Computer Networking

Connection-Oriented Transport: TCP

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**

RFCs: 793, 1122, 1323, 2018, 2581

- **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>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence number</td>
<td>acknowledgement number</td>
</tr>
<tr>
<td>sequence number</td>
<td>acknowledgement number</td>
</tr>
<tr>
<td>checksum</td>
<td>Urg data pointer</td>
</tr>
<tr>
<td>Receive window</td>
<td>Options (variable length)</td>
</tr>
<tr>
<td>application data</td>
<td>(variable length)</td>
</tr>
<tr>
<td>URG: urgent data (generally not used)</td>
<td>ACK: ACK # valid</td>
</tr>
<tr>
<td>PSH: push data now (generally not used)</td>
<td>RST, SYN, FIN: connection estab (setup, teardown commands)</td>
</tr>
<tr>
<td>Internet checksum (as in UDP)</td>
<td></td>
</tr>
</tbody>
</table>

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 implementer

User types ‘C’
Host A
Seq=42, ACK=79, data = ‘C’
host ACKs receipt of ‘C’, echoes back ‘C’
Seq=79, ACK=43, data = ‘C’
Seq=43, ACK=80

Host B
time

simple telnet scenario
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

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

- **Host A**
  - Seq=92, 8 bytes data
  - ACK=100
  - SendBase = 100
  - time
  - lost ACK scenario

- **Host B**
  - Seq=92, 8 bytes data
  - ACK=120

- **Host A**
  - Seq=100, 20 bytes data
  - SendBase = 120
  - time
  - premature timeout

- **Host B**
  - Seq=100, 20 bytes data
  - ACK=120
  - SendBase = 120
  - ACK=120
  - SendBase = 100
  - ACK=100
TCP retransmission scenarios (more)

Event at Receiver | TCP Receiver action
--- | ---
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed | Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending | Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. #. Gap detected | Immediately send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap | Immediate send ACK, provided that segment starts at lower end of gap

TCP ACK generation [RFC 1122, RFC 2581]
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 for that segment
- If sender receives 3 ACKs for same data, it assumes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

TCP Flow Control

- receive 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 send rate to receiving application’s drain rate
- app process may be slow at reading from buffer

TCP
TCP Flow control: how it works

- receiver: advertises unused buffer space by including \( rwnd \) value in segment header
- sender: limits # of unACKed bytes to \( rwnd \)
  - guarantees receiver’s buffer doesn’t overflow

(suppose TCP receiver discards out-of-order segments)
• unused buffer space:
  \( = rwnd \)
  \( = RcvBuffer - [LastByteRcvd - LastByteRead] \)

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
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

TCP

TIME_WAIT

FIN_WAIT_2

FIN_WAIT_1

SYN_SENT

ESTABLISHED

CLOSED

CLOSE_WAIT

LAST_ACK

SYN_RCVD

LISTEN

server application creates a listen socket

receive SYN send SYN & ACK

sending ACK send nothing

receive FIN send ACK

receive FIN send nothing

client application initiates close connection

send FIN

client application initiates a TCP connection

send SYN

receive FIN send ACK

receive FIN send nothing

send 20 seconds

receive FIN send ACK

receive SYN/ACK send ACK

receive ACK send nothing

TCP

TCP client lifecycle