent data</td>
<td>Pointer to urgent data</td>
</tr>
</tbody>
</table>

Options (variable length):

- application data (variable length)
- Internet checksum (as in UDP)

**Fields with special meanings:**

- **URG:** urgent data (generally not used)
- **ACK:** ACK # valid
- **PSH:** push data now (generally not used)
- **RST, SYN, FIN:** connection estab (setup, teardown commands)
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

lost ACK scenario

premature timeout, cumulative ACKs
TCP Flow Control

**flow control**
- sender won’t overrun receiver’s buffers by transmitting too much, too fast

**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`

**RcvBuffer** = size or TCP Receive Buffer

**RcvWindow** = amount of spare room in Buffer

![Diagram of receiver buffering](image)
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?
- \textit{SampleRTT}: measured time from segment transmission until ACK receipt
  - ignore retransmissions, cumulatively ACKed segments
- \textit{SampleRTT} will vary, want estimated RTT “smoother”
  - use several recent measurements, not just current \textit{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: 0.1

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

- **server:** contacted by client
  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

1. Connection request (SYN=1, seq=client_isn)
2. Connection granted (SYN=1, seq=server_isn, ack=client_isn+1)
3. ACK (SYN=0, seq=client_isn+1, ack=server_isn+1)
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.
TCP Connection Management (cont)

TCP client lifecycle

TCP server lifecycle