Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

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>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number of the segment</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number of the segment</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number of the segment</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number of the segment</td>
</tr>
<tr>
<td>head len</td>
<td>Length of the header of the segment</td>
</tr>
<tr>
<td>URG</td>
<td>URG flag</td>
</tr>
<tr>
<td>PSH</td>
<td>PSH flag</td>
</tr>
<tr>
<td>SYN</td>
<td>SYN flag</td>
</tr>
<tr>
<td>RST</td>
<td>RST flag</td>
</tr>
<tr>
<td>FSF</td>
<td>FSF flag</td>
</tr>
<tr>
<td>Receive window</td>
<td>Receive window size</td>
</tr>
<tr>
<td>Urg data pnter</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>Options</td>
<td>Options field (variable length)</td>
</tr>
<tr>
<td>application data</td>
<td>Application data field (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)

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

Simple telnet scenario:
- User types ‘C’
- Host A: Seq=42, ACK=79, data = ‘C’
- Host B: Seq=79, ACK=43, data = ‘C’, echoes back ‘C’
- Host A: Seq=43, ACK=80

TCP seq. #’s and ACKs
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** = \((1- \alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT}\)

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: \(\alpha = 0.125\)
Example RTT estimation:

TCP Round Trip Time and Timeout

Setting the timeout

- EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT → larger safety margin

- first estimate of how much SampleRTT deviates from EstimatedRTT:
  
  \[
  \text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT}-\text{EstimatedRTT}|
  \]

  (typically, \( \beta = 0.25 \))

Then set timeout interval:

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]
Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

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 Sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
    switch(event)
    event: data received from application above
        create TCP segment with sequence number NextSeqNum
        if (timer currently not running)
            start timer
        pass segment to IP
        NextSeqNum = NextSeqNum + length(data)
    event: timer timeout
        retransmit not-yet-acknowledged segment with smallest sequence number
        start timer
    event: ACK received, with ACK field value of y
        if (y > SendBase) {
            SendBase = y
            if (there are currently not-yet-acknowledged segments)
                start timer
        }
} /* end of loop forever */
```
TCP: retransmission scenarios

Host A
Seq=100, 20 bytes data
ACK=100

Host B
Seq=92, 8 bytes data
ACK=120
Seq=92 timeout
ACK=120

Host A
Seq=92, 8 bytes data
ACK=100
loss
timeout

Host B
X
Seq=92, 8 bytes data
ACK=100
time
Seq=92 timeout

SendBase = 120

TCP retransmission scenarios (more)

Host A
Seq=92, 8 bytes data
ACK=100

Host B
X
Seq=100, 20 bytes data
ACK=100
SendBase = 120
ACK=120

Cumulative ACK scenario
TCP ACK generation [RFC 1122, RFC 2581]

<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 #</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>expected seq # already ACKed</td>
<td></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 duplicate ACK, 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
Fast retransmit algorithm:

- **event:** ACK received, with ACK field value of $y$
  - if ($y > \text{SendBase}$) {
    - $\text{SendBase} = y$
    - if (there are currently not-yet-acknowledged segments)
      - start timer
  }
  - else {
    - increment count of dup ACKs received for $y$
    - if (count of dup ACKs received for $y = 3$) {
      - resend segment with sequence number $y$
    }

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control
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 the send rate to the receiving app’s drain rate

- app process may be slow at reading from buffer

TCP Flow control: how it works

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

- spare room in buffer
  - spare room = $RcvWindow$
  - $RcvWindow = RcvBuffer - [LastByteRcvd - LastByteRead]$
  - $Rcvr$ advertises spare room by including value of $RcvWindow$ in segments
  - $Sender$ limits unACKed data to $RcvWindow$
    - guarantees receive buffer doesn’t overflow