TCP congestion control
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 congestion control

- Introduced by Van Jacobson in the late 80's
- Done without changing headers or routers
- Senders try and determine capacity of network
- Implicit congestion signal: packet loss
- ACK from previous packet determines when to send more data, "self-clocking"
TCP congestion control

• Each TCP sender tracks:
  – rwnd = Advertised window, for flow control
  – cwnd = Congestion window, for congestion control

• Sender uses minimum of the two:
  – rwnd prevents overrunning receiver's buffer
  – cwnd prevents overloading network

• Situation is dynamic:
  – Network changes
    • e.g. new high bandwidth link, hosts start/stop sending
  – Sender always searching for best sending rate
Basic TCP congestion control

- Add one packet to window per RTT
  - Works well if we start near capacity
  - Otherwise could take a long time to discover real network capacity
Slow start

- Slow start
  - Increase congestion window rapidly from cold start of 1
  - Add 1 to window for every good ACK
    - Exponential increase in packets in flight
  - On packet loss, start over at 1
  - Slow in comparison to original TCP
    - Immediate sending up to advertised window (caused congestion collapse)

http://history.visualland.net/tcp_swnd.html

http://tcp.cs.st-andrews.ac.uk/index.shtml?page=slow_start
Congestion avoidance, ssthresh

- Congestion avoidance
  - Initially set slow start threshold to large value
  - On multiplicative decrease, \( ssthresh = \frac{cwnd}{2} \)
  - When ramping back up, switch to additive increase upon reaching ssthresh
Fast retransmission

- Problem: Timeouts take a long time

- Fast retransmission
  - Retransmit on suspected loss
  - Triggered after 3rd duplicate ACK
  - 20% increase in throughput

- TCP "Tahoe"
  - Slow start + congestion avoidance + fast retransmission
  - Reset cwnd to 1 on timeout/3rd duplicate ACK
Fast recovery

• Problem: Restarting from 1 takes too long
  – We spend too long below "known" network limit

• Fast recovery
  – ACK clock still working even though packet was lost
  – Count up dup ACKs (including 3 that triggered fast retransmission)
  – Once packets in-flight has reached new threshold, start sending packet on each dup ACK
  – Once lost packet ACK'd, exit fast recovery and start linear increase
Fast recovery

- TCP "Reno"
  - Tahoe + fast recovery

http://www.brunocosari.net/projects_content/2?width=1000&height=500&iframe=true
Summary: TCP congestion control

- **slow start**
  - \( cwnd = 1 \text{ MSS} \)
  - \( ssthresh = 64 \text{ KB} \)
  - \( \text{dupACKcount} = 0 \)

- **congestion avoidance**
  - \( cwnd = cwnd + \text{MSS} \)
  - \( \text{dupACKcount} = 0 \)

- **fast recovery**
  - \( cwnd = ssthresh + 3 \)
  - \( \text{dupACKcount} = 0 \)

- **duplicate ACK**
  - \( \text{dupACKcount}++ \)

- **timeout**
  - \( \text{dupACKcount} = 0 \)

- **new ACK**
  - \( cwnd = \frac{cwnd}{2} \)
  - \( \text{dupACKcount} = 0 \)
  - \( \text{transmit new segment(s), as allowed} \)

- **ssthresh**
  - \( ssthresh = cwnd/2 \)
  - \( cwnd = 1 \text{ MSS} \)
  - \( \text{dupACKcount} = 0 \)

- **retransmit missing segment**
  - \( cwnd = \frac{cwnd}{2} \)
  - \( \text{dupACKcount} = 0 \)

- **fast recovery**
  - \( cwnd = ssthresh + 3 \)
  - \( \text{dupACKcount} = 0 \)

- **duplicate ACK**
  - \( \text{dupACKcount}++ \)

- **new ACK**
  - \( cwnd = cwnd + \text{MSS} \cdot (\text{MSS}/cwnd) \)
  - \( \text{dupACKcount} = 0 \)
  - \( \text{transmit new segment(s), as allowed} \)

- **ssthresh**
  - \( ssthresh = cwnd/2 \)
  - \( cwnd = \frac{cwnd}{2} \)
  - \( \text{dupACKcount} = 0 \)

- **retransmit missing segment**
  - \( cwnd = ssthresh + 3 \)
  - \( \text{dupACKcount} = 0 \)

- **fast recovery**
  - \( cwnd = cwnd + \text{MSS} \)
  - \( \text{transmit new segment(s), as allowed} \)

- **new ACK**
  - \( cwnd = cwnd + \text{MSS} \cdot (\text{MSS}/cwnd) \)
  - \( \text{dupACKcount} = 0 \)
  - \( \text{transmit new segment(s), as allowed} \)

- **ssthresh**
  - \( ssthresh = cwnd/2 \)
  - \( cwnd = \frac{cwnd}{2} \)
  - \( \text{dupACKcount} = 0 \)

- **retransmit missing segment**
  - \( cwnd = ssthresh + 3 \)
  - \( \text{dupACKcount} = 0 \)
### Some of TCP's flavors

<table>
<thead>
<tr>
<th>Name</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tahoe</td>
<td>Slow start, congestion avoidance, fast retransmit.</td>
</tr>
<tr>
<td>Reno</td>
<td>Tahoe's features + fast recovery.</td>
</tr>
<tr>
<td>New Reno</td>
<td>Improves Reno to handle multiple packet loss within window. Changes to fast recovery, allows filling of multiple holes in sequence space.</td>
</tr>
<tr>
<td>Vegas</td>
<td>Monitor for signs of increasing congestion using RTT. Supports linear increase and <em>decrease</em> of congestion window.</td>
</tr>
<tr>
<td>BIC</td>
<td>Binary Increase Congestion control, optimized for high speed, long latency networks (long fat networks). Default in Linux 2.6.8-2.6.18.</td>
</tr>
<tr>
<td>CUBIC</td>
<td>Less aggressive that BIC, based on a cubic growth function. Default in Linux 2.6.19+</td>
</tr>
<tr>
<td>Compound</td>
<td>Microsoft, optimized for long fat networks while trying to remain fair. Default in XP and Vista, available in Windows 7.</td>
</tr>
</tbody>
</table>

...
TCP throughput

- Avg. TCP throughput as function of window size, RTT?
  - Ignore slow start, assume always data to send

- **W**: window size (measured in bytes)
  - Avg. window size (# in-flight bytes) is \( \frac{3}{4} W \)
  - Avg. throughput is \( \frac{3}{4}W \) per RTT

\[
\text{Avg. TCP throughput} = \frac{3}{4} \frac{W}{\text{RTT}} \text{ bytes/sec}
\]
TCP over long, fat pipes

• **Example:**
  – 1500 byte segments, 100ms RTT
  – Want 10 Gbps throughput
  – Requires $W = 83,333$ in-flight segments

• **Throughput in terms of segment loss probability, $L$**
  [Mathis 1997]:

  $$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}$$

  ➔ To achieve 10 Gbps throughput, need a loss rate of $L = 2 \times 10^{-10}$ — *a very small loss rate!*

• **New versions of TCP for high-speed environments**
TCP fairness

**Fairness goal:**

If K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K
Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- Multiplicative decrease decreases throughput proportionally

Why is TCP fair?
Cheating

• Not everybody plays fair:
  – Run multiple TCP connections in parallel
  – Change the TCP implementation
    • Starts your TCP connection off with > 1 MSS
  – Use a protocol without congestion control
    • e.g. UDP
  – Good guys slow down to make way so others can have unfair share of bandwidth

• Possible solutions?
  – Routers detect cheating and drop excess traffic
  – Fair queuing
Network flows

• **Connection flows**
  – IP network is connectionless
  – Datagrams really not independent
  – Stream of datagrams between two hosts
  – Routers can infer current flows, "soft state"
Fair queuing

- Use flows to determine scheduling
  - Prevent hosts from hogging all the router resources
  - Important if hosts don't implement host-based congestion control (e.g. TCP congestion control)
  - Each flow gets its own queue, served round-robin
Wireless networks

• **TCP congestion control uses packet loss as signal**
  – Wireless/satellite links = high error rate
  – TCP may mistake bit errors as congestion

• **Possible solutions:**
  – Link layer acknowledgements and retransmission
  – Forward error correction
  – Split connection into wireless/wired segments
  – Use other signals than packet loss: increasing RTT
TCP splitting

• **Optimize cloud-based services**
  – e.g. Web search, e-mail, social networks
  – Give illusion of operating locally (i.e. low latency)
  – But: data center may be a long way and speed of light is a constant + new connection subject to TCP slow-start

• **TCP splitting**
  – Deploy front-end servers near to users
    • e.g. Google's "enter-deep" clusters at access ISPs
  – Client make TCP connection to front-end server, small RTT
  – Front-end maintains persistent connection to back-end with large congestion window
**Fig. 2.** CDF of response time of 200K search queries by popular search engine for search reply

**Fig. 3.** TCP packet exchange diagram between an HTTP client and a search server with a proxy between them.

**Fig. 6.** Gain of TCP Splitting

Chapter 3 summary

- Principles behind transport layer services:
  - Multiplexing, demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control

- Instantiation in the Internet
  - UDP
  - TCP

Next:
- Leaving the network edge (application, transport layers)
- Into the network core!