Transmission Control Protocol (TCP)

Introduction

TCP: Overview

- point-to-point:
  - one sender, one receiver
- reliable, in-order byte stream:
  - no "message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
- 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

- source port #
- dest port #
- sequence number
- acknowledgement number
- TCP checksum
- options (variable length)
- application data (variable length)

TCP seq. #’s and ACKs

Seq. #’s:
- byte stream "number" of first byte in segment’s data
- seq # of next byte expected from other side
- cumulative ACK: how receiver handles out-of-order segments
  - A: TCP spec doesn’t say, - up to implementation

TCP: reliable data transfer

00 sendbase = initial_sequence number
01 maxsegments = initial_sequence number
02 loop (forever) {
  03 event: data received from application above
    create, send segment
    wait for event
  04 event: timer timeout for segment with seq # y
    retransmit segment
  05 event: ACK received, with ACK # y
    ACK processing
    wait for event
    event: ACK received, with ACK # y
    retransmit segment
  06 event: data received from application above
    create, send segment
    wait for event
    event: timer timeout for segment with seq # y
    retransmit segment
  07 event: ACK received, with ACK # y
    sendbase = y
    cancel all timers for segments with sequence numbers < y
  08 event: ACK received, with ACK # y
    sendbase = y
    cancel all timers for segments with sequence numbers < y
  09 event: ACK received, with ACK # y
    sendbase = y
    cancel all timers for segments with sequence numbers < y
  10 event: ACK received, with ACK # y
    sendbase = y
    cancel all timers for segments with sequence numbers < y
  11 retransmit segment with sequence number y
  12 compute new timeout interval for segment y
  13 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 */
      increment number of duplicate ACKs received for y
      if (number of duplicate ACKs received for y == 3) {
        /* TCP fast retransmit */
        resend segment with sequence number y
        restart timer for segment y
      }
    }
  14 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 */
      increment number of duplicate ACKs received for y
      if (number of duplicate ACKs received for y == 3) {
        /* TCP fast retransmit */
        resend segment with sequence number y
        restart timer for segment y
      }
    }
  15 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 */
      increment number of duplicate ACKs received for y
      if (number of duplicate ACKs received for y == 3) {
        /* TCP fast retransmit */
        resend segment with sequence number y
        restart timer for segment y
      }
    }
  16 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 */
      increment number of duplicate ACKs received for y
      if (number of duplicate ACKs received for y == 3) {
        /* TCP fast retransmit */
        resend segment with sequence number y
        restart timer for segment y
      }
    }
  17 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 */
      increment number of duplicate ACKs received for y
      if (number of duplicate ACKs received for y == 3) {
        /* TCP fast retransmit */
        resend segment with sequence number y
        restart timer for segment y
      }
    }
}

TCP: reliable data transfer

Simplified TCP sender

- simplified sender, assuming:
  - one way data transfer
  - no flow, congestion control

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

simple telnet scenario

User

Host A

seq=42, ACK=79, data = 'C'

seq=79, ACK=43, data = 'C'

seq=43, ACK=80

User types 'C'

Simple telnet scenario
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 expected 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 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

EstimatedRTT = (1-x) * EstimatedRTT + x * SampleRTT

- Exponential weighted moving average
- influence of given sample decreases exponentially fast
- typical value of x: 0.125 (1/8)

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 Flow Control

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

TCP Round Trip Time and Timeout

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**

**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.)**

**Note:** with small modification, can handle simultaneous FINs.

**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.