The Transport Layer: Congestion Control

CS 352, Lecture 10, Spring 2020

http://www.cs.rutgers.edu/~sn624/352

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Course announcements

- Project 2 will go online very soon
  - Due April 3rd at 10 PM
- Mid-term grading almost finished
- Project 1 should be returned sometime next week

Review of concepts

- Permissible window size varies over time.
  - Receiver application's reading speed (`recv`)
  - Communicated by the receiver to the sender through a header field
- How should the receiver socket buffer be sized?
  - What if it's too large?
  - What if it's too small?
  - Want to accommodate bursty transmissions from the sender
- Bandwidth-delay product

Feedback and Actions

The signals and knobs of congestion control

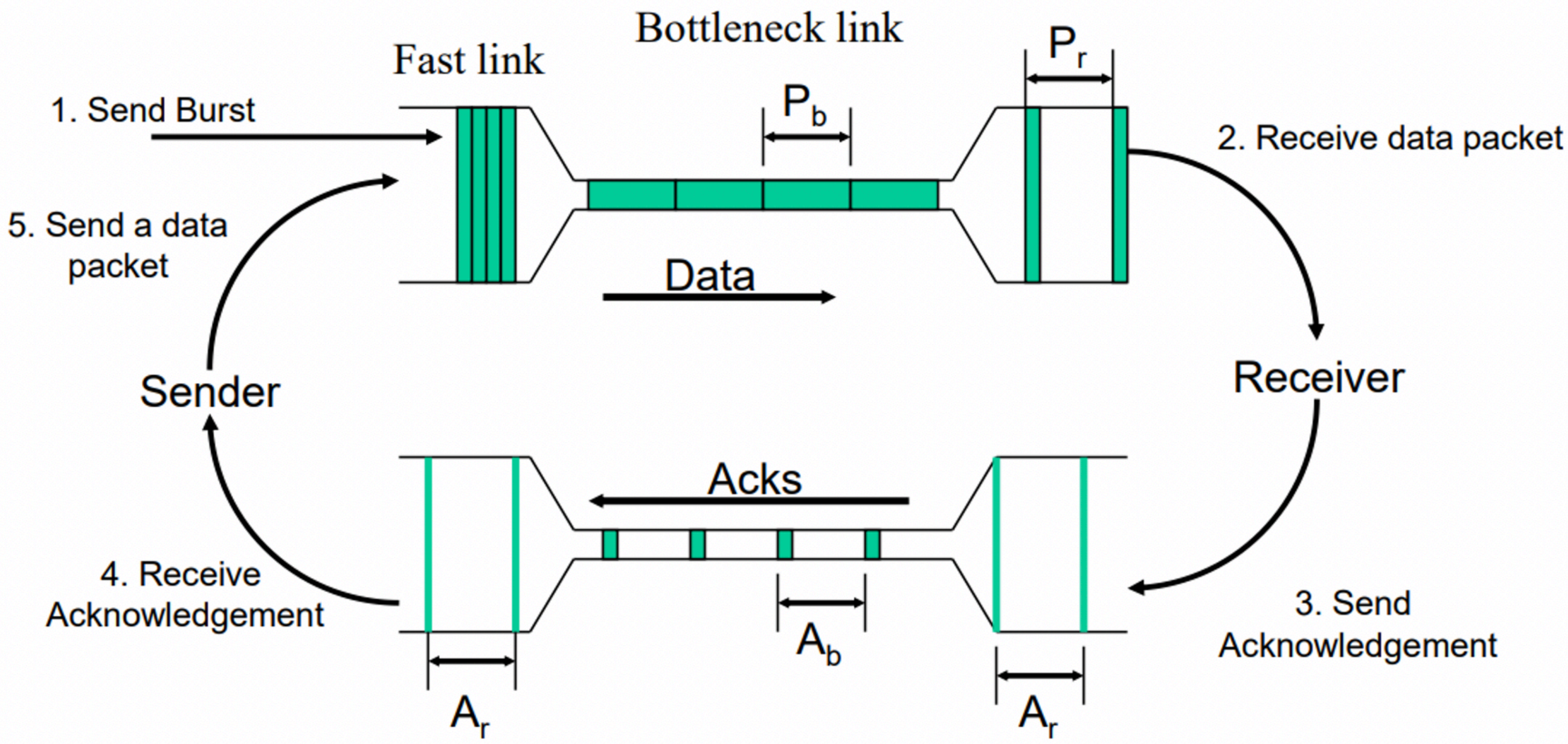
Feedback from network offers clues…

- Signals
  - Packets being dropped (ex, RTO fires)
  - Packets being delayed
  - Rate of incoming ACKs

"Implicit" feedback signals (more on explicit signals later)

- Knobs
  - What can you change to "probe" the sending rate?
  - Suppose receiver buffer is unbounded:
  - Let's call the amount of in-flight data per RTT the congestion window
  - Increase congestion window: e.g., by x or by a factor of x
  - Decrease congestion window: e.g., by x or by a factor of x

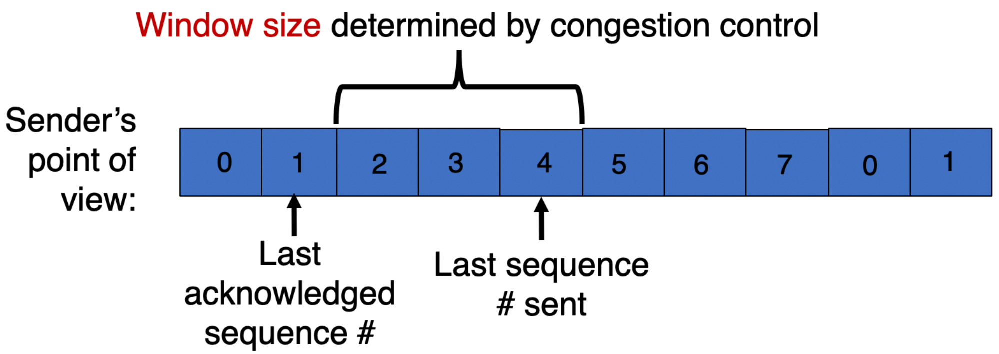
Steady state: Self clocking/ACK clocking



Given: $P_b = P_r = A_r = A_b = A_r$ (in units of time)
Sending a packet on each ACK keeps the bottleneck link busy

TCP congestion window

- Congestion window (cwnd): an estimate of in-flight data needed to keep the pipe full and achieve self-clocking
- Sending window = min(congestion window, receiver advertised window). Why min?
  - Overwhelm neither the receiver nor network routers
- Use sliding window concept
  - Window size is adjusted by congestion



Finding the right window

Time for an activity

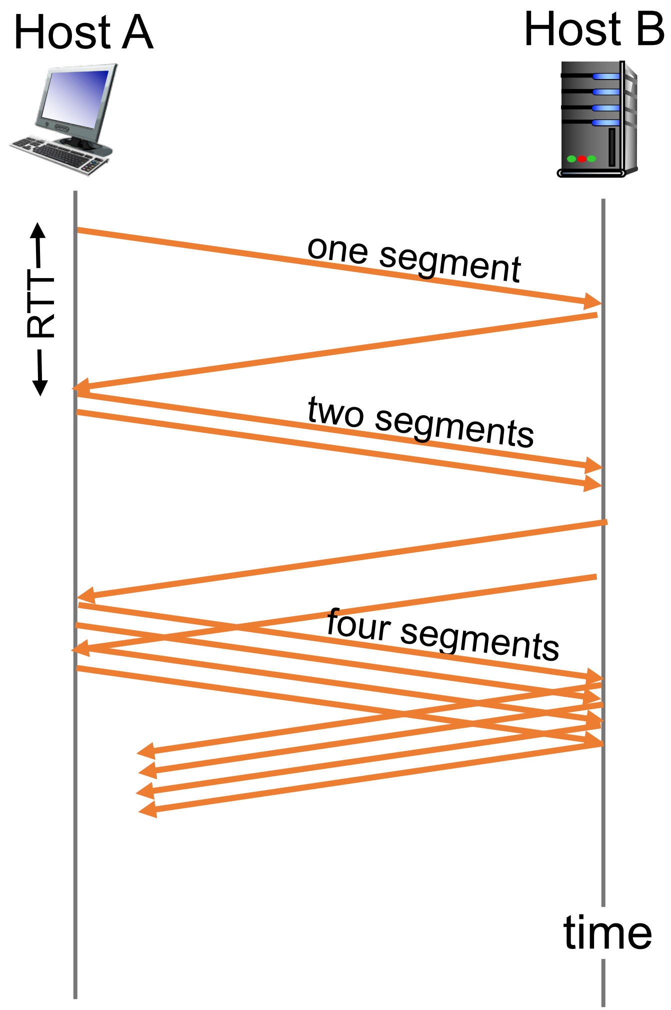
TCP slow start

- When connection begins, increase rate exponentially until first loss event:
  - Initially `cwnd` = 1 MSS
    - MSS is “maximum segment size”
  - Double `cwnd` every RTT
    - Increment `cwnd` for every ACK received

- Initial rate is slow but ramps up exponentially fast

- On loss, restart from `cwnd := 1 MSS`

- Is this good enough?



Problems with slow start

- Congestion window increases too rapidly
  - Example: suppose the right window size `cwnd` is 17
  - `cwnd` would go from 16 to 32 and then dropping down to 1
  - Result: massive packet drops

- Congestion window decreases too rapidly
  - Suppose the right `cwnd` is 31, and there is a loss when `cwnd` is 32
  - Slow start will resume all the way back from `cwnd` 1
  - Result: unnecessarily low throughput

- Perform finer adjustments of `cwnd` based on signals
  - Want a smooth ride, not a jerky one

TCP additive increase

- TCP does slow start until it is "close" to the correct `cwnd`
- And grows `cwnd` much slower after that

- (1) How does TCP know what is the correct `cwnd`?
  - Remember the recent past
  - The last good `cwnd` without packet drop is a good indicator
  - TCP calls this the slow start threshold (`ssthresh`)

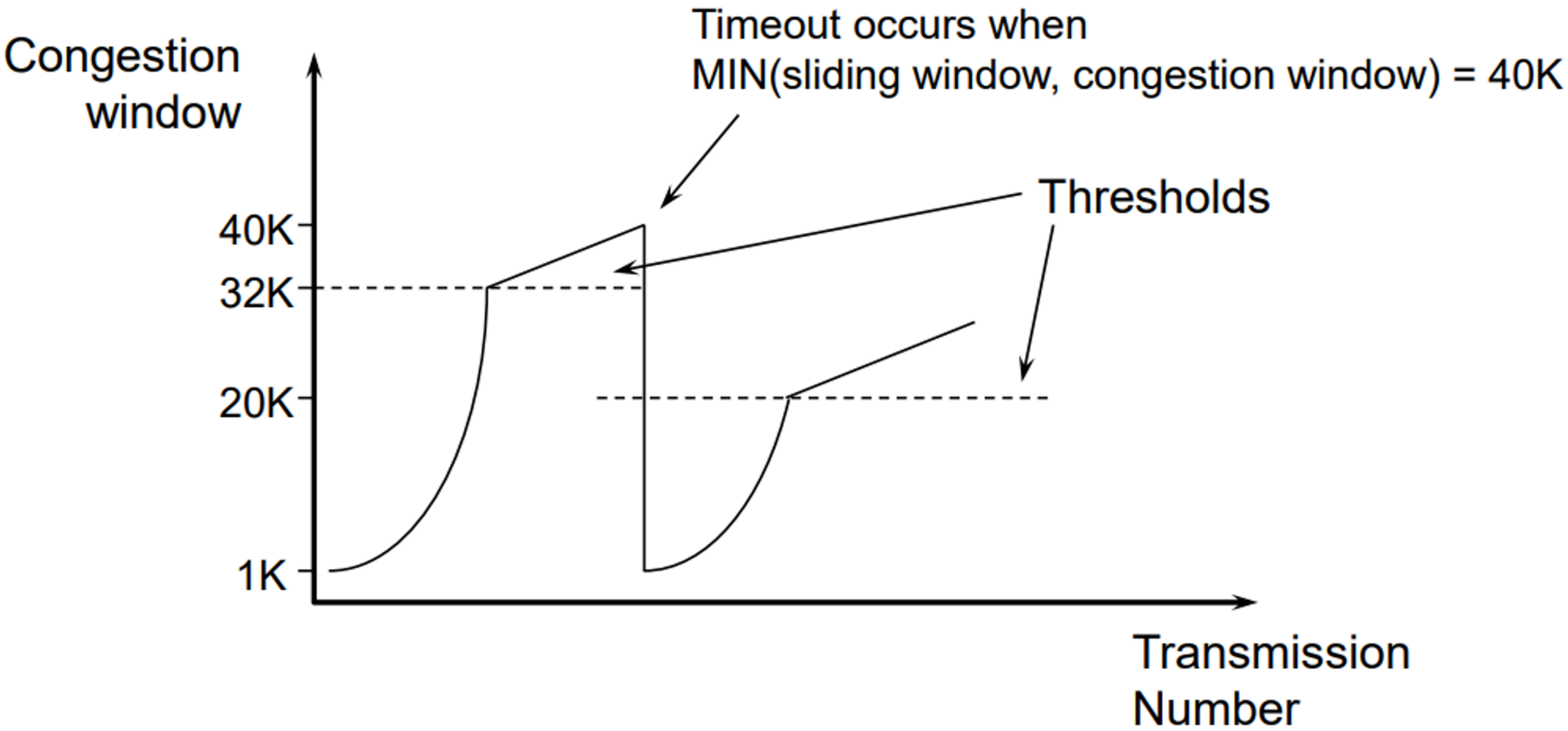
- What is the best way to approach the correct value?
  - TCP uses additive increase after `cwnd` hits `ssthresh`

TCP additive increase

- Start with `ssthresh = 64K bytes`
- Do slow start until `ssthresh`
- Once the threshold is passed, do additive increase
    - For each ACK received, `cwnd = cwnd + (MSS * MSS) / cwnd`
    - i.e., one MSS increase for each `cwnd` worth data ACK'ed
- Upon a TCP timeout,
    - `cwnd = 1 MSS`
    - Set `ssthresh = max(2 * MSS, 0.5 * cwnd)`
    - i.e., the next linear increase will start at half the current `cwnd`

TCP additive increase

Example: Maximum segment size = 1K
Assume thresh=32K



Problems with additive increase

- Current loss detection method: RTO for a given sequence number

- `cwnd` drops rapidly to 1 MSS when there is a timeout
  - When you drive, you don't keep accelerating until you suddenly stop the car right next to a crossing pedestrian

- Could we detect losses earlier (i.e., before RTO) and react proportionately?
  - If you see the pedestrian from afar, you want to slow down gently from a distance

Detecting and reacting to loss

Can we detect loss earlier than RTO?

- Key idea: use the information in the ACKs. How?

- Suppose successive cumulative ACKs contain the same ACK#
  - Also called duplicate ACKs
  - Possible because network is reordering packets
  - Strong indication that the requested sequence number was lost

- Reduce `cwnd` when you see many duplicate ACKs
  - Default threshold: 3 dup ACKs, i.e., triple duplicate ACK
  - TCP sender does two things upon seeing triple dup ACKs

TCP fast retransmit (RFC 2581)

- (1) Gently reduce the `cwnd`, not drop all the way down to 1
- Set `cwnd := cwnd / 2`
  - Multiplicative decrease
- (2) The seq# from dup ACKs is immediately retransmitted
  - i.e., don't wait for RTO if you there is strong evidence packet was lost
- Sender keeps reduced cwnd until a new ACK arrives
  - i.e., an ACK for a seq# transmitted before fast retransmit entered

TCP performs additive increase and multiplicative decrease of its congestion window.

This is often termed AIMD.

Why AIMD?

- Converges to fairness
- Converges to efficiency
- Increments to rate smaller as fairness increases