CS 640 Introduction to Computer Networks

Lecture16

Today's lecture

- TCP congestion control
  - Overview of RENO TCP
  - Reacting to Congestion
  - SS/AIMD example

TCP Congestion Control

- Idea
  - assumes best-effort network (FIFO or FQ routers) each source determines network capacity for itself
  - uses implicit feedback
  - ACKs pace transmission (*self-clocking*)
- Challenge
  - determining the available capacity in the first place
  - adjusting to changes in the available capacity

TCP RENO Overview

- Standard TCP functions
  - Listed in last lecture: connections, reliability, etc.
- Jacobson/Karles RTT/RTO calculation
- Slow Start
- Congestion control/management
  - Additive Increase/ Multiplicative Decrease (AIMD)
  - Fast Retransmit/Fast Recovery

Additive Increase/Multiplicative Decrease

- Objective: adjust to changes in the available capacity
- New state variable per connection: **`CongestionWindow`**
  - limits how much data source has in transit

```
MaxWin = MIN(CongestionWindow,
             AdvertisedWindow)
EffWin = MaxWin - (LastByteSent -
                   LastByteAcked)
```

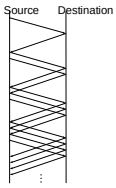
- Idea:
  - increase **`CongestionWindow`** when congestion goes down
  - decrease **`CongestionWindow`** when congestion goes up

AIMD (cont)

- Question: how does the source determine whether or not the network is congested?
- Answer: a timeout occurs
  - timeout signals that a packet was lost
  - packets are seldom lost due to transmission error
  - lost packet implies congestion
  - RTO calculation is critical

AIMD (cont)

- Algorithm
  - increment **`CongestionWindow`** by one packet per RTT (*linear increase*)
  - divide **`CongestionWindow`** by two on timeouts (*multiplicative decrease – fast!!*)
  - **`CongestionWindow`** always >= 1 MSS



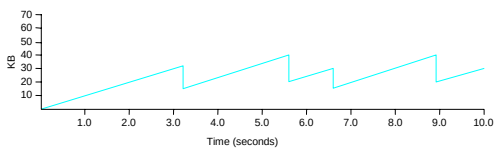
- In practice: increment a little for each ACK

```
Increment = 1/CongestionWindow
CongestionWindow += Increment
```

MSS = max segment size = size of a single packet

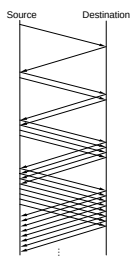
AIMD (cont)

- Trace: sawtooth behavior



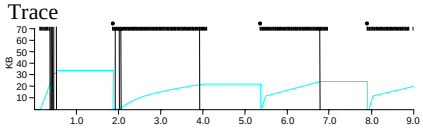
Slow Start

- Objective: determine the available capacity in the first
  - Additive increase is too slow
    - One additional packet per RTT
- Idea:
  - begin with **`CongestionWindow`** = 1 pkt
  - double **`CongestionWindow`** each RTT (increment by 1 packet for each ACK)
  - This is exponential increase to probe for available bandwidth
- SSTHRESH indicates when to begin additive increase



Slow Start contd.

- Exponential growth, but slower than all at once
- Used…
  - when first starting connection
  - when connection goes dead waiting for timeout
- Trace

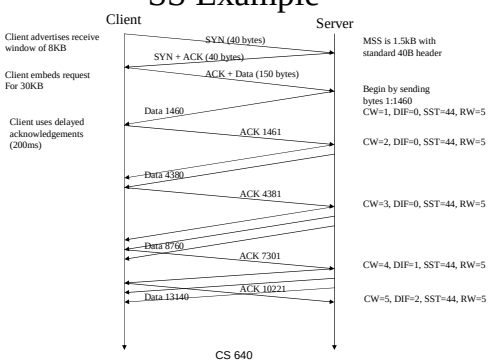


- Problem: lose up to half a **`CongestionWindow`**'s worth of data

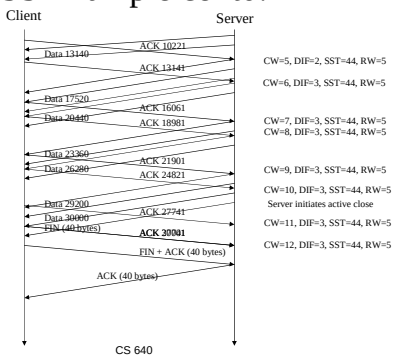
SSTHRESH and CWND

- SSTHRESH called `CongestionThreshold` in book
- Typically set to very large value on connection setup
- Set to one half of `CongestionWindow` on packet loss
  - So, SSTHRESH goes through multiplicative decrease for each packet loss
  - If loss is indicated by timeout, set `CongestionWindow = 1`
    - SSTHRESH and `CongestionWindow` always >= 1 MSS
- After loss, when new data is ACKed, increase CWND
  - Manner depends on whether we're in slow start or congestion avoidance

SS Example

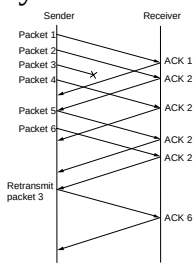


SS Example contd.

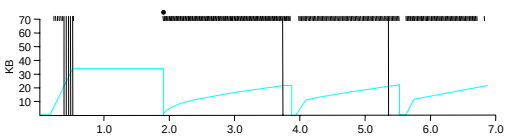


Fast Retransmit and Fast Recovery

- Problem: coarse-grain TCP timeouts lead to idle periods
- Fast retransmit: use 3 duplicate ACKs to trigger retransmission
- Fast recovery: start at SSTHRESH and do additive increase after fast retransmit



Fast Retransmit Results



- This is a graph of fast retransmit only
  - Avoids some of the timeout losses
- Fast recovery
  - skip the slow start phase in this graph at 3.8 and 5.5 sec
  - go directly to half the last successful **`CongestionWindow (ssthresh)`**