Goals of Today’s Lecture

• Finish up computing Round-Trip Time (RTT) & Retransmission Timeout (RTO)

• Principles of congestion control
  – Learning that congestion is occurring
  – Adapting to alleviate the congestion

• TCP congestion control
  – Additive-increase, multiplicative-decrease (AIMD)
  – NACK- (“fast retransmission”) and timeout-based detection
  – How to begin transmitting: Slow Start

Problem: Ambiguous RTT Measurement

• How to differentiate between the real ACK, and ACK of the retransmitted packet?

Karn/Partridge Algorithm

• Measure SampleRTT only for original transmissions
  – Once a segment has been retransmitted, do not use it for any further measurements

• Also, employ exponential backoff
  – Every time RTO timer expires, set RTO ← 2·RTO
  – (Up to maximum ≥ 60 sec)
  – Every time new measurement comes in (= successful original transmission), collapse RTO back to computed value

How Fast Should TCP Send?

Flow Control
**Sliding Window**

- Allow a larger amount of data “in flight”
  - Allow sender to get ahead of the receiver
  - … though not too far ahead

---

**TCP Header for Receiver Buffering**

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td>Acknowledgment</td>
</tr>
<tr>
<td>Advertised window</td>
<td>Checksum</td>
</tr>
<tr>
<td>Options (variable)</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

---

**Advertised Window Limits Rate**

- If the window is \( W \), then sender can send no faster than \( W/RTT \) bytes/sec
  - Receiver implicitly limits sender to rate that receiver can sustain
  - If sender is going too fast, window advertisements get smaller & smaller
  - Termed Flow Control
- In original TCP design, that was it - sole protocol mechanism controlling sender’s rate

- What’s missing?

---

**How Fast Should TCP Send?**

Congestion Control

---

**It’s Not Just The Sender & Receiver**

- Flow control keeps one fast sender from overwhelming a slow receiver
- Congestion control keeps a set of senders from overloading the network

- Three congestion control problems:
  - Adjusting to bottleneck bandwidth
    - Without any a priori knowledge
    - Could be a Gbps link; could be a modem
  - Adjusting to variations in bandwidth
  - Sharing bandwidth between flows

---

**Congestion is Unavoidable**

- Two packets arrive at the same time
  - The node can only transmit one
  - … and either buffers or drops the other
- If many packets arrive in a short period of time
  - The node cannot keep up with the arriving traffic
  - … and the buffer may eventually overflow
Load, Delay, and Power

Typical behavior of queuing systems with bursty arrivals:

A simple metric of how well the network is performing:

\[ \text{Power} = \frac{\text{Load}}{\text{Delay}} \]

Load

Delay

“optimal load”

Goal: maximize power

Congestion Collapse

- Definition: Increase in network load results in a decrease of useful work done
- Due to:
  - Undelivered packets
  - Spurious retransmissions of packets still in flight
    - Unnecessary retransmissions lead to more load!
    - Pouring gasoline on a fire
- Mid-1980s: Internet grinds to a halt
  - Until Jacobson/Karels (Berkeley!) devise TCP congestion control

View from a Single Flow

- **Knee** — point after which
  - Throughput increases very slowly
  - Delay increases quickly

- **Cliff** — point after which
  - Throughput starts to decrease very fast to zero (congestion collapse)
  - Delay approaches infinity

General Approaches

- Send without care
  - Many packet drops
  - Disaster: leads to congestion collapse
- (1) Get network snapshot, compute global optimum
  - Difficult to scale or to construct consistent snapshot
- (2) Reservations
  - Pre-arrange bandwidth allocations
  - Requires negotiation before sending packets
  - Potentially low utilization (difficult to stat-mux)
- (3) Pricing
  - Don’t drop packets for the highest bidders
  - Requires payment model

General Approaches (cont’d)

- (4) Dynamic Adjustment by end systems
  - Probe network to test level of congestion
  - Speed up when no congestion
  - Slow down when congestion
  - Drawbacks:
    - Suboptimal
    - Messy dynamics
    - Seems complicated to implement
    - But clever algorithms actually pretty simple (Jacobson/Karels ’88)
- All four techniques have their place
  - But for generic Internet usage, dynamic adjustment is the most appropriate …
  - … due to pricing structure, traffic characteristics, and good citizenship

5 Minute Break

Questions Before We Proceed?
Idea of TCP Congestion Control

- Each source determines the available capacity
  - ...so it knows how many packets to have in flight
- Congestion window (CWND)
  - Maximum # of unacknowledged bytes to have in flight
  - Congestion-control equivalent of receiver window
  - MaxWindow = \( \min(\text{congestion window, receiver window}) \)
  - Send at the rate of the slowest component
- Adapting the congestion window
  - Decrease upon detecting congestion
  - Increase upon lack of congestion: optimistic exploration
- Note: TCP congestion control done only by end systems, not by mechanisms inside the network

Detecting Congestion

- How can a TCP sender determine that network is under stress?
  - Network could tell it (ICMP Source Quench)
    - Risky, because during times of overload the signal itself could be dropped!
  - Packet delays go up (knee of load-delay curve)
    - Tricky, because a noisy signal (delay often varies considerably)
  - Packet loss
    - Fail-safe signal that TCP already has to detect
      - Complication: non-congestive loss (checksum errors)

Additive Increase, Multiplicative Decrease

- How much to increase and decrease?
  - Increase linearly, decrease multiplicatively (AIMD)
  - Necessary condition for stability of TCP
  - Consequences of over-sized window much worse than having an under-sized window
    - Over-sized window: packets dropped and retransmitted
    - Under-sized window: somewhat lower throughput
- Additive increase
  - On success for last window of data, increase linearly
    - TCP uses an increase of one packet (MSS) per RTT
- Multiplicative decrease
  - On loss of packet, TCP divides congestion window in half

Leads to the TCP “Sawtooth”

Managing the Congestion Window

- Increasing CWND
  - Increase by MSS on success (= no loss) for last window of data
    - One approach: track first packet in flight, new window starts when it’s ack’d, at which point: \( \text{CWND} += \text{MSS} \)
    - Another: increase a fraction of MSS per received ACK
      - If packets (and thus ACKs) per window: \( \text{CWND} / \text{MSS} \)
      - Increment per ACK: \( \text{CWND} += \text{MSS} \times (\text{MSS} / \text{CWND}) \)
      - Is actually slightly sub-linear
- Decreasing the congestion window
  - Cut in half on loss detected by NACK (“fast retransmit”)
  - Cut all the way to 1 MSS on loss detected by timeout
  - Never drop CWND below 1 MSS

Getting Started

Need to start with a small CWND to avoid overloading the network.

But, could take a long time to get started!
“Slow Start” Phase

- Start with a small congestion window
  - Initially, CWND is 1 MSS (*)
  - So, initial sending rate is MSS/RTT
- That could be pretty wasteful
  - Might be much less than the actual bandwidth
  - Linear increase takes a long time to accelerate
- Slow-start phase (actually “fast start”)
  - Sender starts at a slow rate (hence the name)
  - ... but increases the rate exponentially
  - ... until the first loss event

Slow Start in Action

- Double CWND per round-trip time
- Simple implementation:
  - on each ack, CWND += MSS

Loss Detection in TCP, Scheme #1

- Quick NACK-based detection
- Triple duplicate ACK (“three dups”)
  - Packet n is lost, but packets n+1, n+2, ..., arrive
  - On each arrival of a packet not in sequence, receiver generates an ACK
    - As always, ACK is for seq.no. just beyond highest in-sequence
    - So as n+1, n+2, ... arrive, receiver generates repeated ACKs for seq.no. n
    - “duplicate” acknowledgments since they all look the same
  - Sender sees 3 of these and immediately retransmits packet n (and only n)
  - Multiplicative decrease and keep going
- Termed Fast Retransmission

Loss Detection in TCP, Scheme #2

- Timeout
- Sender starts a timer that runs for RTO seconds
- Every time ack for new data arrives, restart timer
- If timer expires:
  - Set SSTHRESH ← CWND / 2 (“Slow-Start Threshold”)
  - Set CWND ← MSS (avoid a burst)
  - Retransmit first lost packet
  - Execute Slow Start until CWND > SSTHRESH
  - After which switch to Additive Increase
    - Termed: Congestion Avoidance

Fast Retransmission

- Resend a segment after 3 duplicate ACKs
  - Duplicate ACK means that an out-of-sequence segment was received
- Notes:
  - ACKs are for next expected packet
  - Packet reordering can cause duplicate ACKs
  - Window may be too small to generate enough duplicate ACKs
Repeating Slow Start After Timeout

Slow-start restart: Go back to CWND of 1 MSS, but take advantage of knowing the previous value of CWND.

Summary

- Congestion is inevitable
  - Internet does not reserve resources in advance
  - TCP actively tries to grab capacity
- Congestion control critical for avoiding collapse
  - **AIMD**: Additive Increase, Multiplicative Decrease
  - Congestion detected via packet loss (fail-safe)
    - NACK-based fast retransmission on “three dups”
    - Timeout
- Slow start to find initial sending rate & to restart after timeout
- Next class
  - TCP performance