CSCI-1680
Transport Layer IV

Data over TCP

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, David Mazières, Phil Levis, John Jannotti

Warmup

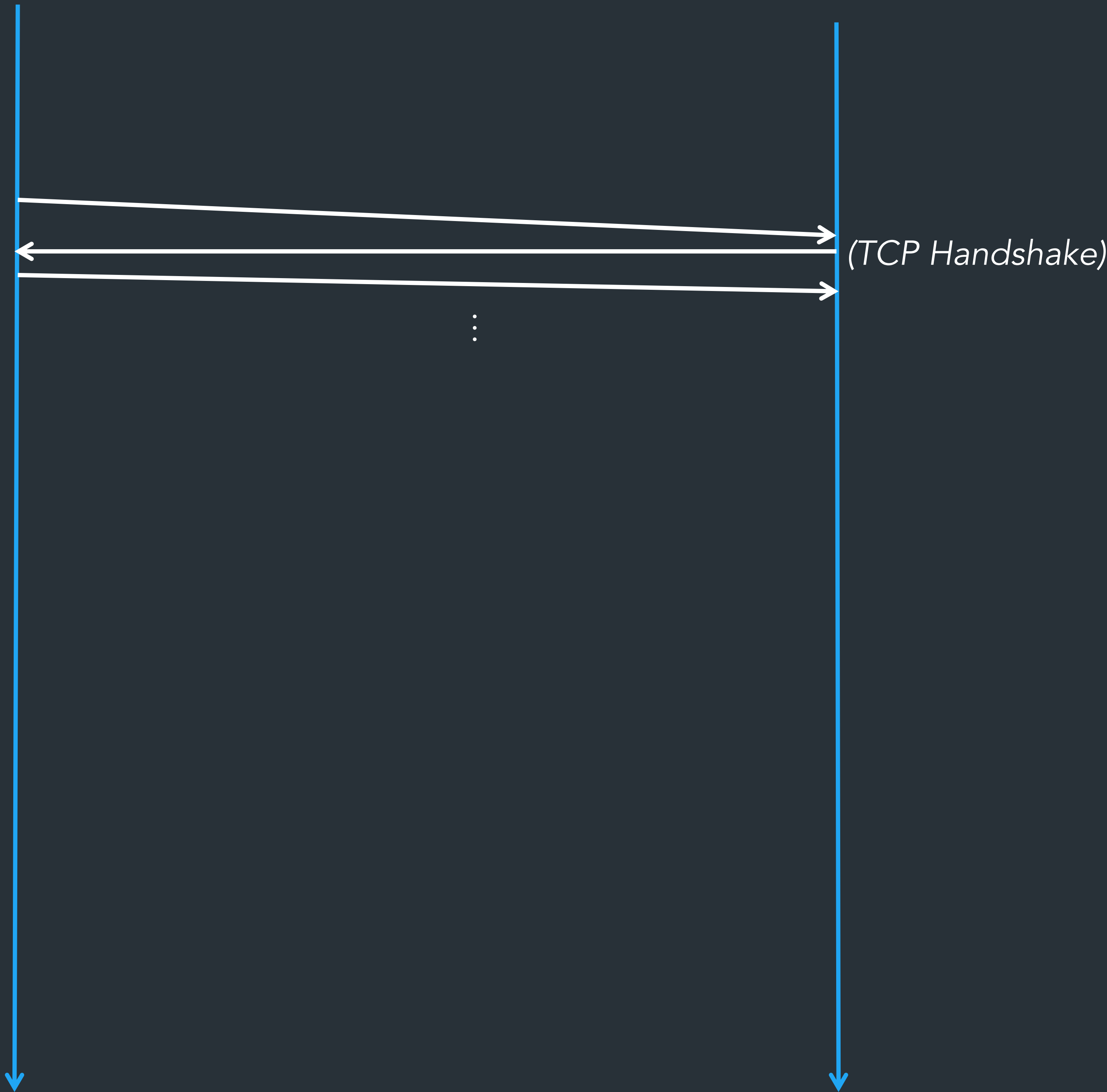
- Sender wants to send “abcdef”
- Max segment size (MSS) = 1
- Receiver’s window = 4

How many packets are sent before the first ACK?

Warmup: Sliding Window

Max segment size (MSS) = 1
Receiver's window = 4
Sender sends "abcdef"

How many packets are sent before the first ACK?
(and what's in them?)



conn.Write("abcdef")

Administrivia

- Sign up for TCP milestone I: this meeting should be this week
- HW4 (short!): out today, one problem, practice for TCP

- TCP Gearup I: new video + notes—take a look if you haven't
- TCP Gearup II: Thursday (11/2) 5-7pm, CIT368
  - Sliding window, how to test/debug

Grading is in progress… we are prioritizing your milestone meetings so you get real-time feedback

Topics for today

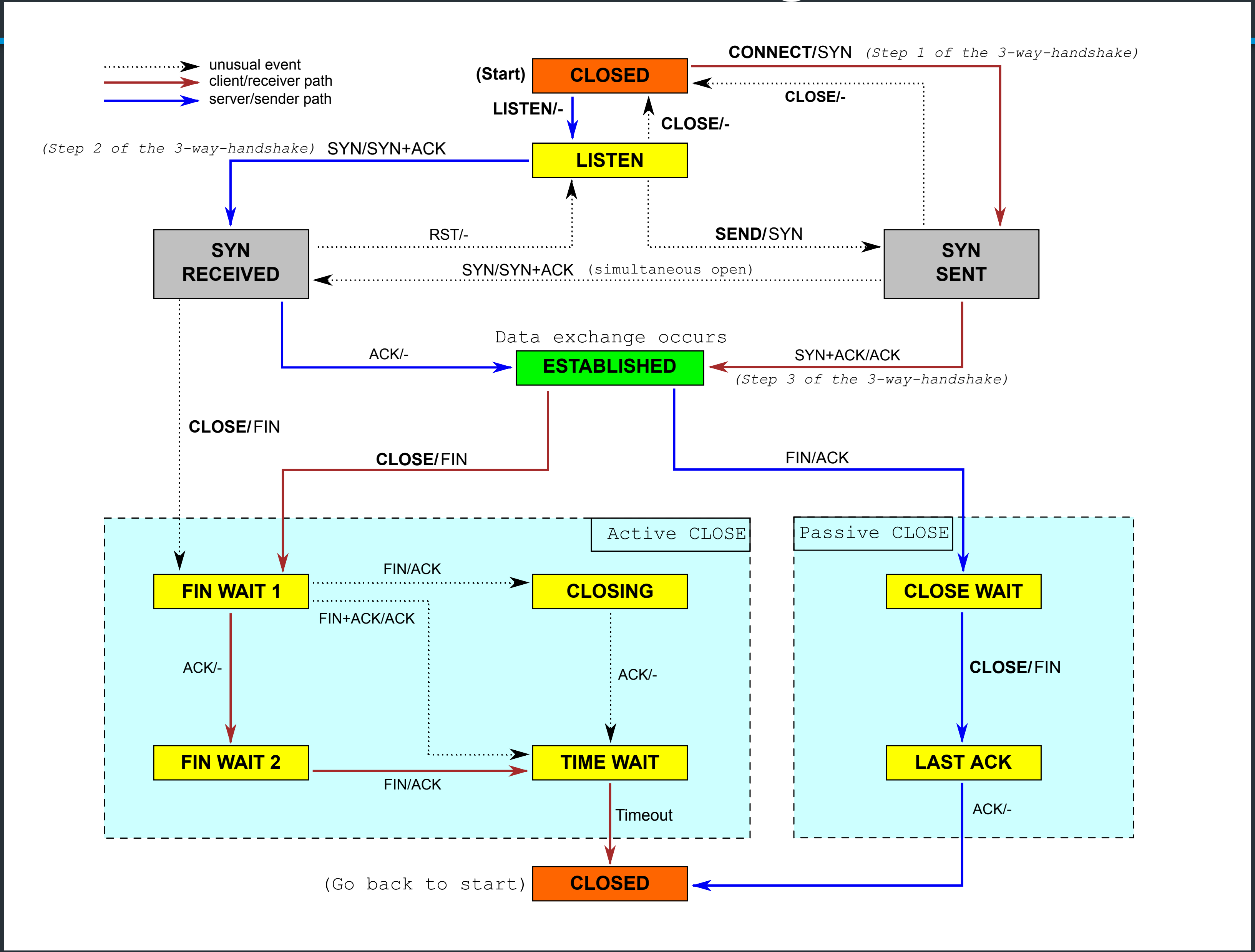
- Connection termination
- Some sending mechanics
- Motivation for congestion control

Connection termination

A 4-step process

- When you have no more data to send, send a FIN
- Both sides close connection separately!

TCP State Diagram



Connection termination

A 4-step process

- When you have no more data to send, send a FIN
- Both sides close connection separately!

- How to know when last ACK received?
- Initiating side must wait for 2*MSL before deleting TCB

  => MSL = Longest time a segment might be delayed
  (configurable, ~1min)

Why do we need to wait this long?

Other mechanics for sending packets
(used in modern TCPs, not required for project)

Example: telnet/SSH

Terminal input <=> TCP connection

Example: telnet/SSH

Terminal input <=> TCP connection

Problems

=> Tiny packets means high overhead!

=> But also don’t want to add latency

=> How to decide when to send? Multiple strategies.

One way: add some more logic to the sender

Nagle’s algorithm

Goal: reduce the overhead of small packets

```
if (there is data to send) and (window >= MSS)
  Send a MSS segment
else
  if there is unAcked data in flight
          buffer the new data until ACK arrives
  else
          send all the new data now
```

One way: add some more logic to the sender

Nagle's algorithm

Goal: reduce the overhead of small packets

```
if (there is data to send) and (window >= MSS)
  Send a MSS segment
else
  if there is unAcked data in flight
          buffer the new data until ACK arrives
  else
          send all the new data now
```

Recommended in some cases, but waiting to send not always a great idea
=> Configurable on socket creation

Another way: change how the receiver advertises the window

What if receiving app only reads 1 byte at a time?

Another way: change how the receiver advertises the window

What if receiving app only reads 1 byte at a time?

Silly Window Syndrome (SWS) Avoidance: when window is zero, wait until 1MSS of receive buffer space is available before advertising nonzero window

Another way: change the receiver

What if receiving app only reads 1 byte at a time?

> Silly Window Syndrome (SWS) Avoidance: when window is zero, wait until 1MSS of receive buffer space is available before advertising nonzero window

Yet another way: receiver could delay sending ACK for short time (400ms), in case it has data to send

=> All data segments are ACKs, so why send packet again?

Delayed Acknowledgments

- Goal: Piggy-back ACKs on data
  - Delay ACK for 200ms in case application sends data
  - If more data received, immediately ACK second segment
  - Note: never delay duplicate ACKs (if missing a segment)

- Warning: can interact badly with Nagle for some applications
  - Nagle waits for ACK until send => Temporary deadlock
  - App can disable Nagle with `TCP_NODELAY`
  - App should also avoid many small writes

Congestion control: the start

The story so far

Flow control provides reliable, in-order delivery

Goal: send as much data as receiver can handle

- Receiver's advertised window: sent with every ACK
- Sliding window: increase throughput by having multiple packets in flight

Problems?

What would happen with our current sliding window implementation?

What else do we need?

- Flow control provides *correctness: reliable, in order delivery*
- Need more for performance
  - What if the network is the bottleneck?

How do we know when the network is overloaded?

What can go wrong?

Congestion control

We must not send more data than the network can handle

What happens if we do?

A Short History of TCP

- 1974: 3-way handshake
- 1978: IP and TCP split
- 1983: January 1st, ARPAnet switches to TCP/IP
- 1984: Nagle predicts congestion collapses
- 1986: Internet begins to suffer congestion collapses
  - LBL to Berkeley drops from 32Kbps to 40bps
- 1987/8: Van Jacobson fixes TCP, publishes seminal paper*: (TCP Tahoe)
- 1990: Fast transmit and fast recovery added (TCP Reno)

* Van Jacobson and Michael Karels. Congestion avoidance and control. SIGCOMM ’88

Congestion Collapse

Nagle, rfc896, 1984

- Mid 1980's: Problem with the protocol *implementations*, not the protocol!
- What was happening?
- If close to capacity, and, e.g., a large flow arrives suddenly…
  - RTT estimates become too short
  - Lots of retransmissions → increase in queue size
  - Eventually many drops happen (full queues)
  - Fraction of useful packets (not copies) decreases

The problem

- https://witestlab.poly.edu/respond/sites/genitutorial/files/tcp-aimd.ogv

TCP Congestion Control

- 3 Key Challenges
  - Determining the available capacity in the first place
  - Adjusting to changes in the available capacity
  - Sharing capacity between flows

- Idea
  - Each source determines network capacity for itself
  - Rate is determined by window size
  - Uses implicit feedback (drops, delay)
  - ACKs pace transmission (self-clocking)

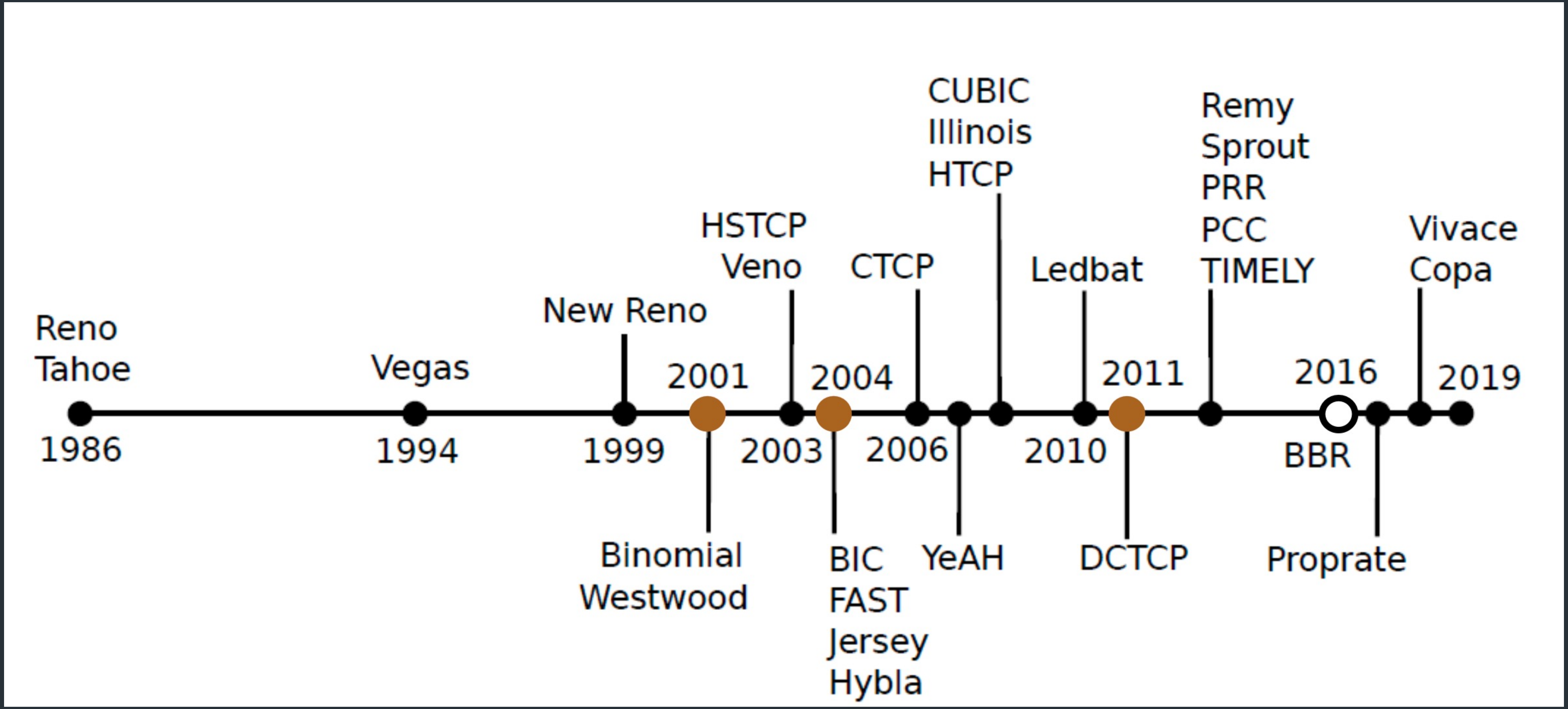
Congestion control has a long history

- Active research area for ~40 years

- I am nowhere close to being an expert

- My hope is to get you to understand the problems involved

Timeline of (some!) congestion control implementations



[“The great Internet congestion control census” (2019)]

Just a few TCP implementations

What's the difference?

General usage

- Reno (1980s)
- Tahoe
- Vegas
- New Vegas
- Westwood
- Cubic
- BBR (2016)
- …

The main idea

Goals

- Determine initial network capacity
- Adjust sending rate as capacity changes

- How? Maintain two windows:
  - Advertised Window (from receiver)
  - Congestion window (cwnd)

Sending rate = min(Advertised Window, cwnd)

- Ideally, want to have sending rate: ~= Window/RTT

Dealing with Congestion

To start:

- Assume losses are due to congestion
- After a loss, reduce congestion window
  - How much to reduce?
- Idea: conservation of packets at equilibrium
  - Want to keep roughly the same number of packets in network
  - Analogy with water in fixed-size pipe
  - Put new packet into network when one exits

Classical Congestion Control

- Loss-based: assume packet loss => congestion
- TCP Tahoe (1988)
  - Slow start, congestion avoidance, fast retransmit
- TCP Reno (1990)
  - TCP Tahoe + Fast recovery
- Many variations developed from this… (see optional readings)

Modes of operation

- Slow start (SS)
  - Determine initial window, recover after loss
- Congestion avoidance (CA)
  - Steady state, slowly probe for changes in capacity

Congestion Avoidance

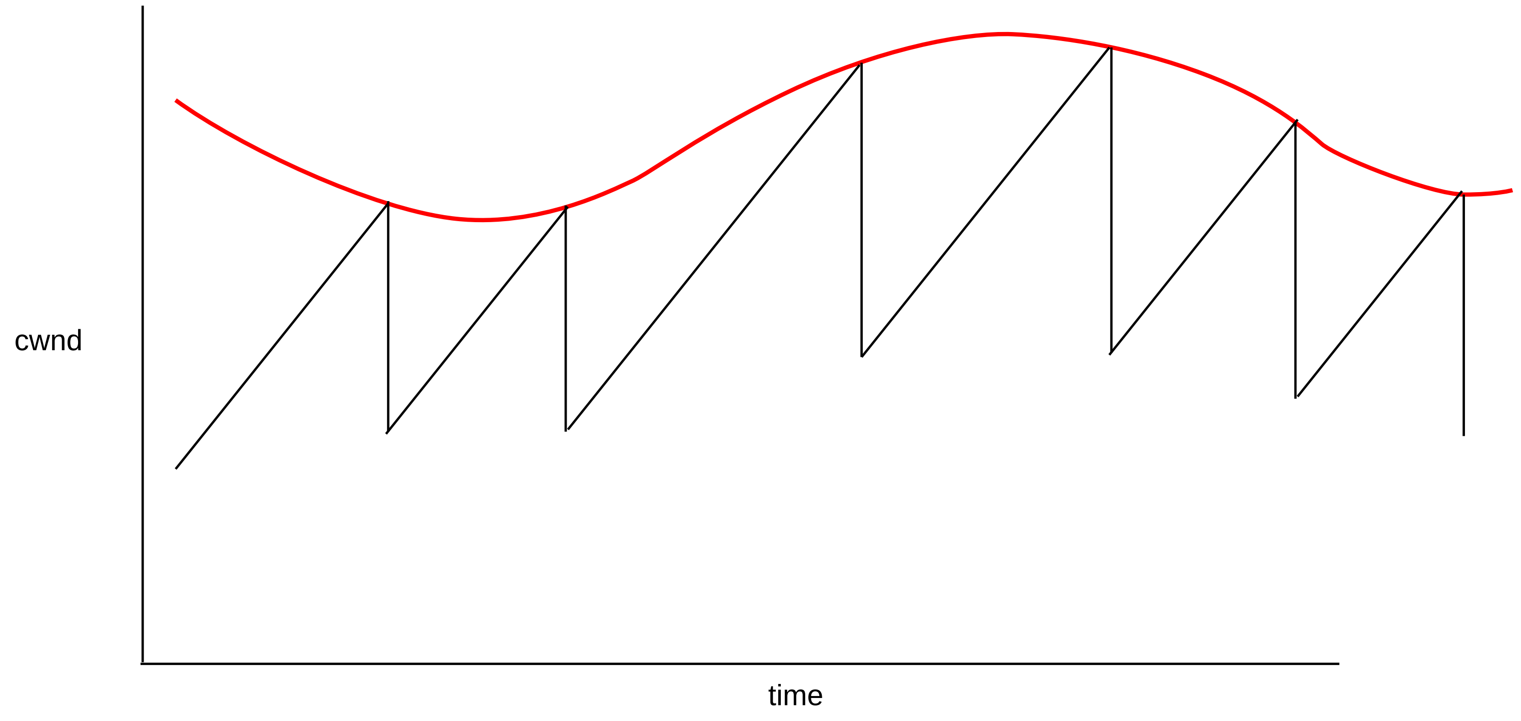
After finishing a window, recompute cwnd:

- If no losses, cwnd = cwnd + MSS
  - (Often written as cwnd += 1)
- If packets were lost: cwnd = cwnd/2

This is known as additive increase, multiplicative decrease (AIMD)

- Slowly increase capacity
- Dramatically scale back on loss

AIMD Example



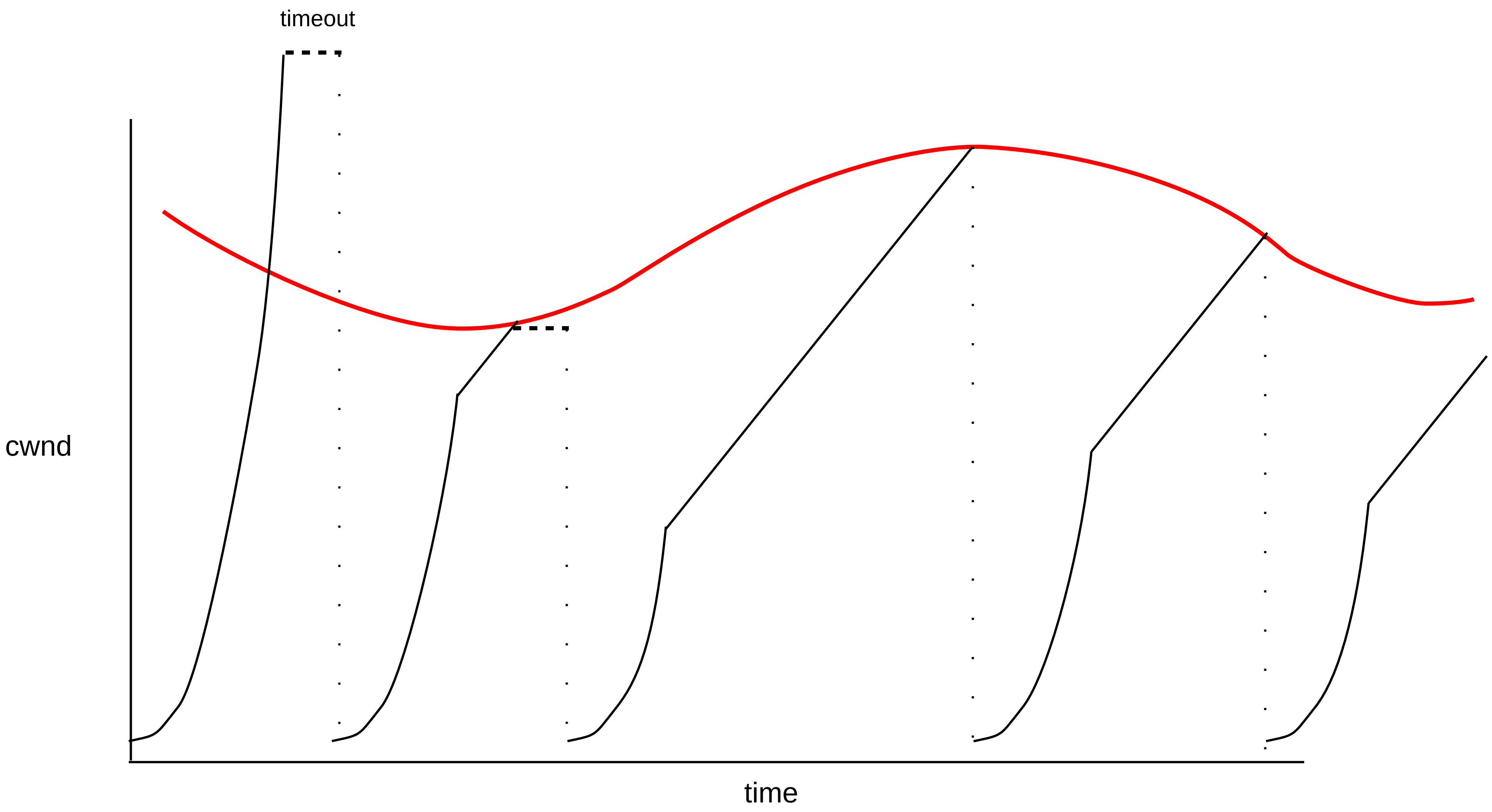
TCP Sawtooth, red curve represents the network capacity

Slow Start

After finishing a window

- cwnd = cwnd * 2
- Continue doing this until you experience a loss

- After first loss, keep slow-start threshold (ssthresh):
  - If window < ssthresh: slow-start
  - If window > ssthresh: congestion avoidance
- After first loss: ssthresh = cwnd / 2

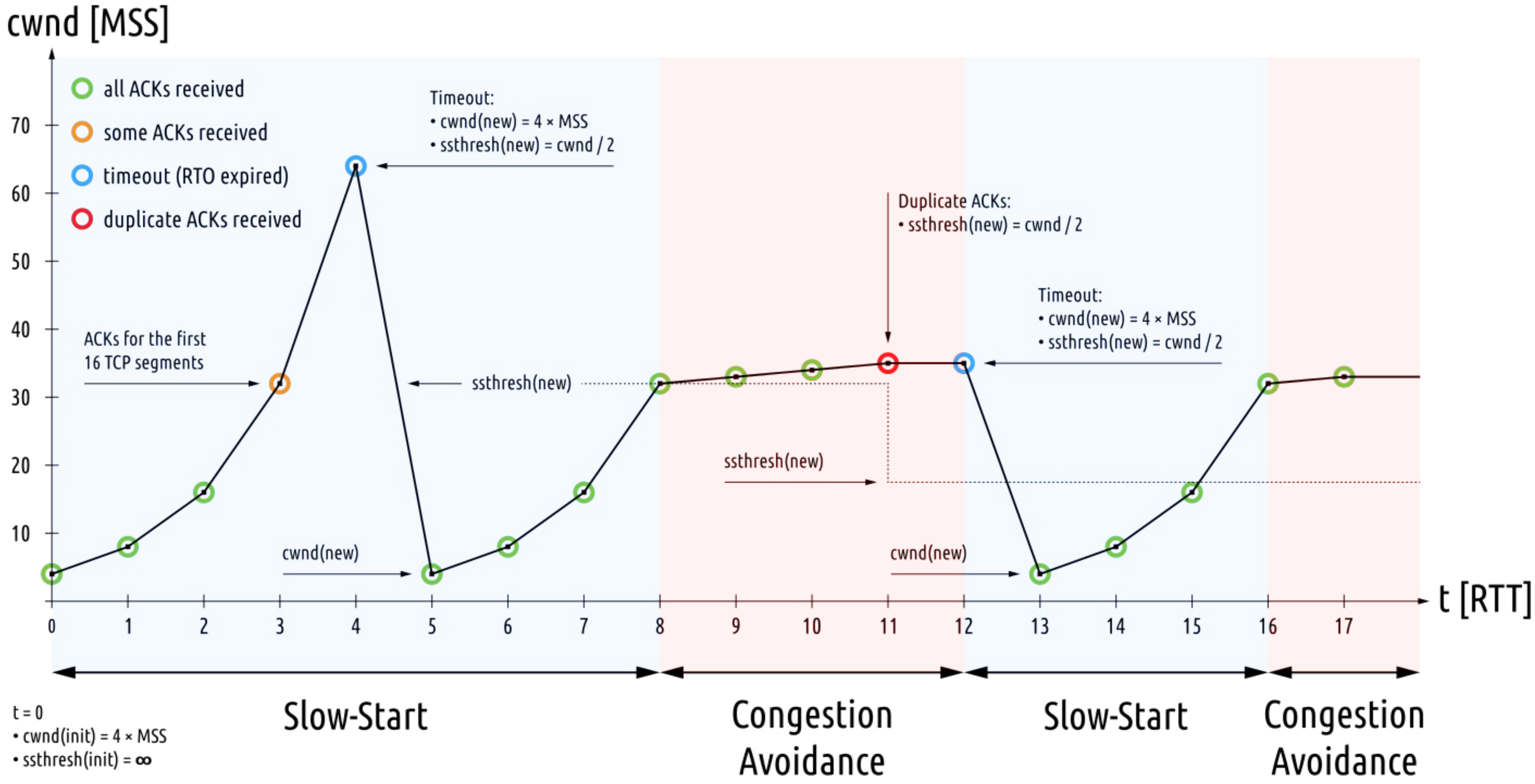


TCP Tahoe Sawtooth, red curve represents the network capacity
Slow Start is used after each packet loss until ssthresh is reached

How to Detect Loss

- Timeout
- Any other way?
  - Gap in sequence numbers at receiver
  - Receiver uses cumulative ACKs: drops => duplicate ACKs
- “Fast recovery”: 3 Duplicate ACKs considered loss

- Which one is worse?


cwnd [MSS]
all ACKs received
some ACKs received
timeout (RTO expired)
duplicate ACKs received
70
60
50
40
30
20
10
Timeout:
• cwnd(new) = 4 × MSS
• ssthresh(new) = cwnd / 2
ACKs for the first
16 TCP segments
ssthresh(new)
cwnd(new)
Duplicate ACKs:
• ssthresh(new) = cwnd / 2
Timeout:
• cwnd(new) = 4 × MSS
• ssthresh(new) = cwnd / 2
ssthresh(new)
cwnd(new)
t [RTT]
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
t = 0
• cwnd(init) = 4 × MSS
• ssthresh(init) = ∞
Slow-Start
Congestion Avoidance
Slow-Start
Congestion Avoidance

Slow start every time?!

- Losses have large effect on throughput
- Fast Recovery (TCP Reno)
  - Same as TCP Tahoe on Timeout: w = 1, slow start
  - On triple duplicate ACKs: w = w/2
  - Retransmit missing segment (fast retransmit)
  - Stay in Congestion Avoidance mode
- Why 3 dup-acks instead of just 1?

This is just the beginning…

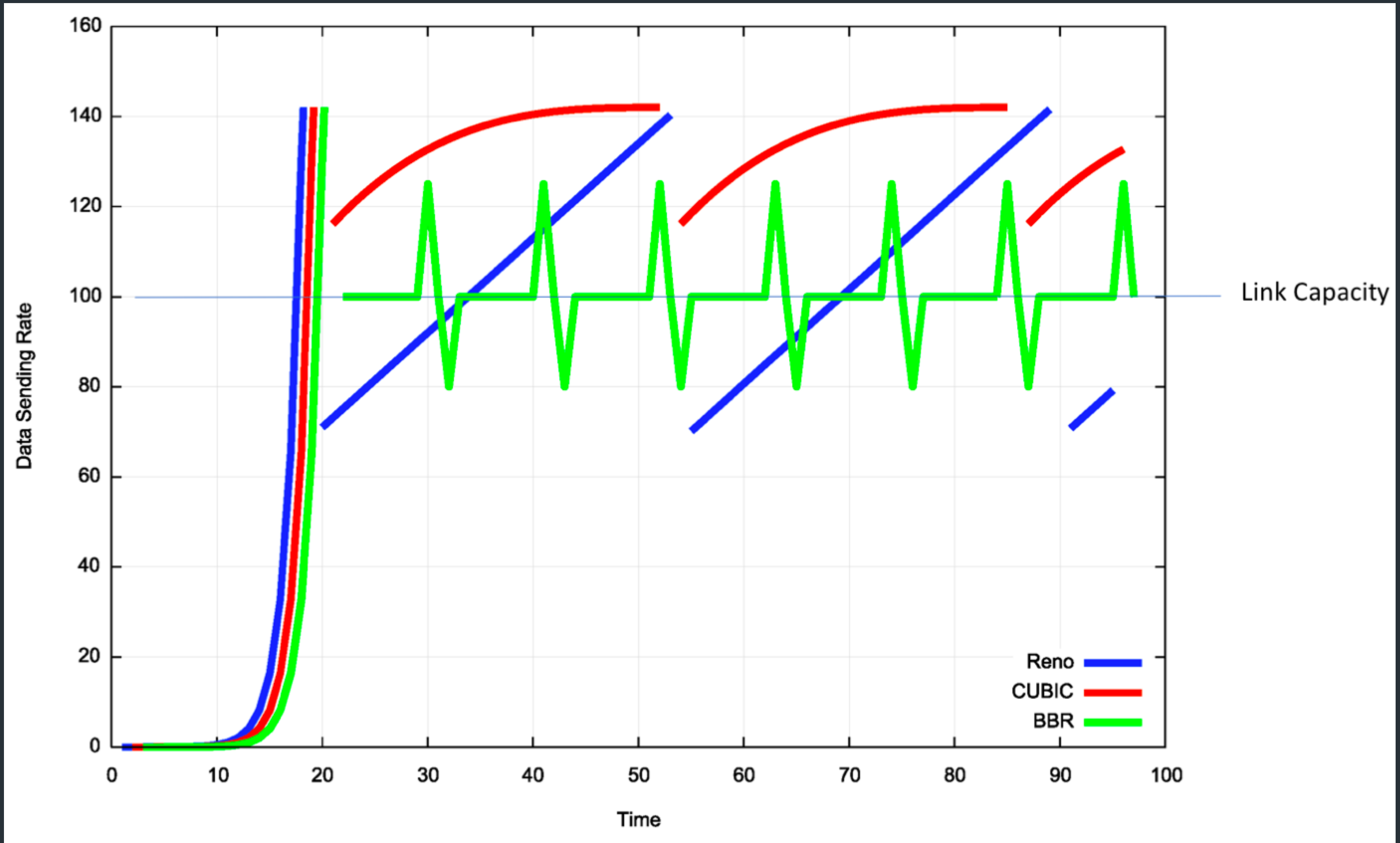
Lots of congestion control schemes, with different strategies/goals:

- Tahoe (1988)
- Reno (1990)
- Vegas (1994): Detect based on RTT
- New Reno: Better recovery multiple losses
- Cubic (2006): Linux default, window size scales by cubic function
- BBR (2016): Used by Google, measures bandwidth/RTT
- …

BBR

- Problem: can’t measure both RTT_{prop} and Bottleneck BW at the same time
- BBR:
  - Slow start
  - Measure throughput when RTT starts to increase
  - Measure RTT when throughput is still increasing
  - Pace packets at the BDP
  - Probe by sending faster for 1RTT, then slower to compensate

BBR



From: https://labs.ripe.net/Members/gih/bbr-tcp

Help from the network

- What if routers could *tell* TCP that congestion is happening?
  - Congestion causes queues to grow: rate mismatch
- TCP responds to drops
- Idea: Random Early Drop (RED)
  - Rather than wait for queue to become full, drop packet with some probability that increases with queue length
  - TCP will react by reducing cwnd
  - Could also mark instead of dropping: ECN

Help from the network

- What if routers could *tell* TCP that congestion is happening?
  - Congestion causes queues to grow: rate mismatch

Know: TCP responds to drops

- Idea: Random Early Drop (RED)
  - Rather than wait for queue to become full, drop packet with some probability that increases with queue length
  - TCP will react by reducing cwn

RED Advantages

- Probability of dropping a packet of a particular flow is roughly proportional to the share of the bandwidth that flow is currently getting
- Higher network utilization with low delays
- Average queue length small, but can absorb bursts

But can we do better?

ECN

What if we didn’t have to drop packets?

- Routers/switches set bits in packet to indicate congestion

ECN

What if we didn’t have to drop packets?

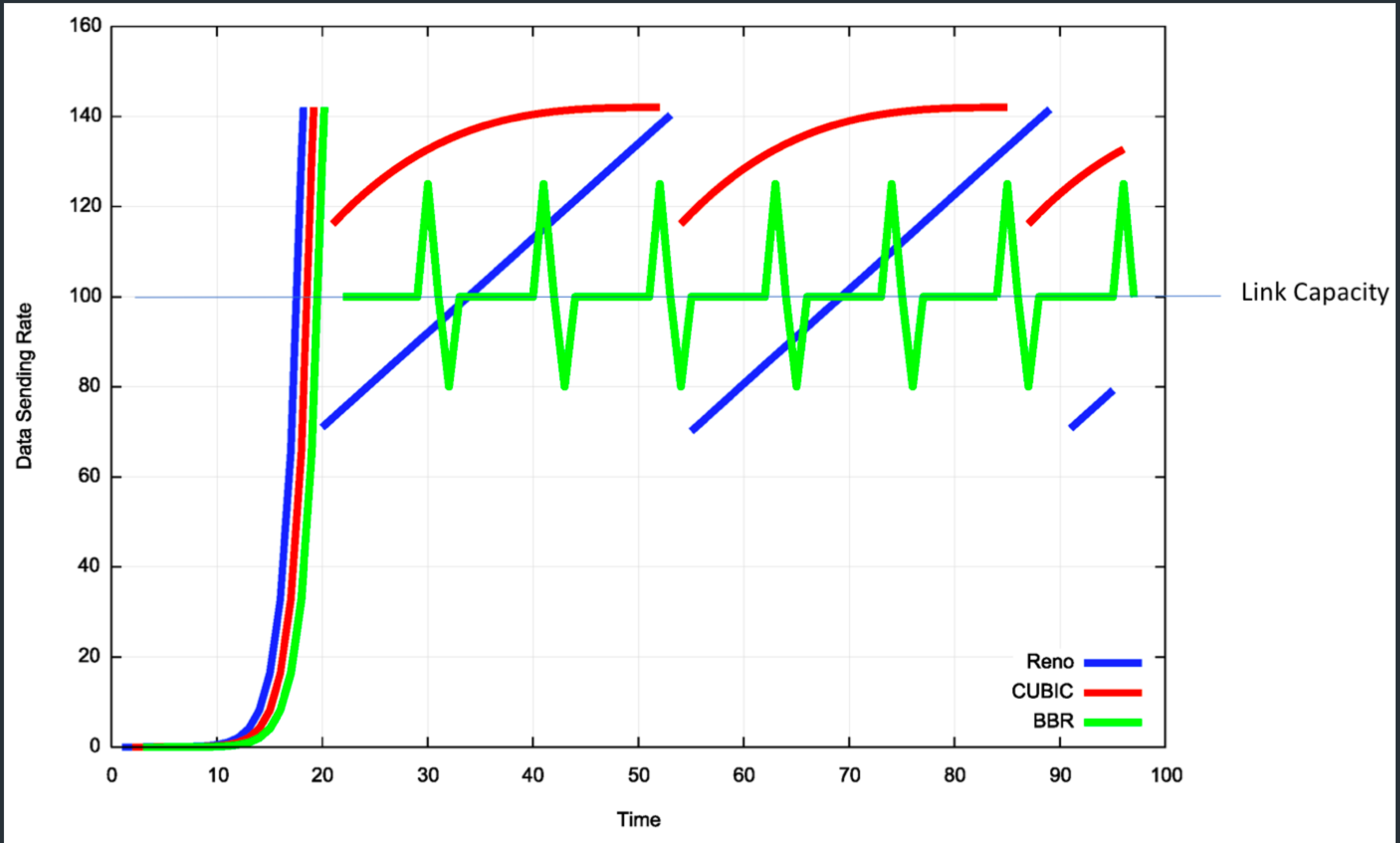
- Routers/switches set bits in packet to indicate congestion

- When sender sees congestion bit, scales back cwnd
- Must be supported by both sender and receiver

=>Avoids retransmissions optionally dropped packets

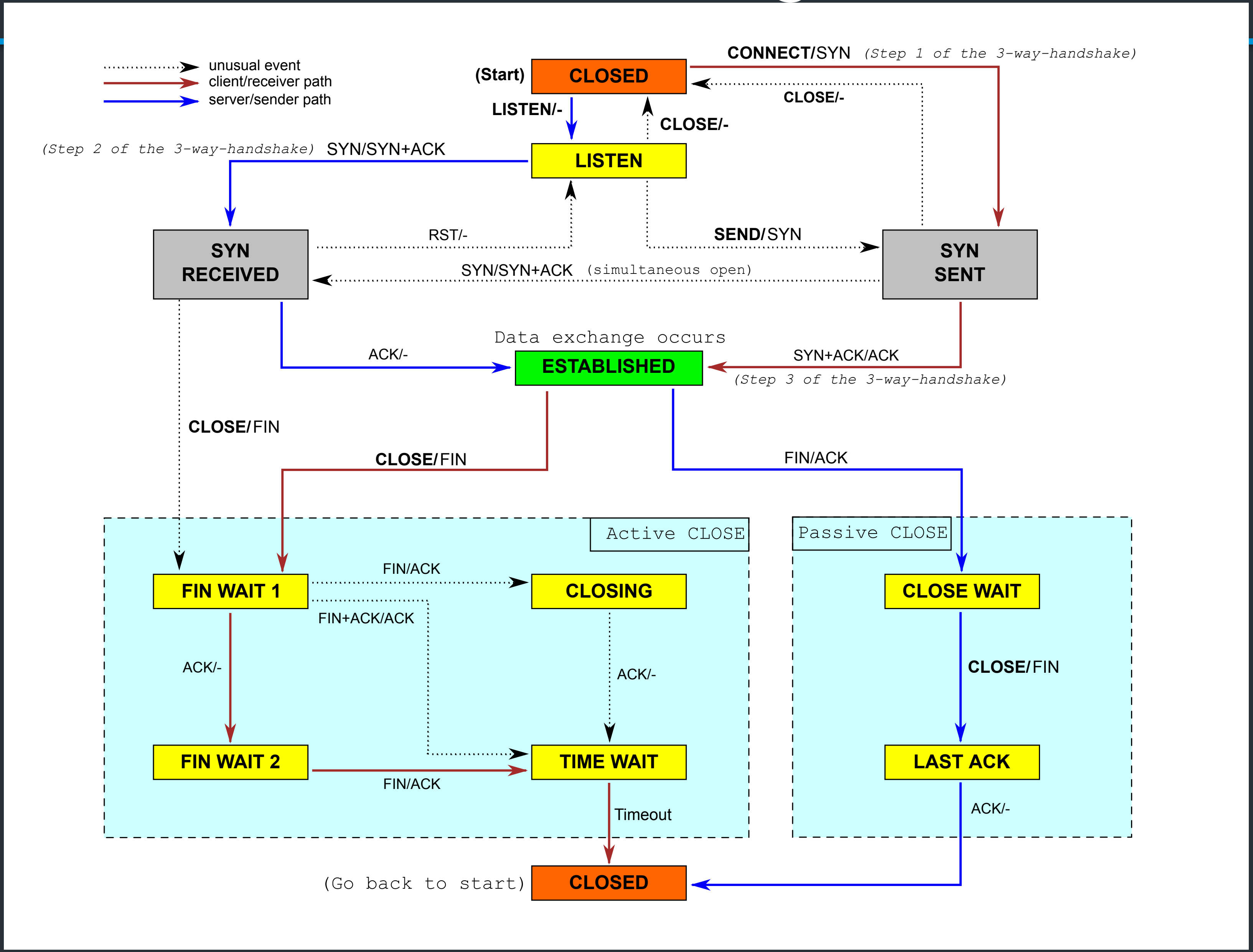
Special purpose example: DCTCP

BBR



From: https://labs.ripe.net/Members/gih/bbr-tcp

TCP State Diagram

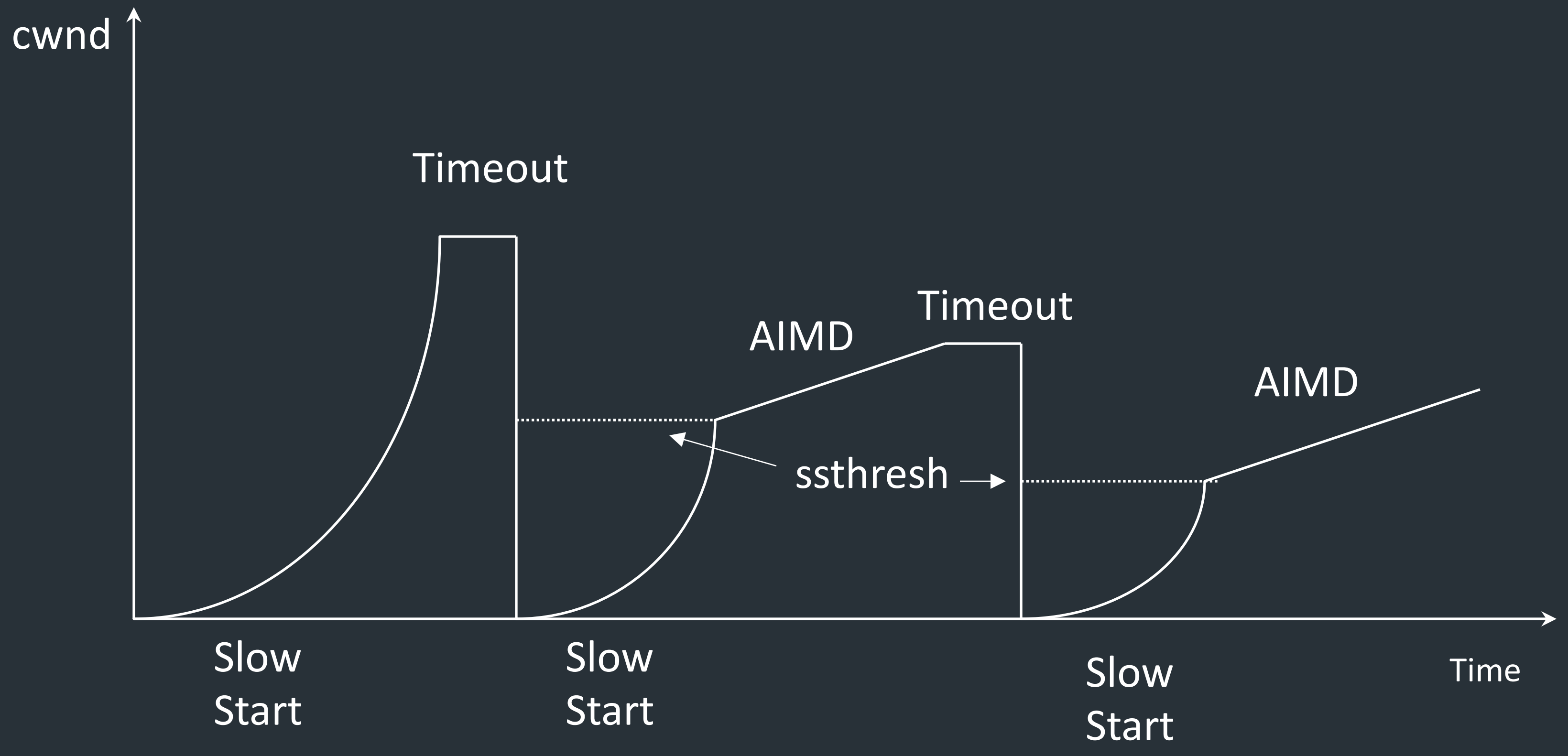


TCP Header

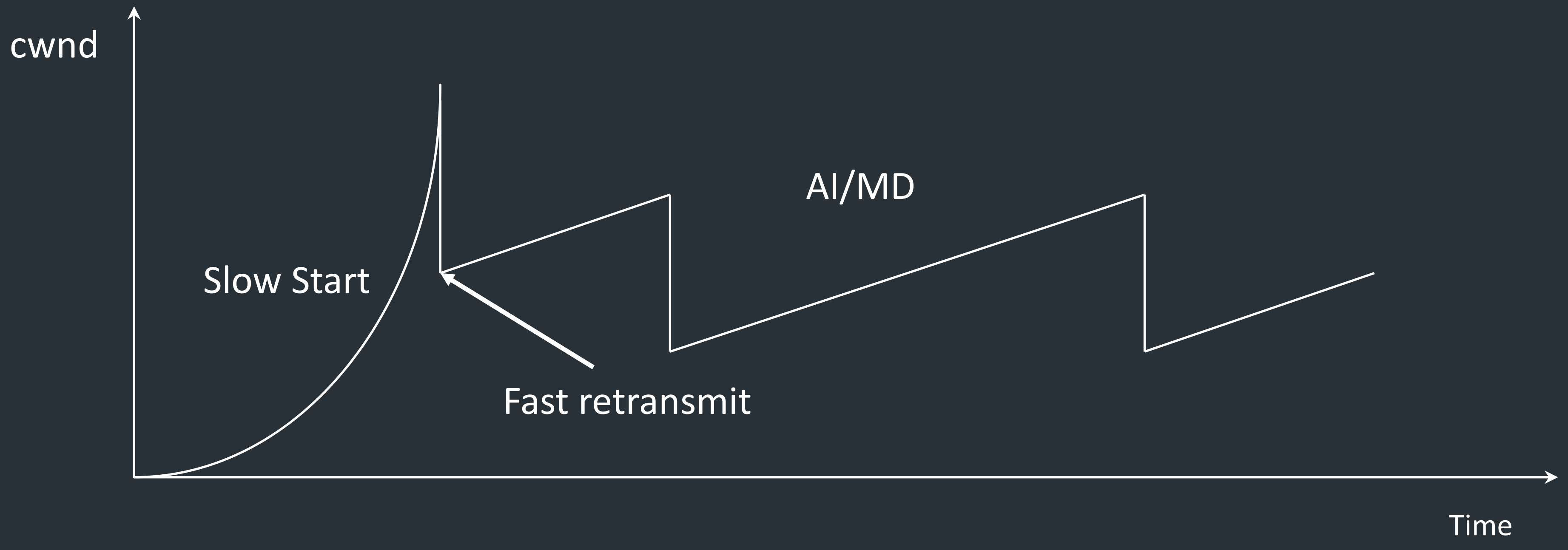
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Extra congestion control content

Putting it all together

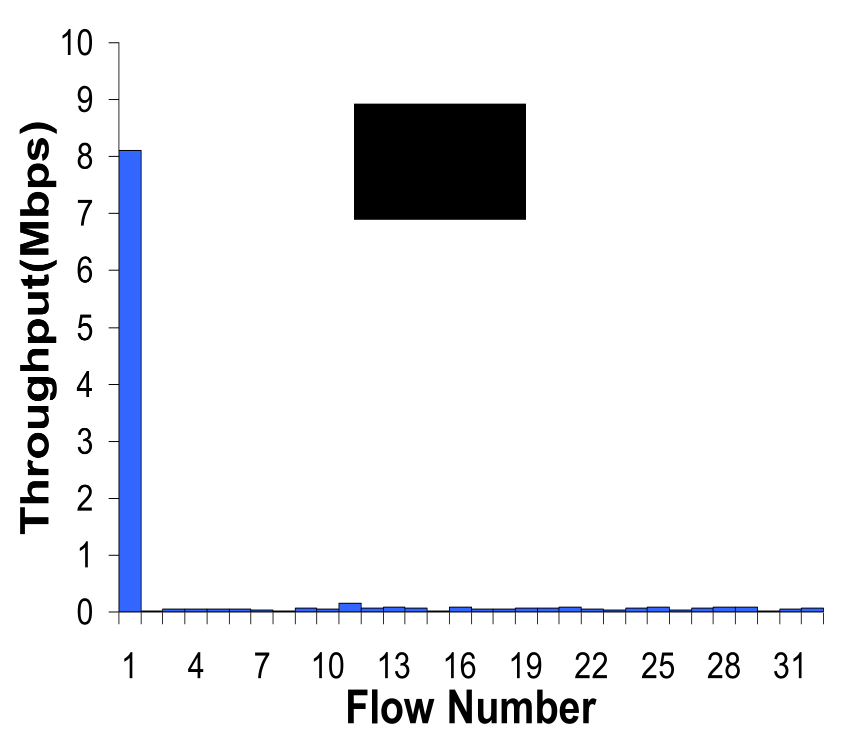


Fast Recovery and Fast Retransmit



TCP Friendliness

- Can other protocols co-exist with TCP?
  - E.g., if you want to write a video streaming app using UDP, how to do congestion control?



[redacted]

1 UDP Flow at 10MBps
31 TCP Flows
Sharing a 10MBps link

TCP Friendliness

- Can other protocols co-exist with TCP?
  - E.g., if you want to write a video streaming app using UDP, how to do congestion control?
- Equation-based Congestion Control
  - Instead of implementing TCP's CC, estimate the rate at which TCP would send. Function of what?
  - RTT, MSS, Loss
- Measure RTT, Loss, send at that rate!

TCP Throughput

- Assume a TCP congestion of window W (segments), round-trip time of RTT, segment size MSS
  - Sending Rate $S = W$ x MSS / RTT (1)
- Drop: W = W/2
  - grows by *MSS* for *W/2 RTTs*, until another drop at $W \approx W$
- Average window then 0.75x*S*
  - From (1), $S = 0.75\ W\ MSS / RTT$ (2)
- Loss rate is 1 in number of packets between losses:
  - Loss = 1 / (1 + ($W/2 + W/2{+}1 + W/2 + 2\ + \ldots + W$)
    = 1 / (3/8 W^2) (3)

TCP Throughput (cont)

- Loss = $8/(3W^2)$ (4)

$$\Rightarrow W = \sqrt{\frac{8}{3 \cdot Loss}}$$

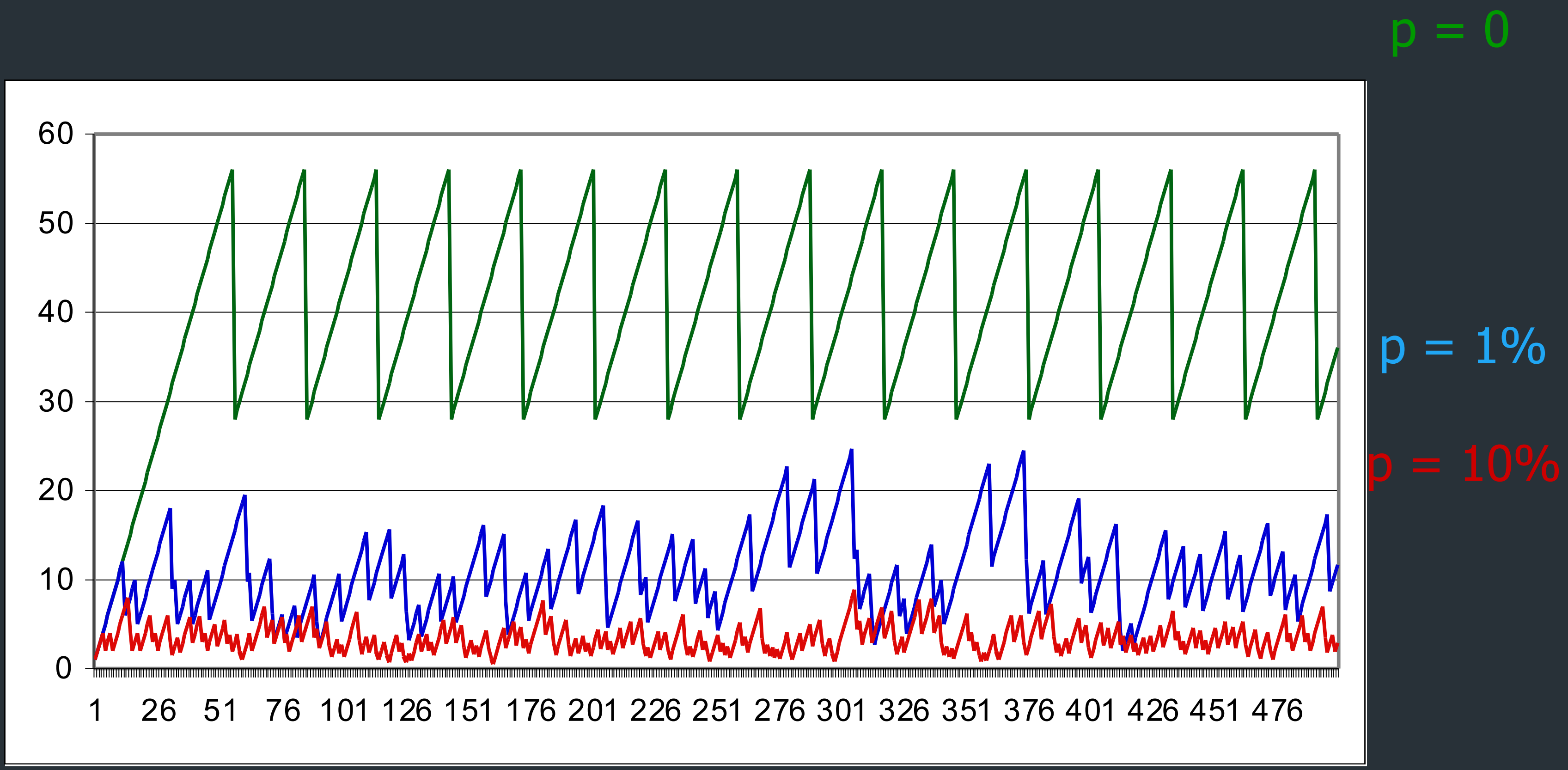
- Substituting (4) in (2), $S = 0.75\ W\ MSS\ /\ RTT$,

Throughput ≈ $$1.22 \times \frac{MSS}{RTT \cdot \sqrt{Loss}}$$

- Equation-based rate control can be TCP friendly and have better properties, e.g., small jitter, fast ramp-up…

What Happens When Link is Lossy?

- Throughput ≈ 1 / sqrt(Loss)



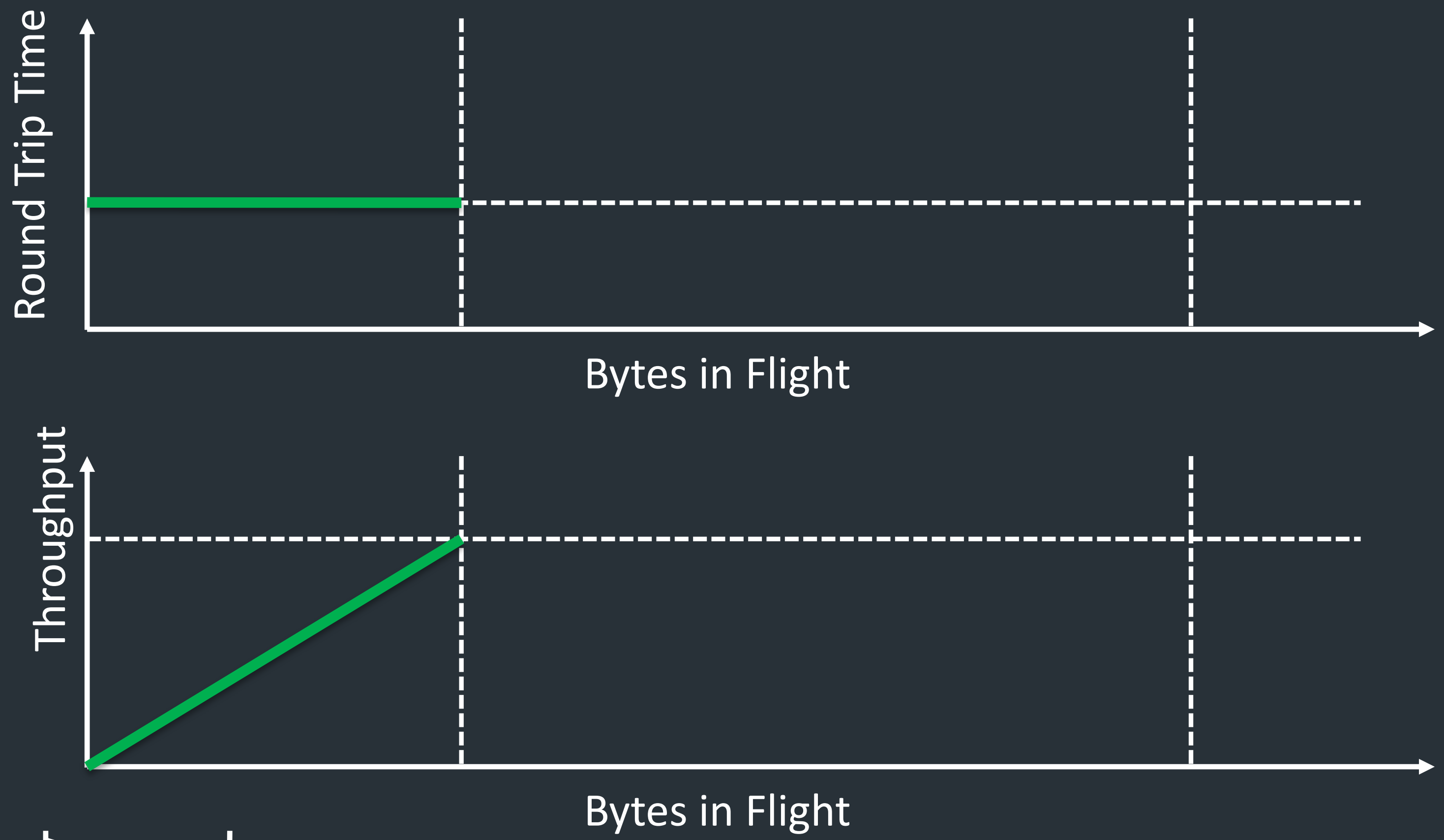
What can we do about it?

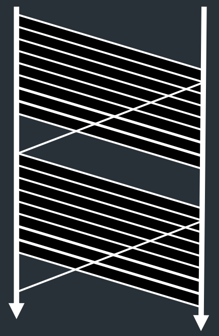
- Two types of losses: congestion and corruption
- One option: mask corruption losses from TCP
  - Retransmissions at the link layer
  - E.g. Snoop TCP: intercept duplicate acknowledgments, retransmit locally, filter them from the sender
- Another option:
  - Tell the sender about the cause for the drop
  - Requires modification to the TCP endpoints

Congestion Avoidance

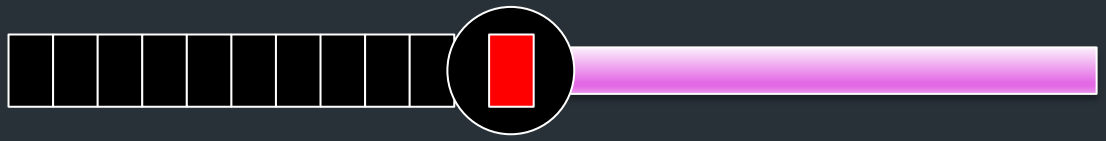
- TCP creates congestion to then back off
  - Queues at bottleneck link are often full: increased delay
  - Sawtooth pattern: jitter
- Alternative strategy
  - Predict when congestion is about to happen
  - Reduce rate early
- Other approaches
  - Delay Based: TCP Vegas (not covered)
  - Better model of congestion: BBR
  - Router-centric: RED, ECN, DECBit, DCTCP

Another view of Congestion Control



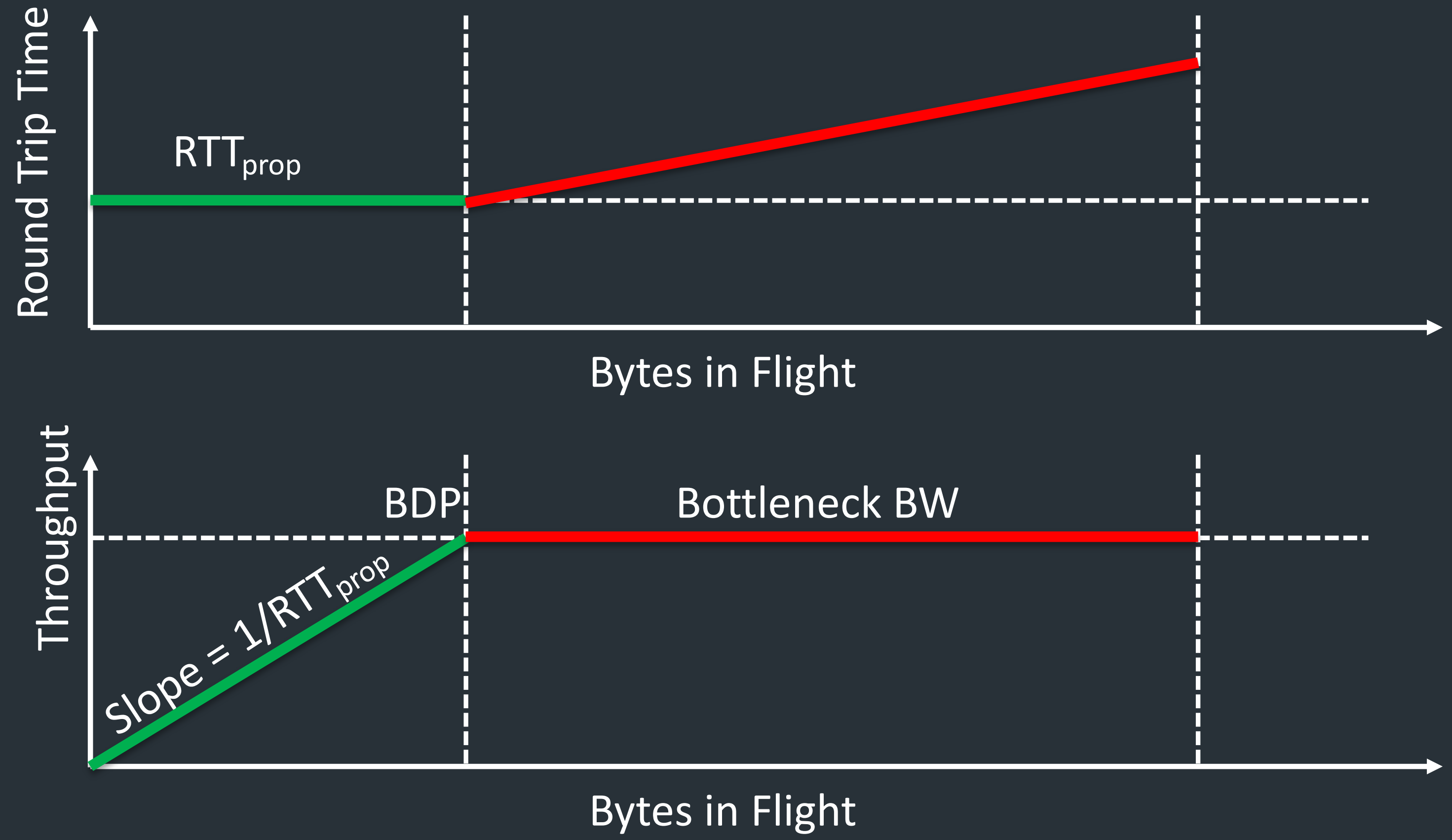


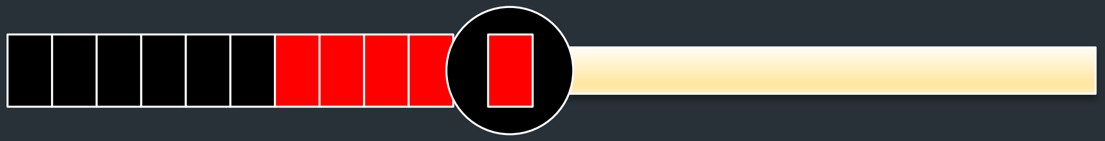
Tput =
InFlight/
RTT_{prop}



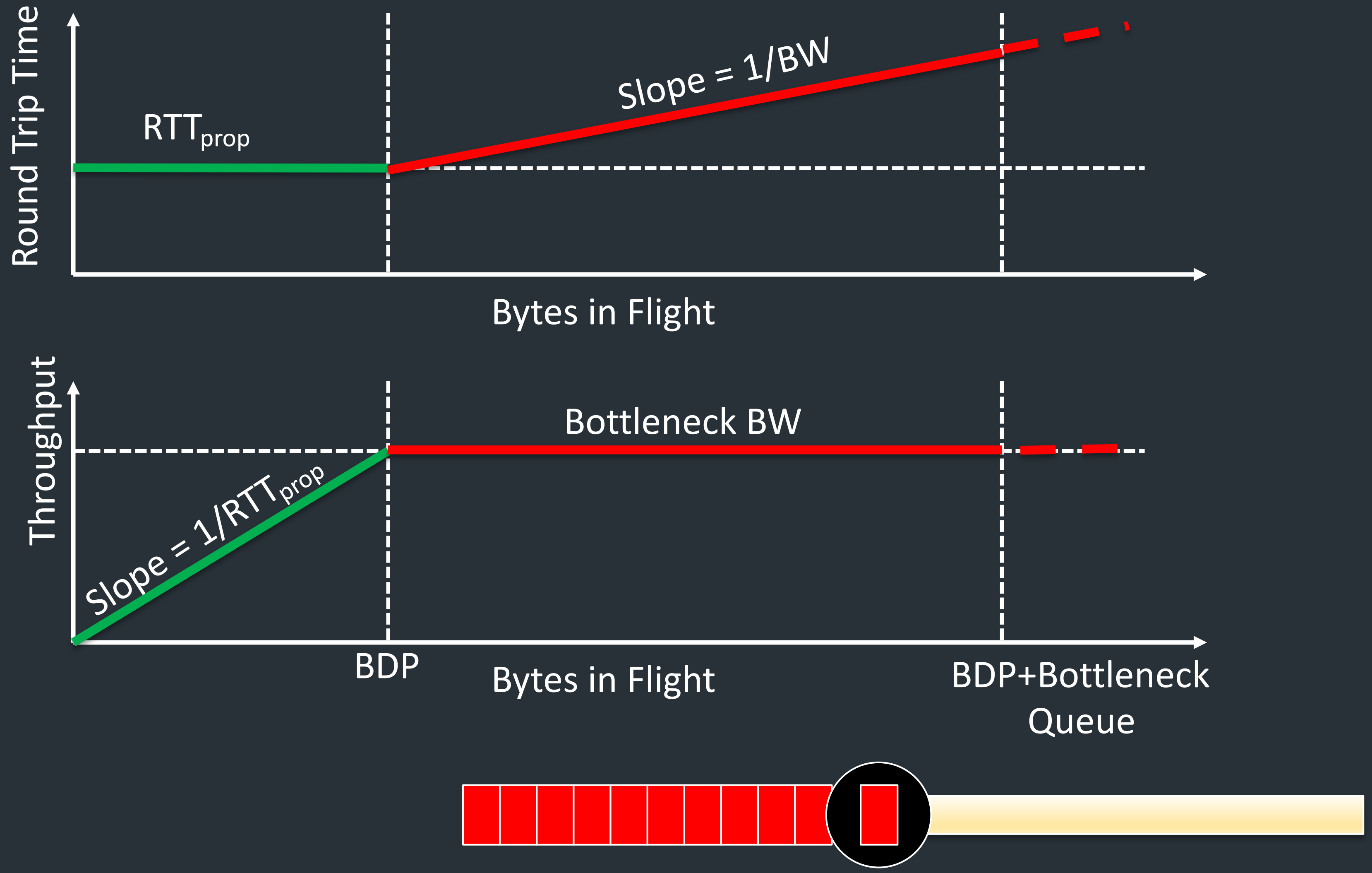
Diagrams based on Cardwell et al., BBR: Congestion Based Congestion Control," Communications of the ACM, Vol. 60 No. 2, Pages 58-66.

Another view of Congestion Control

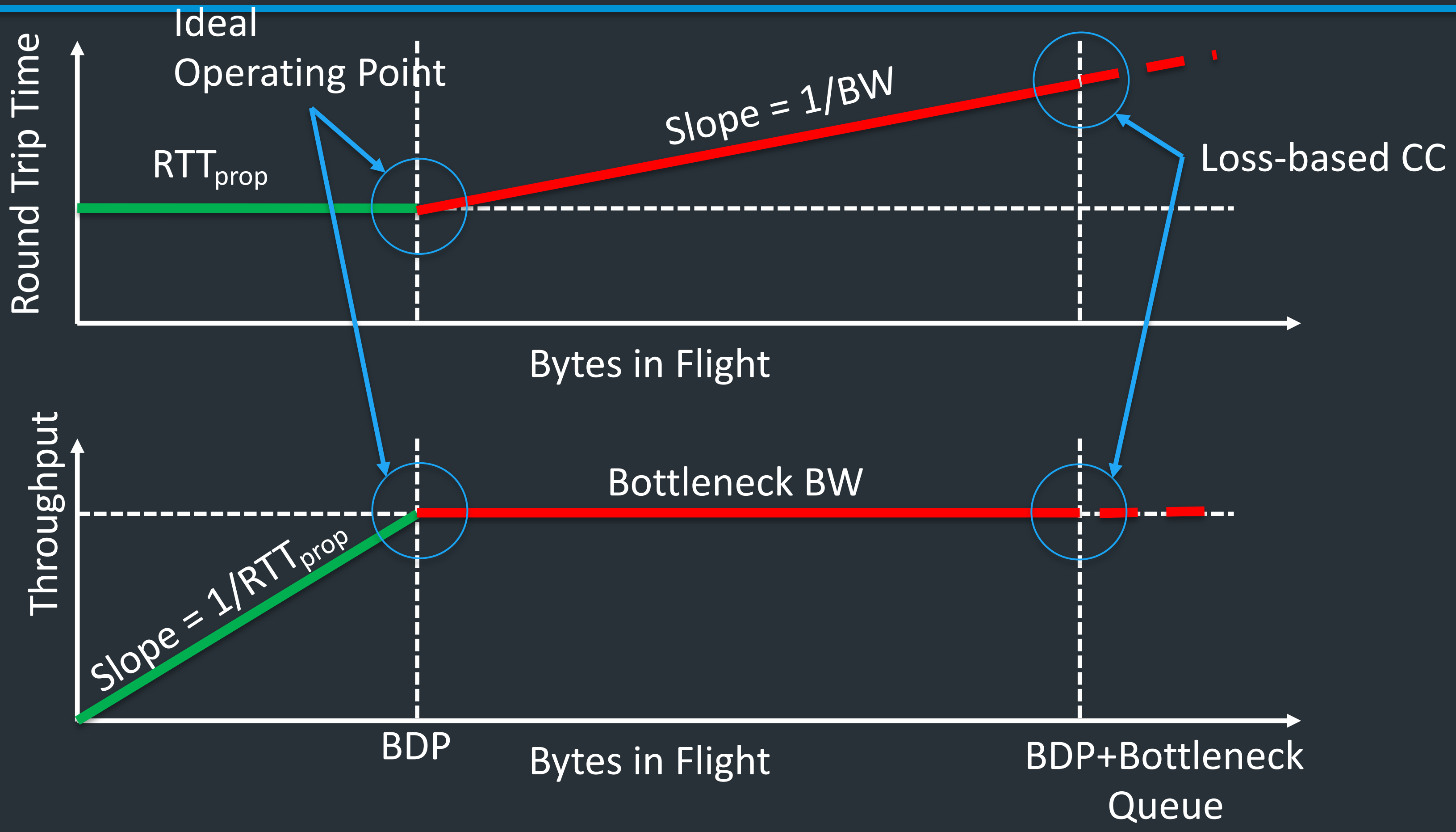




Another view of Congestion Control



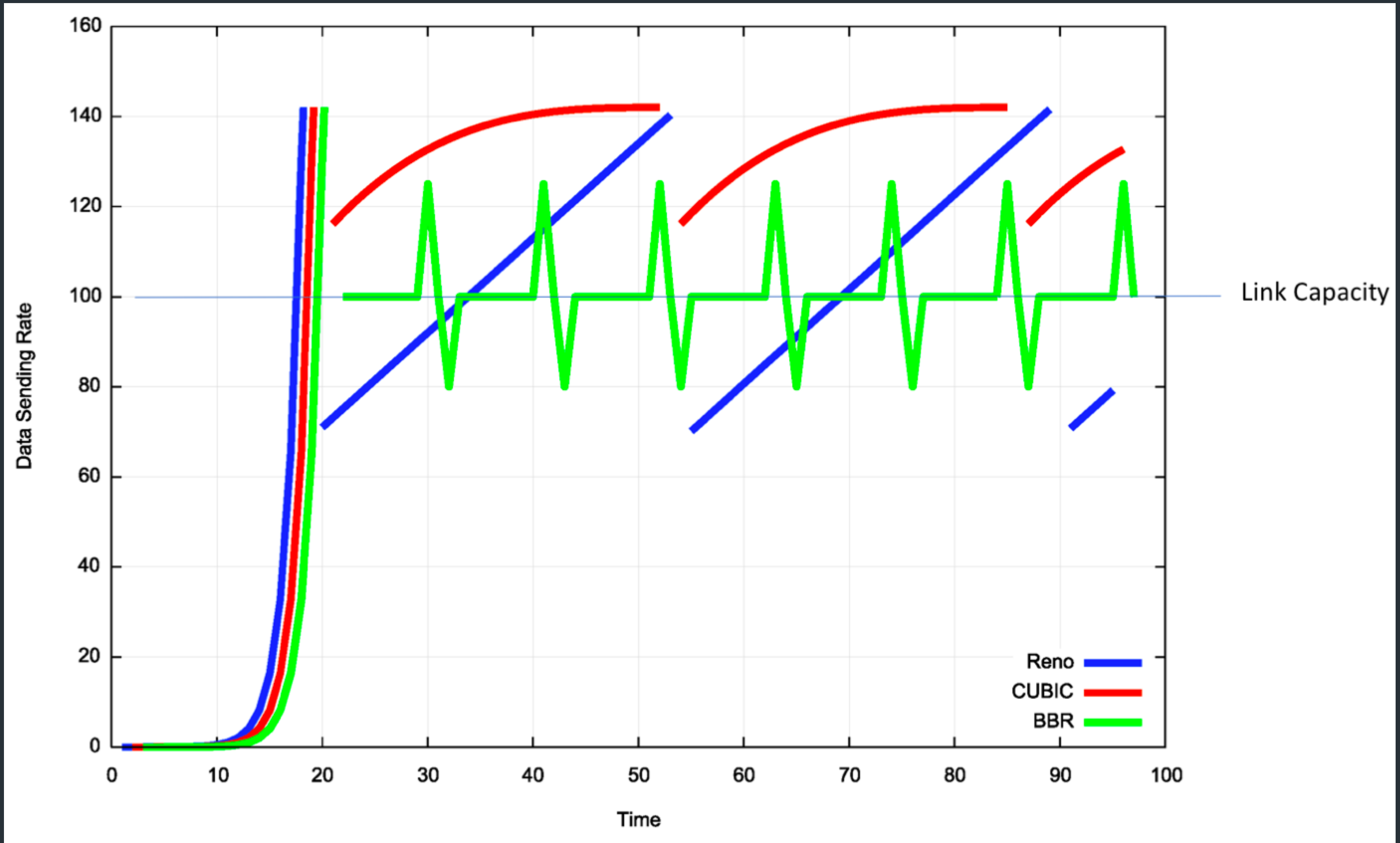
Another view of Congestion Control



BBR

- Problem: can’t measure both RTT_{prop} and Bottleneck BW at the same time
- BBR:
  - Slow start
  - Measure throughput when RTT starts to increase
  - Measure RTT when throughput is still increasing
  - Pace packets at the BDP
  - Probe by sending faster for 1RTT, then slower to compensate

BBR



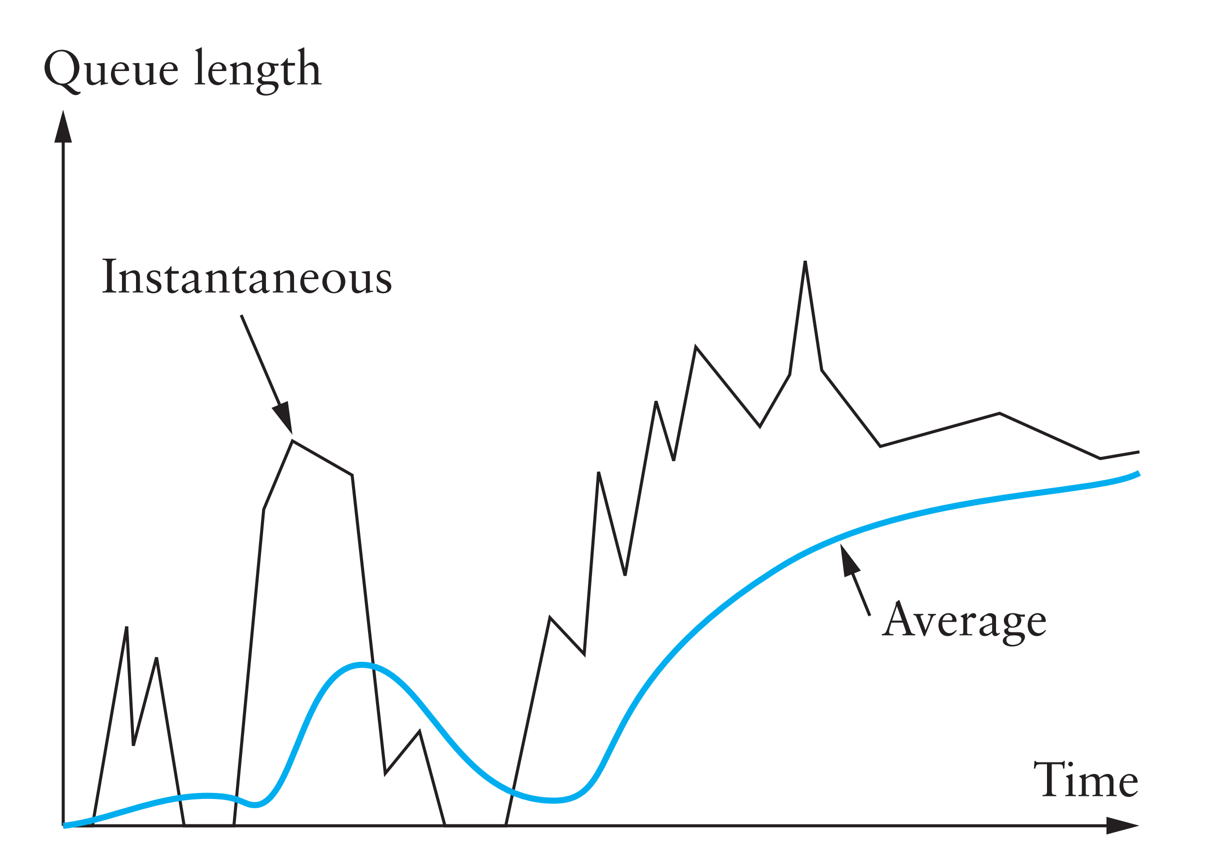
From: https://labs.ripe.net/Members/gih/bbr-tcp

Help from the network

- What if routers could *tell* TCP that congestion is happening?
  - Congestion causes queues to grow: rate mismatch
- TCP responds to drops
- Idea: Random Early Drop (RED)
  - Rather than wait for queue to become full, drop packet with some probability that increases with queue length
  - TCP will react by reducing cwnd
  - Could also mark instead of dropping: ECN

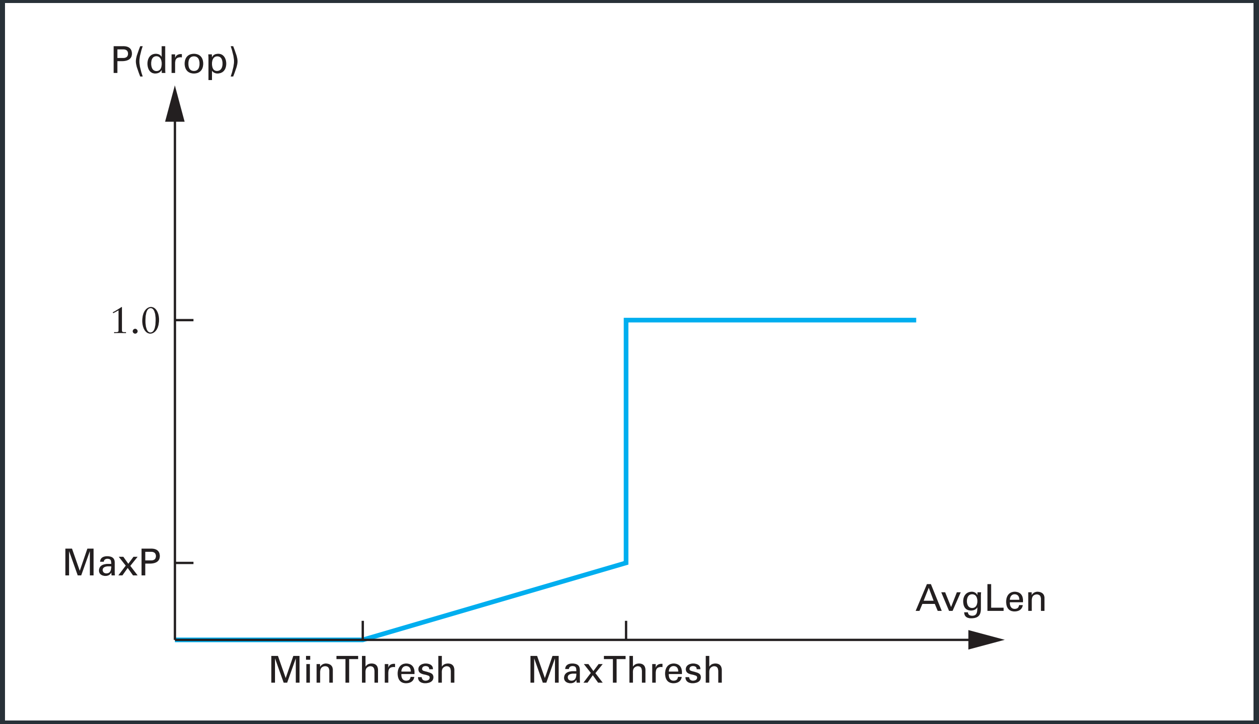
RED Details

- Compute average queue length (EWMA)
  - Don’t want to react to very quick fluctuations



RED Drop Probability

- Define two thresholds: MinThresh, MaxThresh
- Drop probability:



- **Improvements to spread drops (see book)**

RED Advantages

- Probability of dropping a packet of a particular flow is roughly proportional to the share of the bandwidth that flow is currently getting
- Higher network utilization with low delays
- Average queue length small, but can absorb bursts
- ECN
  - Similar to RED, but router sets bit in the packet
  - Must be supported by both ends
  - Avoids retransmissions optionally dropped packets

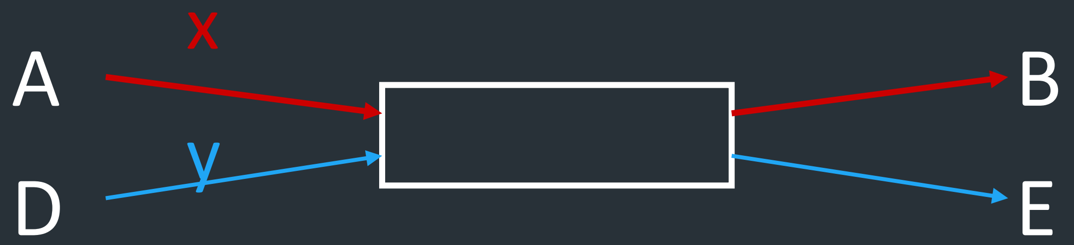
What happens if not everyone cooperates?

- TCP works extremely well when its assumptions are valid
  - All flows correctly implement congestion control
  - Losses are due to congestion

Cheating TCP

- Possible ways to cheat
  - Increasing cwnd faster
  - Large initial cwnd
  - Opening many connections
  - Ack Division Attack

Larger Initial Window



x starts SS with cwnd = 4

y starts SS with cwnd = 1

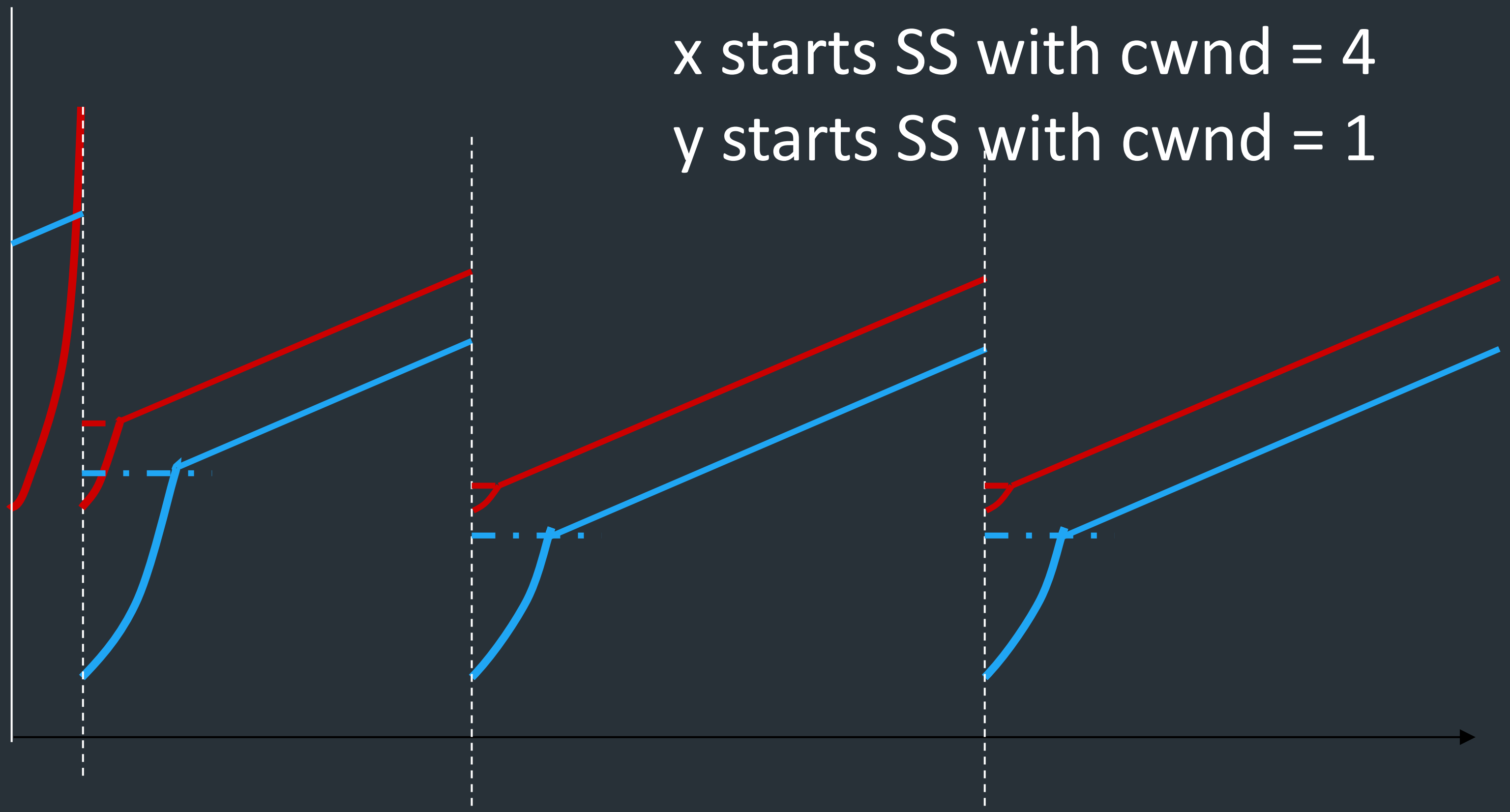
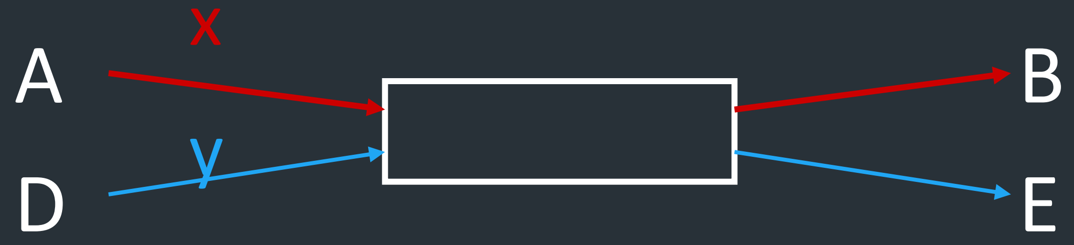


Figure from Walrand, Berkeley EECS 122, 2003

Open Many Connections

- Web Browser: has to download k objects for a page
  - Open many connections or download sequentially?



Assume:

- A opens 10 connections to B
- B opens 1 connection to E

- TCP is fair among connections
  - A gets 10 times more bandwidth than B

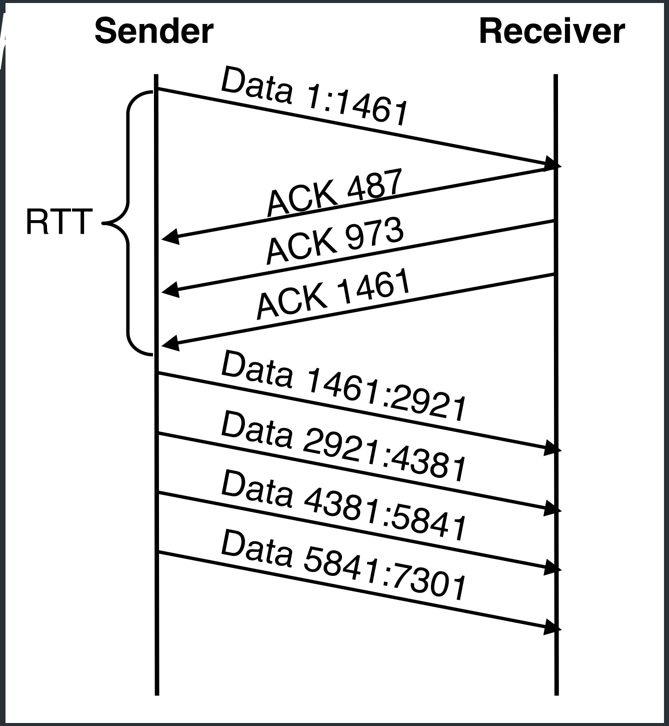
Figure from Walrand, Berkeley EECS 122, 2003

Exploiting Implicit Assumptions

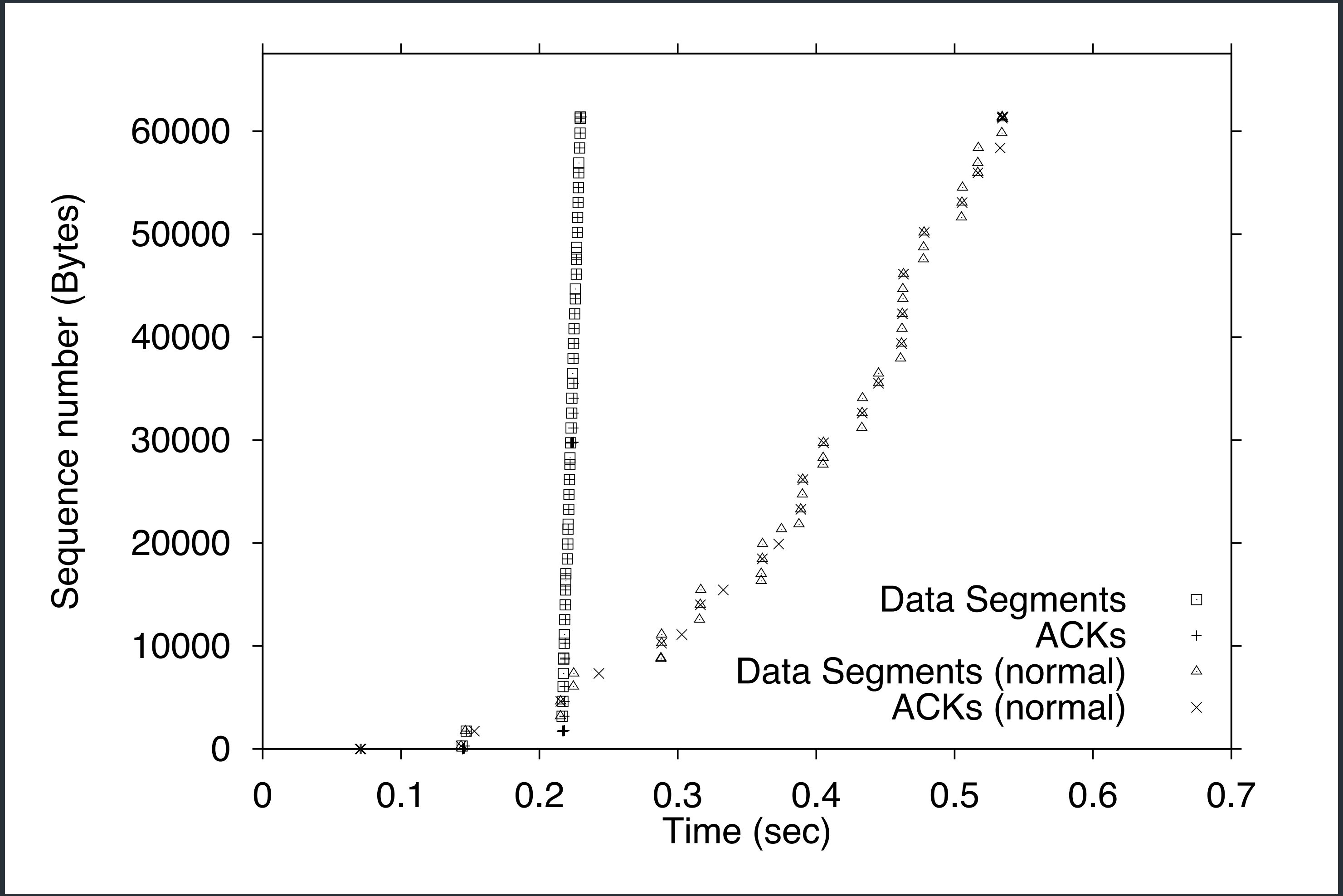
- Savage, et al., CCR 1999:
  - "TCP Congestion Control with a Misbehaving Receiver"
- Exploits ambiguity in meaning of ACK
  - ACKs can specify any byte range for error control
  - Congestion control assumes ACKs cover entire sent segments
- What if you send multiple ACKs per segment?

ACK Division Attack

- Receiver: *"upon receiving a segment with N bytes, divide the bytes in M groups and acknowledge eac*
- Sender will grow window M times faster
- Could cause growth to 4GB in 4 RTTs!
  - M = N = 1460



TCP Daytona!


Sequence number (Bytes)
60000
50000
40000
30000
20000
10000
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
Time (sec)
Data Segments □
ACKs +
Data Segments (normal) △
ACKs (normal) ×

Defense

- Appropriate Byte Counting
  - [RFC3465 (2003), RFC 5681 (2009)]
  - In slow start, cwnd += min (N, MSS)

  where N is the number of newly acknowledged bytes in the received ACK

More help from the network

- Problem: still vulnerable to malicious flows!
  - RED will drop packets from large flows preferentially, but they don’t have to respond appropriately
- Idea: Multiple Queues (one per flow)
  - Serve queues in Round-Robin
  - Nagle (1987)
  - Good: protects against misbehaving flows
  - Disadvantage?
  - Flows with larger packets get higher bandwidth

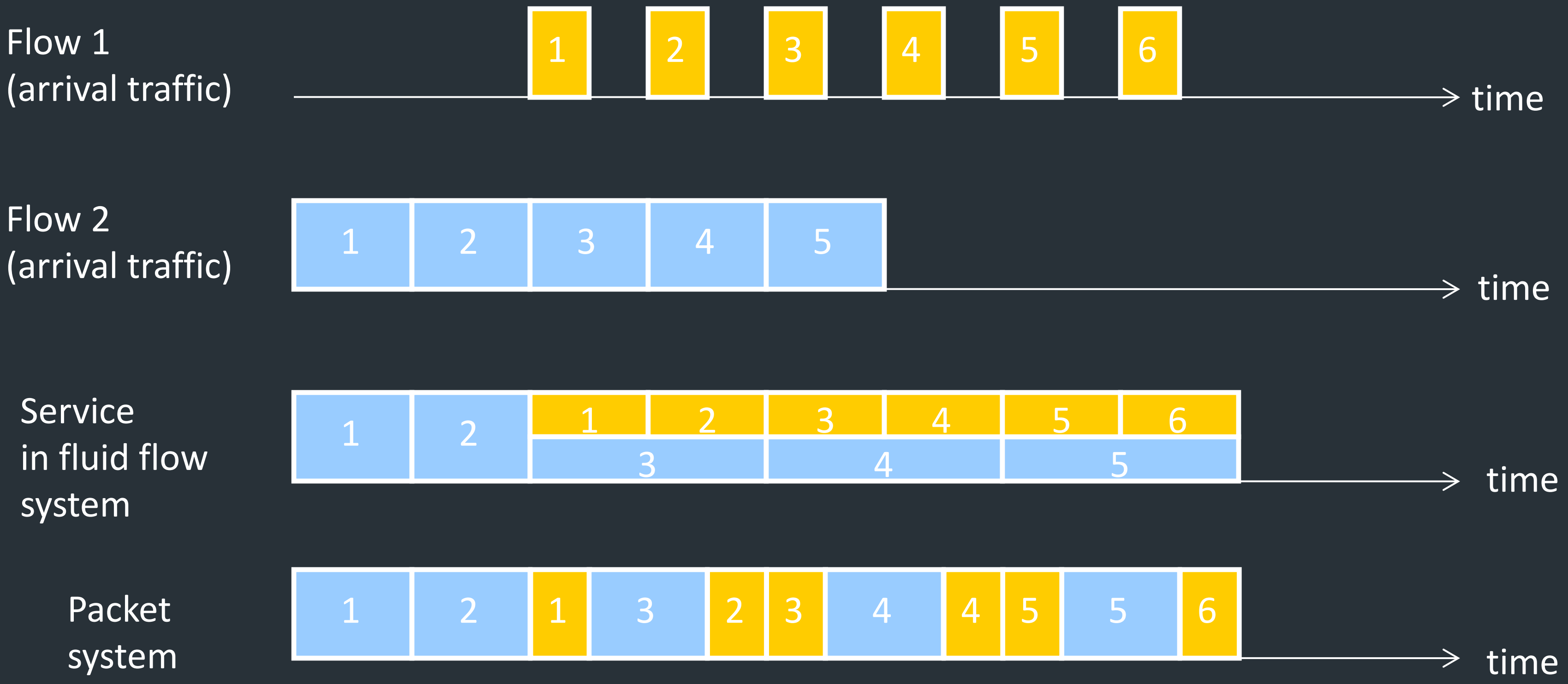
Solution

- Bit-by-bit round robing
- Can we do this?
  - No, packets cannot be preempted!
- We can only approximate it…

Fair Queueing

- Define a *fluid flow* system as one where flows are served bit-by-bit
- Simulate *ff*, and serve packets in the order in which they would finish in the *ff* system
- Each flow will receive exactly its fair share

Example


Flow 1
(arrival traffic)
1
2
3
4
5
6
time
Flow 2
(arrival traffic)
1
2
3
4
5
time
Service
in fluid flow
system
1
2
1
2
3
4
5
6
3
4
5
time
Packet
system
1
2
1
3
2
3
4
4
5
5
6
time

Implementing FQ

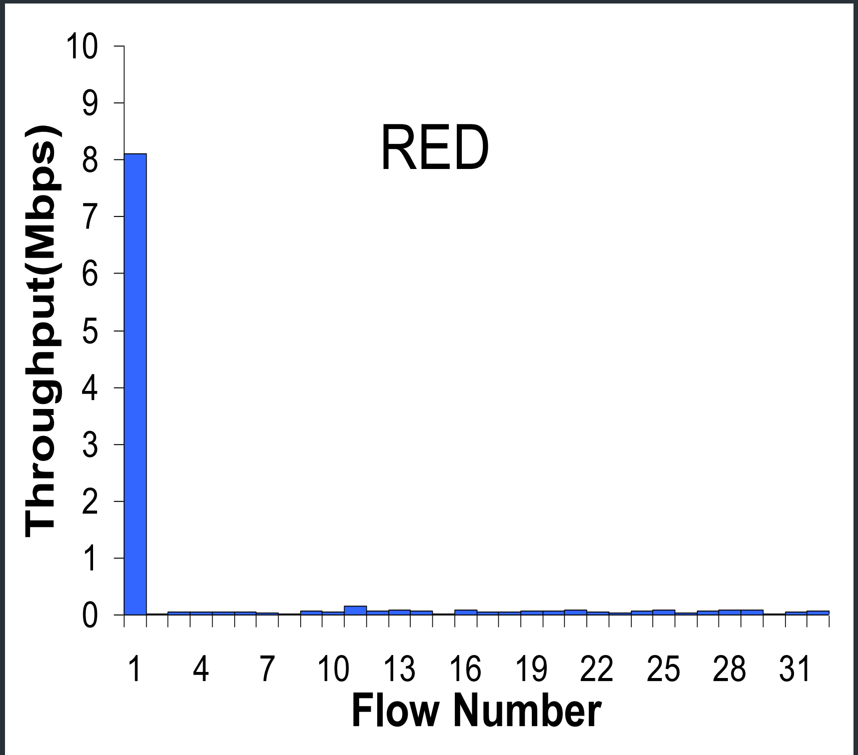
- Suppose clock ticks with each bit transmitted
  - (RR, among all active flows)
- P_i is the length of the packet
- S_i is packet i's start of transmission time
- F_i is packet i's end of transmission time
- $F_i = S_i + P_i$
- When does router start transmitting packet i?
  - If arrived before F_{i-1}, $S_i = F_{i-1}$
  - If no current packet for this flow, start when packet arrives (call this A_i): $S_i = A_i$
- Thus, $F_i = \max(F_{i-1}, A_i) + P_i$

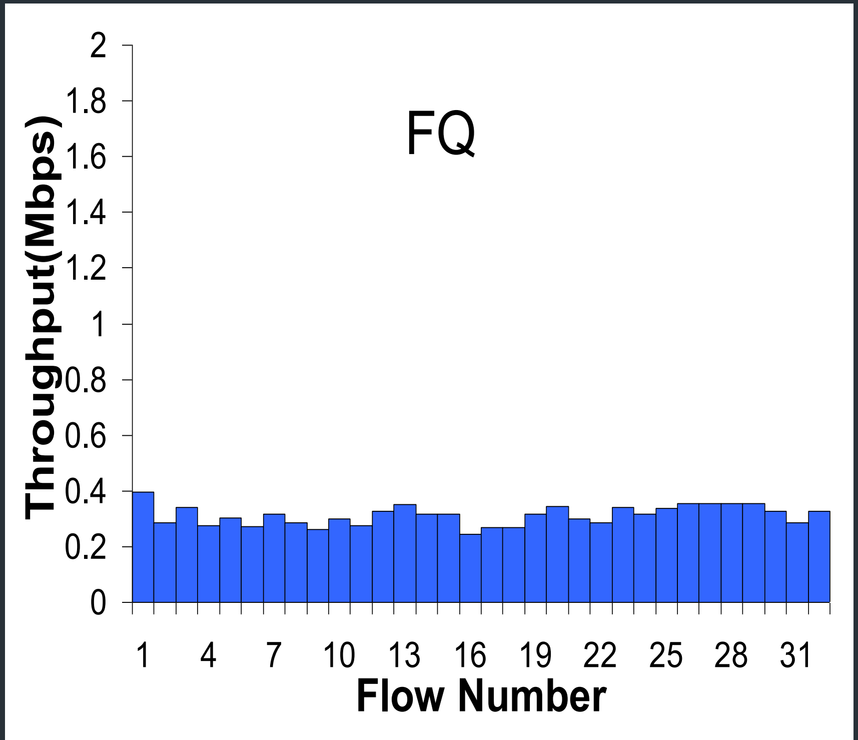
Fair Queueing

- Across all flows
  - Calculate F_i for each packet that arrives on each flow
  - Next packet to transmit is that with the lowest F_i
  - Clock rate depends on the number of flows
- Advantages
  - Achieves max-min fairness, independent of sources
  - Work conserving
- Disadvantages
  - Requires non-trivial support from routers
  - Requires reliable identification of flows
  - Not perfect: can’t preempt packets

Fair Queueing Example

- 10Mbps link, 1 10Mbps UDP, 31 TCPs





Big Picture

- Fair Queuing doesn’t eliminate congestion: just manages it
- You need both, ideally:
  - End-host congestion control to adapt
  - Router congestion control to provide isolation

Congestion control: motivation

The story so far

- Flow control: reliable, in-order delivery
- Goal: send as much data as receiver can handle
  - Receiver's advertised window: sent with every ACK
- Sliding window: increase throughput by having multiple packets in flight

Summary: flow control

- Flow control provides *correctness: reliable, in order delivery*
- Need more for performance
  - What if the network is the bottleneck?
- Sending too fast will cause queue overflows, heavy packet loss
- Need more for performance: congestion control

A Short History of TCP

- 1974: 3-way handshake
- 1978: IP and TCP split
- 1983: January 1st, ARPAnet switches to TCP/IP
- 1984: Nagle predicts congestion collapses
- 1986: Internet begins to suffer congestion collapses
  - LBL to Berkeley drops from 32Kbps to 40bps
- 1987/8: Van Jacobson fixes TCP, publishes seminal paper*: (TCP Tahoe)
- 1990: Fast transmit and fast recovery added (TCP Reno)

* Van Jacobson and Michael Karels. Congestion avoidance and control. SIGCOMM ’88

Congestion Collapse

Nagle, rfc896, 1984

- Mid 1980's: Problem with the protocol *implementations*, not the protocol!
- What was happening?
- If close to capacity, and, e.g., a large flow arrives suddenly…
  - RTT estimates become too short
  - Lots of retransmissions → increase in queue size
  - Eventually many drops happen (full queues)
  - Fraction of useful packets (not copies) decreases

The problem

- https://witestlab.poly.edu/respond/sites/genitutorial/files/tcp-aimd.ogv

TCP Congestion Control

- 3 Key Challenges
  - Determining the available capacity in the first place
  - Adjusting to changes in the available capacity
  - Sharing capacity between flows

- Idea
  - Each source determines network capacity for itself
  - Rate is determined by window size
  - Uses implicit feedback (drops, delay)
  - ACKs pace transmission (self-clocking)

Congestion control has a long history

- Active research area for ~40 years

- I am nowhere close to being an expert

- My hope is to get you to understand the problems involved

Just a few TCP implementations

What’s the difference?

General usage

- Reno (1980s)
- Tahoe
- Vegas
- New Vegas
- Westwood
- Cubic
- BBR (2016)
- …

Dealing with Congestion

- Maintain two windows:
  - Advertised Window (from receiver)
  - Congestion window (cwnd)

Sending rate = min(Advertised Window, cwnd)

- Ideally, want to have sending rate: ~= Window/RTT

Dealing with Congestion

- Assume losses are due to congestion
- After a loss, reduce congestion window
  - How much to reduce?
- Idea: conservation of packets at equilibrium
  - Want to keep roughly the same number of packets in network
  - Analogy with water in fixed-size pipe
  - Put new packet into network when one exits

Next time

- TCP Tahoe/Reno
- Overview of other CC schemes