TCP Congestion Control

EE 122: Intro to Communication Networks

Fall 2006 (MW 4-5:30 in Donner 155)

Vern Paxson

TAs: Dilip Antony Joseph and Sukun Kim

http://inst.eecs.berkeley.edu/~ee122/

Materials with thanks to Jennifer Rexford, Ion Stoica, and colleagues at Princeton and UC Berkeley

Announcements

- Project #3 should be out tonight
  - Can do individual or in a team of 2 people
  - First phase due November 16 - **no slip days**
  - Exercise good (better) time management

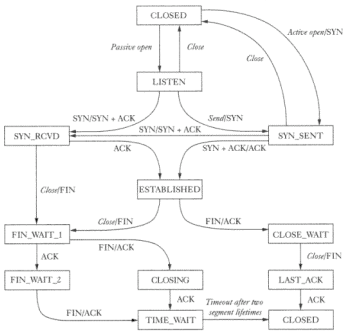
Goals of Today's Lecture

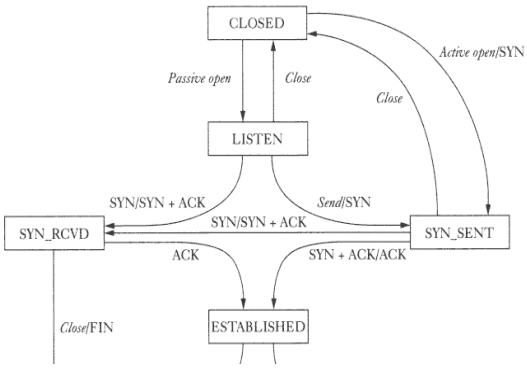
- State diagrams
  - Tool for understanding complex protocols
- Principles of congestion control
  - Learning that congestion is occurring
  - Adapting to alleviate the congestion
- TCP congestion control
  - Additive-increase, multiplicative-decrease
  - NACK- ("fast retransmission") and timeout-based detection
  - Slow start and slow-start restart

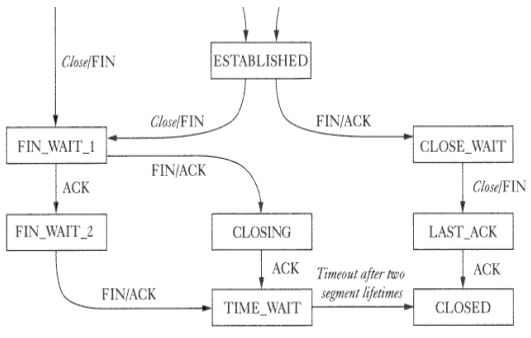
State Diagrams

- For complicated protocols, operation depends critically on current mode of operation
- Important tool for capture this: state diagram
- At any given time, protocol endpoint is in a particular state
  - Dictates its current behavior
- Endpoint transitions to other states on events
  - Interaction with lower layer
    - Reception of certain types of packets
  - Interaction with upper layer
    - New data arrives to send, or received data is consumed
  - Timers

TCP State Diagram







7

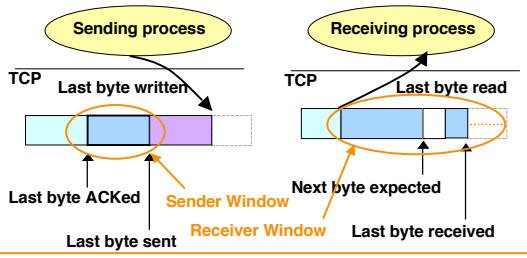
How Fast Should TCP Send?

Flow Control

8

Sliding Window

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



9

TCP Header for Receiver Buffering

Advertised window informs sender of receiver's buffer space

Source port			Destination port
Sequence number			
Acknowledgment			
HdrLen	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			
Data			

10

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 to 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?

11

How Fast Should TCP Send?

Congestion Control

12

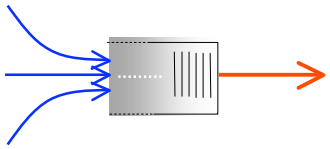
It's Not Just The Sender & Receiver

- 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

13

Congestion is Unavoidable

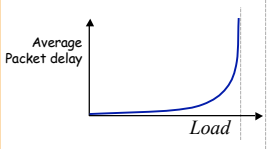
- Two packets arrive at the same time
  - The node can only transmit one
  - … and either buffers or drops the other
- If many packets arrive in a short period of time
  - The node cannot keep up with the arriving traffic
  - … and the buffer may eventually overflow



14

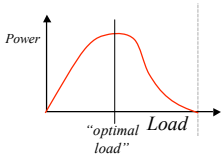
Load, Delay, and Power

Typical behavior of queuing systems with bursty arrivals:



A simple metric of how well the network is performing:

$$Power = \frac{Load}{Delay}$$



Goal: maximize power

15

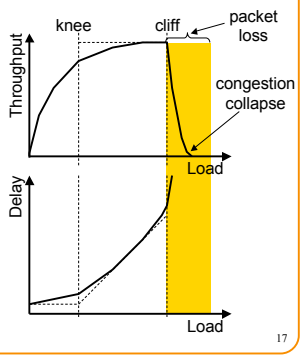
Congestion Collapse

- Definition: Increase in network load results in a **decrease** of useful work done
- Due to:
  - Undelivered packets
    - Packets consume resources and are dropped later in network
  - 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 devise TCP congestion control

16

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



17

General Approaches

- Send without care
  - Many packet drops
  - **Disaster**: leads to **congestion collapse**
- (1) Reservations
  - Pre-arrange bandwidth allocations
  - Requires negotiation before sending packets
  - Potentially low utilization (difficult to stat-mux)
- (2) Pricing
  - Don't drop packets for the highest bidders
  - Requires payment model

18

General Approaches (cont'd)

- (3) Dynamic Adjustment
  - 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 three techniques have their place
  - But for generic Internet usage, dynamic adjustment is the most appropriate …
  - … due to pricing structure, traffic characteristics, and good citizenship

19

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{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

20

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)

21

5 Minute Break

Questions Before We Proceed?

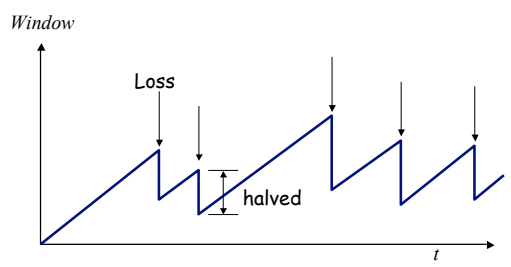
22

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
    - **One packet (MSS) per RTT**
- Multiplicative decrease
  - On loss of packet, divide congestion window in **half**

23

Leads to the TCP "Sawtooth"



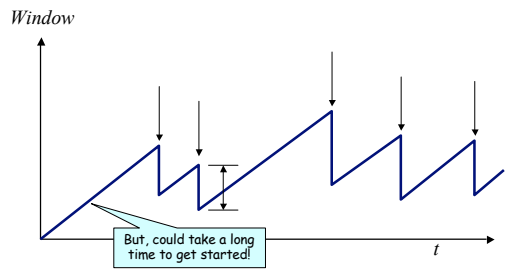
24

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: CWND += MSS
  - Another: increase a fraction of MSS per received ACK
    - # packets (and thus ACKs) per window: CWND / MSS
    - Increment per ACK: CWND += MSS * (MSS / CWND)
    - Is actually slightly super-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.



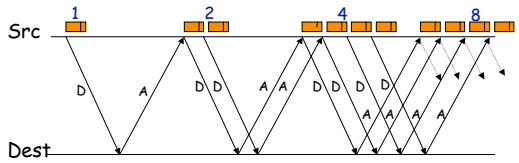
"Slow Start" Phase

- 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

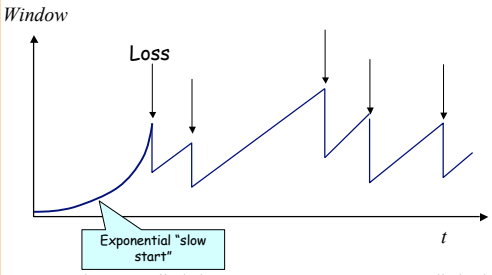
Slow Start in Action

Double CWND per round-trip time

Simple implementation:
on each ack, CWND += MSS



Slow Start and the TCP Sawtooth



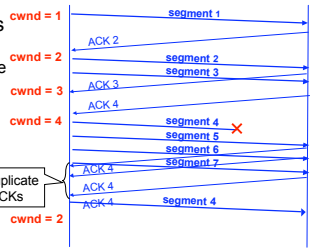
Why is it called slow-start? Because TCP originally had no congestion control mechanism. The source would just start by sending a whole window's worth of data.

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 three of these and immediately retransmits packet **n** (and only **n**)
  - Multiplicative decrease and keep going
- Termed **Fast Retransmission**

Fast Retransmission

- Resend a segment after 3 duplicate ACKs
  - Duplicate ACK means that an out-of sequence segment was received

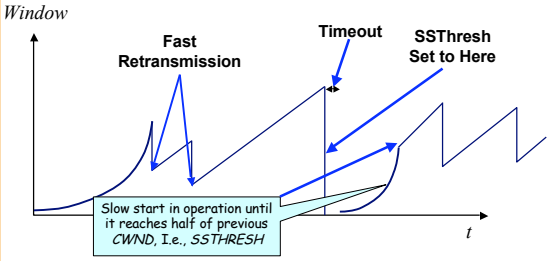


- Notes:
  - ACKs are for next expected packet
  - Packet reordering can cause duplicate ACKs
  - Window may be too small to generate enough duplicate ACKs

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 ("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

Repeating Slow Start After Timeout



Slow-start restart: Go back to CWND of 1, 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