Chapter 3: TCP

TCP: Overview

- Point-to-point:
  - one sender, one receiver
- Reliable, in-order byte steam:
  - no “message boundaries”
  - But TCP chops it up into segments for transmission internally
- Pipelined (window) flow control:
  - Window size decided by receiver and network
- Send & receive buffers

TCP: Overview

- 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 & Congestion Control:
  - sender will not overwhelm receiver or the network

TCP segment structure

- Sequence Numbers:
  - 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 seq. #’s and ACKs (I)
TCP Seq. #’s and ACKs (II)

User types ‘C’

status report receipt

simple telnet scenario

Temporal Redundancy Model

- Sequence Numbers
- CRC or Checksum
- ACKs
- NAKs,
- SACKs
- Bitmaps
- Packets
- FEC information

Status Report Design

- Cumulative acks:
  - Robust to losses on the reverse channel
  - Can work with go-back-N retransmission
  - Cannot pinpoint blocks of data which are lost
    - The first lost packet can be pinpointed because the receiver would generate duplicate acks

TCP: reliable data transfer (I)

- one way data transfer
- no flow, congestion control

TCP ACK generation

<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>Immediate send single cumulative ACK</td>
</tr>
<tr>
<td>In-order segment arrival, no gaps, one delayed ACK pending</td>
<td>Send duplicate ACK, indicating seq. # of next expected byte</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

- **Host A**: Seq = 92, 8 bytes data
- **Host B**: Seq = 92, 8 bytes data
- **TCP Flow Control**
  - **receiver**: explicitly informs sender of free buffer space
  - **sender**: keeps the amount of transmitted, unACKed data less than most recently received RcvWindow

Timeout and RTT Estimation

- **Timeout**: for robust detection of packet loss
- **Problem**: How long should timeout be?
  - Too long => underutilization
  - Too short => wasteful retransmissions
- **Solution**: adaptive timeout; based on estimate of max RTT

How to estimate max RTT?

- **RTT** = prop + queuing delay
  - Queuing delay highly variable
  - So, different samples of RTTs will give different random values of queuing delay
- **Chebyshev’s Theorem**:
  - \( \text{MaxRTT} = \text{Avg RTT} + k \times \text{Deviation} \)
  - Error probability is less than \( \frac{1}{(k^2)} \)
  - Result true for ANY distribution of samples

Round Trip Time and Timeout (II)

**Q**: how to estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions, cumulatively ACKed segments
  - SampleRTT will vary wildly => want estimated RTT “smoother”
  - use several recent measurements, not just current SampleRTT to calculate “AverageRTT”

TCP Round Trip Time and Timeout (III)

- **AverageRTT** = \( (1-x) \times \text{AverageRTT} + x \times \text{SampleRTT} \)
  - Exponential weighted moving average (EWMA)
  - Influence of given sample decreases exponentially fast; \( x = 0.1 \)

**Setting the timeout**

- **AverageRTT** plus “safety margin” proportional to variation
  
  \[
  \text{Timeout} = \text{AverageRTT} + 4 \times \text{Deviation} \\
  \text{Deviation} = (1-x)^7 \times \text{Deviation} + x^7 \times \text{SampleRTT} - \text{AverageRTT} \]

**Rensselaer Polytechnic Institute © Shivkumar Kalvanaram © Biplab Sikdar**
TCP Connection Management - 1
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();

TCP Connection Management - 2
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. #

TCP Connection Management - 3
Closing a connection:
client closes socket: 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 - 4

TCP client lifecycle

TCP Connection Management - 5

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 - 6
Recap: Stability of a Multiplexed System

- Average Input Rate > Average Output Rate
  => system is unstable!

How to ensure stability?
1. Reserve enough capacity so that demand is less than reserved capacity
2. Dynamically detect overload and adapt either the demand or capacity to resolve overload

Congestion Problem in Packet Switching

- **Problem:** demand outstrips available capacity

- If information about \( \lambda_i, \lambda, \) and \( \mu \) is known in a central location where control of \( \lambda_i \) or \( \mu \) can be effected with zero time delays,
  - the congestion problem is solved!

The Congestion Problem (Continued)

- **Problems:**
  - Incomplete information (e.g., loss indications)
  - Distributed solution required
  - Congestion and control/measurement locations different
  - Time-varying, heterogeneous time-delay

- **Static fixes may not solve congestion**
  - a) Memory becomes cheap (infinite memory)
    - No buffer
    - Too late
  - b) Links become cheap (high speed links)?
    - All links 19.2 kb/s
    - Replace with 1 Mb/s
    - File Transfer time = 5 mins
    - File Transfer Time = 7 hours
The Congestion Problem (Continued)

- c) **Processors** become cheap (fast routers & switches)

```
A   S   C
   B   D
```

**Scenario:** All links 1 Gb/s.
A & B send to C

⇒ “high-speed” congestion!!
(lose more packets faster!)

---

Principles of Congestion Control

**Congestion:**
- informally: “too many sources sending too much data too fast for network to handle”
- different from flow control (receiver overload)
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queuing in router buffers)
  - a top-10 problem!

---

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission

**Causes/costs of congestion:**
- large delays when congested
- maximum achievable throughput

---

Causes/costs of congestion: scenario 2

- one router, finite buffers
- sender retransmission of lost packet

**“Costs” of congestion:**
- More work (retrans) for given “goodput”
- Unneeded retransmissions: link carries multiple copies of pkt due to spurious timeouts

---

Causes/costs of congestion: scenario 2 (continued)

---

Causes/costs of congestion: scenario 3

**Another “cost” of congestion:**
- when packet dropped, any “upstream transmission capacity used for that packet was wasted!”
Two broad approaches towards congestion control:

**End-end congestion control:**
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

**Network-assisted congestion control:**
- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at

TCP congestion control - 1
- end-end control (no network assistance)
- transmission rate limited by congestion window size, Congwin, over segments:

TCP congestion control - 2
- w segments, each with MSS bytes sent in one RTT:

\[ \text{throughput} = \frac{w \times \text{MSS}}{\text{RTT}} \text{ Bytes/sec} \]

TCP congestion control - 3
- “Probing” for usable bandwidth:
  - Window flow control: avoid receiver overrun
  - Dynamic window congestion control: avoid/control network overrun
- Policy:
  - Increase Congwin until loss (congestion)
  - Loss \(\Rightarrow\) decrease Congwin, then begin probing (increasing) again

Additive Increase/Multiplicative Decrease (AIMD) Policy
- For stability:
  - rate-of-decrease > rate-of-increase
  - Decrease performed “enough” times as long as congestion exists
- AIMD policy satisfies this condition, provided packet loss is congestion indicator
Fairness

Fairness goal: if N TCP sessions share same bottleneck link, each should get 1/N of link capacity

TCP connection 1
TCP connection 2
bottleneck router capacity R

Fairness Analysis

AIMD Converges to Fairness

TCP congestion control - 4

- TCP uses AIMD policy in “steady state”
- Two “phases”
  - Transient phase: aka “Slow start”
  - Steady State: aka “Congestion avoidance”
- Important variables:
  - Congwin
  - threshold: defines threshold between two slow start phase, congestion avoidance phase

TCP Slowstart - 1

Slowstart algorithm

initialize: Congwin = 1
for (each segment ACKed)
  Congwin++
  until (loss event OR CongWin > threshold)

- Exponential increase (per RTT) in window size (not so slow!)
- Loss event: timeout (Tahoe TCP) and/or three duplicate ACKs (Reno TCP)

TCP Slowstart - 2

- asdf Host A
  - Host B

- User 1’s Allocation x1
  - User 2’s Allocation x2
  - Optimal point
  - EQM
  - Underload
  - Overhead
  - Fairness Line
  - Efficiency Line
  - Fairness Goal
  - Congestion Line
  - Fairness Line

Fig. 1. AIMD increases bandwidth/throughput. Efficiency converges to the optimal point.

Rensselaer Polytechnic Institute © Shivkumar Kalvanaraman & © Biplab Sikdar
TCP Dynamics

• Rate of acks determines rate of packets: "Self-clocking" property.

TCP Congestion Avoidance - 1

/* slowstart is over */
/* Congwin > threshold */
Until (loss event) {
every w segments ACKed:
    Congwin++
} 
threshold = Congwin/2 
Congwin = 1 
perform slowstart

1: TCP Reno skips slowstart (aka fast recovery) after three duplicate ACKs and performs close to AIMD

TCP Congestion Avoidance - 2

TCP window dynamics (more)

TCP latency modeling - 1

Q: How long does it take to receive an object from a Web server after sending a request?
• TCP connection establishment
• data transfer delay

TCP latency modeling - 2

Notation, assumptions:
• Assume one link between client and server of rate R
• Assume fixed congestion window, W segments
• S: MSS (bits)
• O: object size (bits)
• no retransmissions (no loss, no corruption)
TCP latency modeling - 3

Two cases to consider:
- $WS/R > RTT + S/R$: ACK for first segment in window returns before window’s worth of data sent
- $WS/R < RTT + S/R$: wait for ACK after sending window’s worth of data sent

TCP latency modeling - 4

Case 1: latency = $2RTT + O/R$

TCP latency modeling - 5

Case 2: latency = $2RTT + O/R + (K-1)[S/R + RTT - WS/R]$

TCP latency modeling:
slow start - 1

- Now suppose window grows according to slow start.
- Will show that the latency of one object of size $O$ is:

$$\text{where } P \text{ is the number of times TCP stalls at server:}$$

TCP latency modeling:
slow start - 2

- where $Q$ is the number of times the server would stall if the object were of infinite size.
- and $K$ is the number of windows that cover the object.

TCP latency modeling:
slow start - 3

Example:
$O/S = 15$ segments
$K = 4$ windows
$Q = 2$
$P = \min(K-1,Q) = 2$
Server stalls $P = 2$ times.
TCP latency modeling:
slow start - 4

TCP latency modeling:
slow start - 5

Sample Results

<table>
<thead>
<tr>
<th>R</th>
<th>O/R</th>
<th>F</th>
<th>Minimum Latency: O/R + 2 RTT</th>
<th>Latency with slow start</th>
</tr>
</thead>
<tbody>
<tr>
<td>28 Kbps</td>
<td>28.6 sec</td>
<td>1</td>
<td>28.8 sec</td>
<td>28.9 sec</td>
</tr>
<tr>
<td>100 Kbps</td>
<td>8 sec</td>
<td>2</td>
<td>8.2 sec</td>
<td>8.4 sec</td>
</tr>
<tr>
<td>1 Mbps</td>
<td>800 msec</td>
<td>5</td>
<td>1 sec</td>
<td>1.5 sec</td>
</tr>
<tr>
<td>10 Mbps</td>
<td>90 msec</td>
<td>7</td>
<td>0.28 sec</td>
<td>0.98 sec</td>
</tr>
</tbody>
</table>

Summary: Chapter 3

- Principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- Instantiation and implementation in the Internet
  - UDP, TCP