CSCI-1680

Congestion Control Mechanics

Nick DeMarinis

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

Administrivia

- TCP Milestone I: Sign up for a meeting this week, if you haven't already!
- TCP gearup II TONIGHT (11/2) 5-7pm in CIT68 (+Zoom, +Recorded)
  - Any questions you have
  - Stuff for milestone II
  - How to test
- HW3: Out now, due next Wed => practice for milestone II

Warmup

Which of the following contribute to congestion:

a. Packets queueing up at switches

b. High CPU usage on the receiver ⇒ FLOW CONTROL PROBLEMS BUT NOT NETWORK CONGESTION

c. Many TCP connections on the same link

d. Many UDP connections on the same link

e. Poor wifi connection on the sender

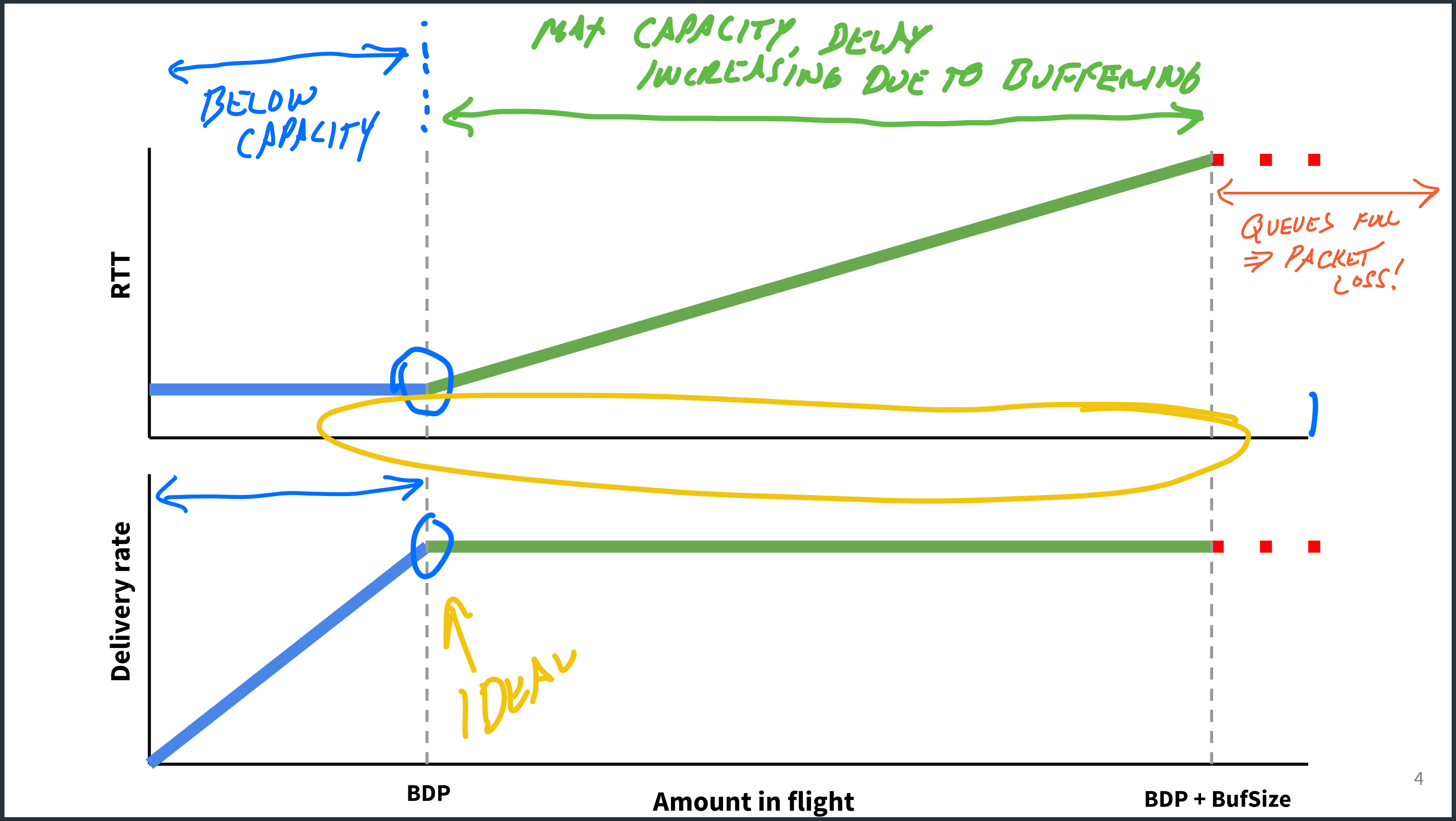
Flow control: making sure we don't overwhelm the receiver

→ ADVERTISED WINDOW (WIN)

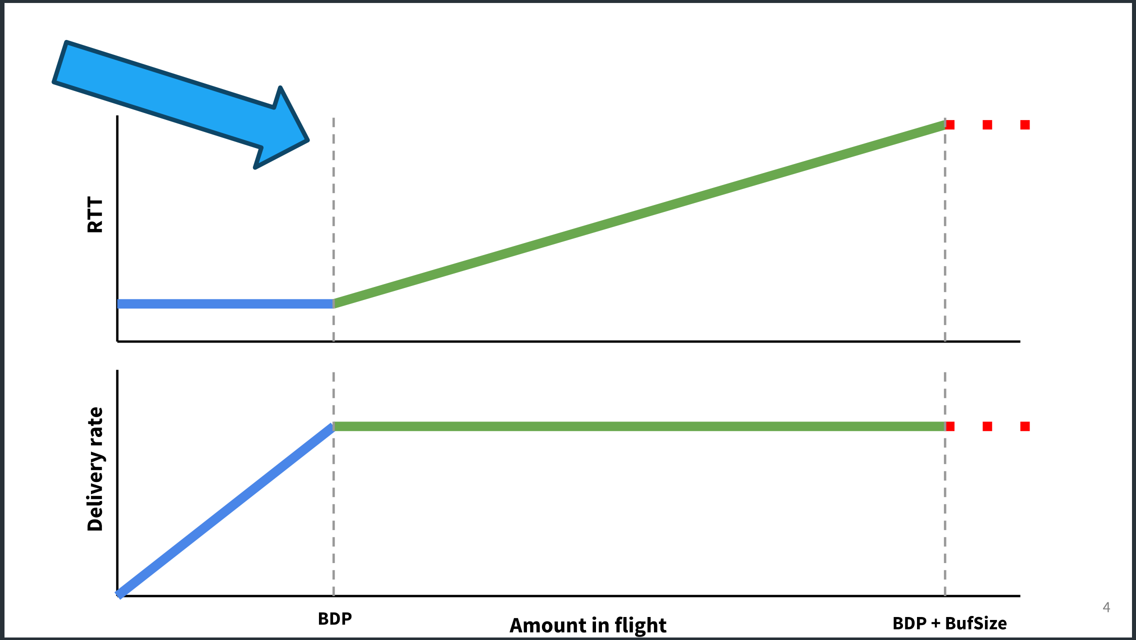
RECEIVING TCP STACK / RECEIVING APP

Congestion control: making sure we don't overwhelm the network

Thinking about congestion



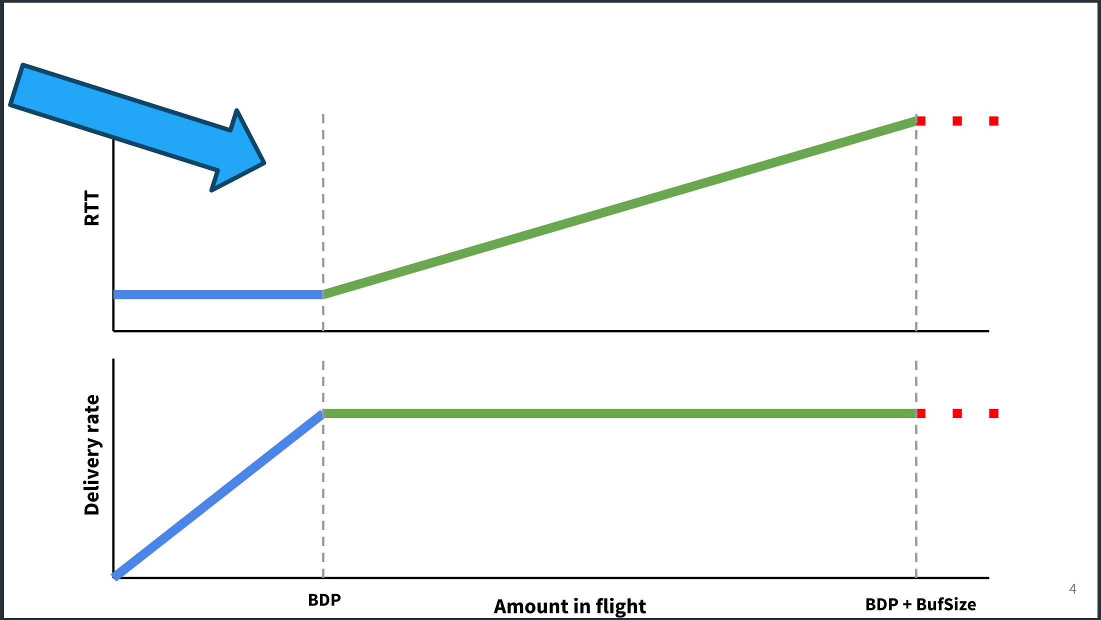
"BBR congestion control"



"BBR congestion control"

<u>Bandwidth-delay product (BDP)</u>: maximum amount of data that can be in-transit on a network link at any given time

(Link capacity (bits/sec)) * (RTT (sec))
= (bytes)

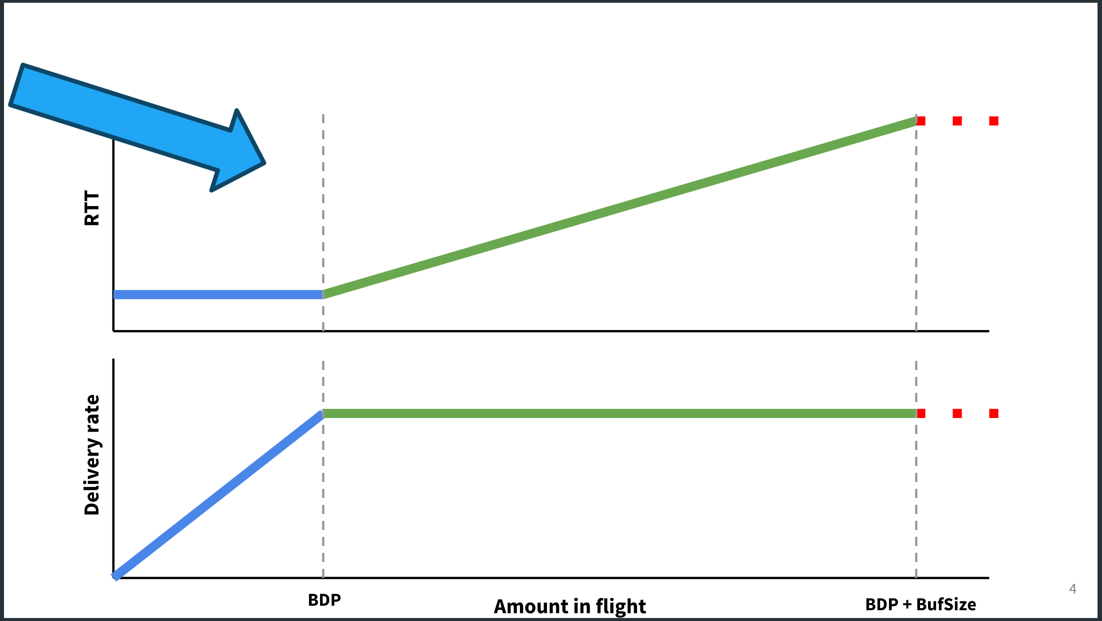


"BBR congestion control"

Bandwidth-delay product (BDP): **maximum amount of data that can be in-transit on a network link at any given time**

(Link capacity (bits/sec)) * (RTT (sec))
= (bytes)

Eg. 1Gbps link * 1ms RTT = 125KiB BDP



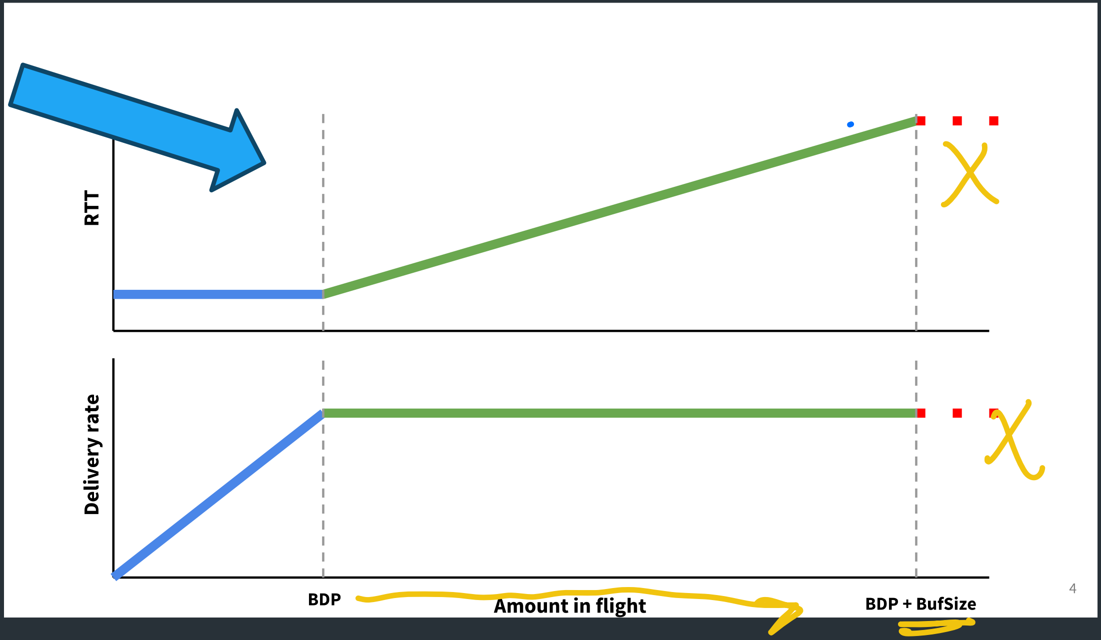
["BBR congestion control"](#)

Bandwidth-delay product (BDP): maximum amount of data that can be in-transit on a network link at any given time

(Link capacity (bits/sec)) * (RTT (sec))

= (bytes)

Eg. 1Gbps link * 1ms RTT = 125KiB BDP



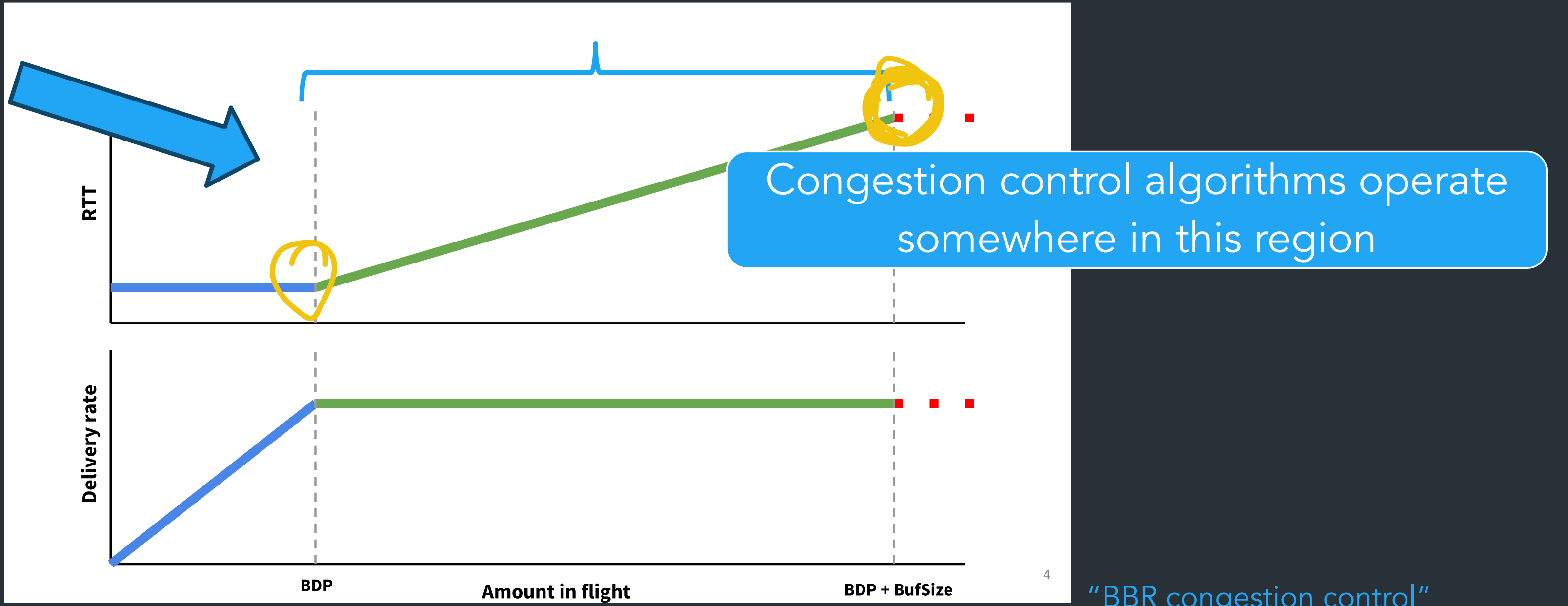
["BBR congestion control"](#)

Bandwidth-delay product (BDP): **maximum amount of data that can be in-transit on a network link at any given time**

(Link capacity (bits/sec)) * (RTT (sec))
= (bytes)

Eg. 1Gbps link * 1ms RTT = 125KiB BDP

=> After exceeding BDP, network is queueing packets. After queues are full, packets getting dropped due to congestion.



"BBR congestion control"

<u>Bandwidth-delay product (BDP)</u>: maximum amount of data that can be in-transit on a network link at any given time

(Link capacity (bits/sec)) * (RTT (sec))
= (bytes)

Eg. 1Gbps link * 1ms RTT = 125KiB BDP

=> After exceeding BDP, network is queueing packets. After queues are full, packets getting dropped due to congestion.

Why is this hard?

Sender doesn't know the network capacity

- The network can't (generally) tell us this

… and the network may change

- New connections start up
- Connections end
- Link characteristics may change…

– NEED TO DO A START

– CONTINUALLY WHILE SENDING.

Why is this hard?

Sender doesn't know the network capacity

- The network can't (generally) tell us this

… and the network may change

- New connections start up
- Connections end
- Link characteristics may change…

=> Need to measure or model what is going on in the network as we are sending, adapt accordingly

The basic principle

Signals from the network

(ACKs, other TCP packet info, more...)

↓

Congestion Control (CC) algorithm

↓

Congestion window: cwnd

Lots of CC variants designed with different strategies and goals

Network Signals

- Packet loss ("loss-based") ← CLASSIC MODELS
- Delay/RTT ("delay-based")
- "Marks" added on packets by routers

Goals

- Maximize throughput
- Recover from packet loss or high RTT
- Short-long "flows"
- Datacenter-specific (low-latency)

Lots of CC variants designed with different strategies and goals

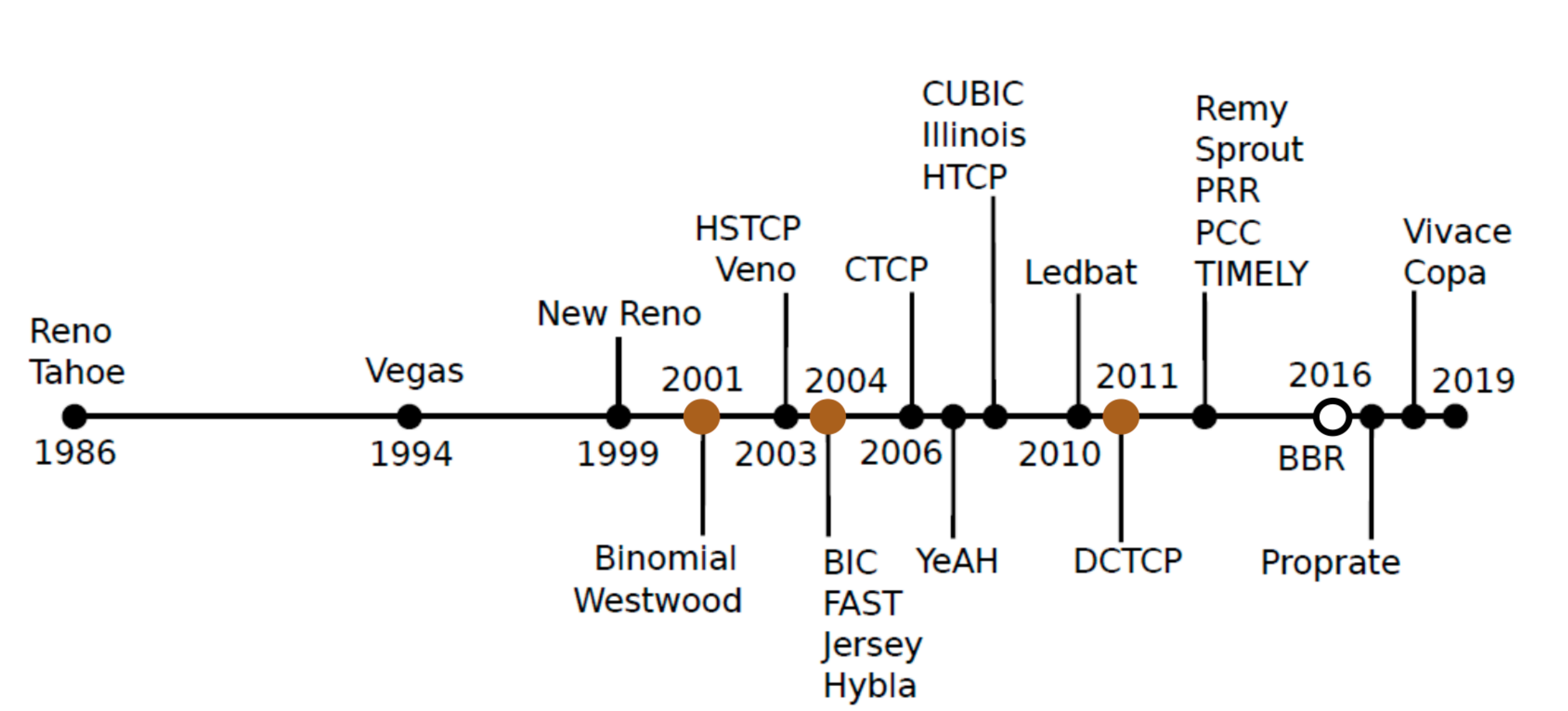
Network Signals

- Packet loss (“loss-based”)
- Delay/RTT (“delay-based”)
- ”Marks” added on packets by routers

Goals

- Maximize throughput
- Recover from packet loss or high RTT
- Short-long “flows”
- Datacenter-specific (low-latency)

⇒ This is a big research area!


Reno
Tahoe
1986
Vegas
1994
New Reno
1999
2001
Binomial
Westwood
HSTCP
Veno
2003
2004
BIC
FAST
Jersey
Hybla
CTCP
2006
YeAH
CUBIC
Illinois
HTCP
Ledbat
2010
2011
DCTCP
Remy
Sprout
PRR
PCC
TIMELY
2016
BBR
Proprate
Vivace
Copa
2019

SIGNALS

Variant	Feedback	Required changes	Benefits	Fairness
(New) Reno	Loss	—	—	Delay
Vegas	Delay	Sender	Less loss	Proportional
High Speed	Loss	Sender	High bandwidth	
BIC	Loss	Sender	High bandwidth	
CUBIC	Loss	Sender	High bandwidth	
C2TCP[11][12]	Loss/Delay	Sender	Ultra-low latency and high bandwidth	
NATCP[13]	Multi-bit signal	Sender	Near Optimal Performance	
Elastic-TCP	Loss/Delay	Sender	High bandwidth/short & long-distance	
Agile-TCP	Loss	Sender	High bandwidth/short-distance	
H-TCP	Loss	Sender	High bandwidth	
FAST	Delay	Sender	High bandwidth	Proportional
Compound TCP	Loss/Delay	Sender	High bandwidth	Proportional
Westwood	Loss/Delay	Sender	Lossy links	
Jersey	Loss/Delay	Sender	Lossy links	
BBR[14]	Delay	Sender	BLVC, Bufferbloat	
CLAMP	Multi-bit signal	Receiver, Router	Variable-rate links	Max-min
TFRC	Loss	Sender, Receiver	No Retransmission	Minimum delay
XCP	Multi-bit signal	Sender, Receiver, Router	BLFC	Max-min
VCP	2-bit signal	Sender, Receiver, Router	BLF	Proportional
MaxNet	Multi-bit signal	Sender, Receiver, Router	BLFSC	Max-min
JetMax	Multi-bit signal	Sender, Receiver, Router	High bandwidth	Max-min
RED	Loss	Router	Reduced delay	
ECN	Single-bit signal	Sender, Receiver, Router	Reduced loss	

Wikipedia’s list

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

Classical Congestion Control

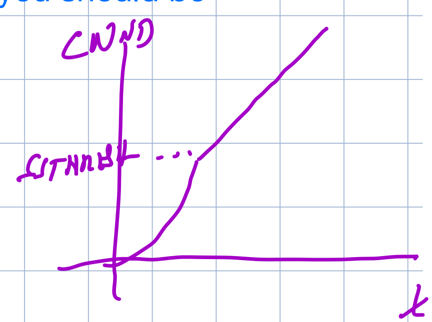
Loss-based: assume packet loss => congestion

- TCP Tahoe (1988)
  - Slow start, congestion avoidance, fast retransmit
- TCP Reno (1990)
  - TCP Tahoe + Fast recovery
- Many variations developed from this… (see optional readings)

Loss-based congestion control
- Slow start (SS): figure out initial capacity, determine capacity again after a packet oss
  - => After finishing a window, cwnd = cwnd * 2
  - => Do this until we observe a packet loss

After first loss, save ssthresh <= cwnd
If cwnd < ssthresh => slow start mode (window is so low you should be in startup phase

if cwnd > ssthresh: congestion-avoidance mode



- Congestion avoidance (CA): steady state process
  - => continually look for capacity to change, if so, increase how much you send. If you see a packet loss, scale back

Start with some initial window cwnd

After sending whole window
- If no packet loss observed cwnd += MSS
  Also written as. Cwnd += 1. => increase by one segment

==> If nothing went wrong, try to increase capacity

- If packets were lost (timeout occurred):
cwnd = cwnd / 2

=> ADDITIVE INCREASE MULTIPLICATIVE DECREASE!

=> Reduce amount we can send in next window dramatically, gives network some time to recover

(AIMD)

Modes of operation

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

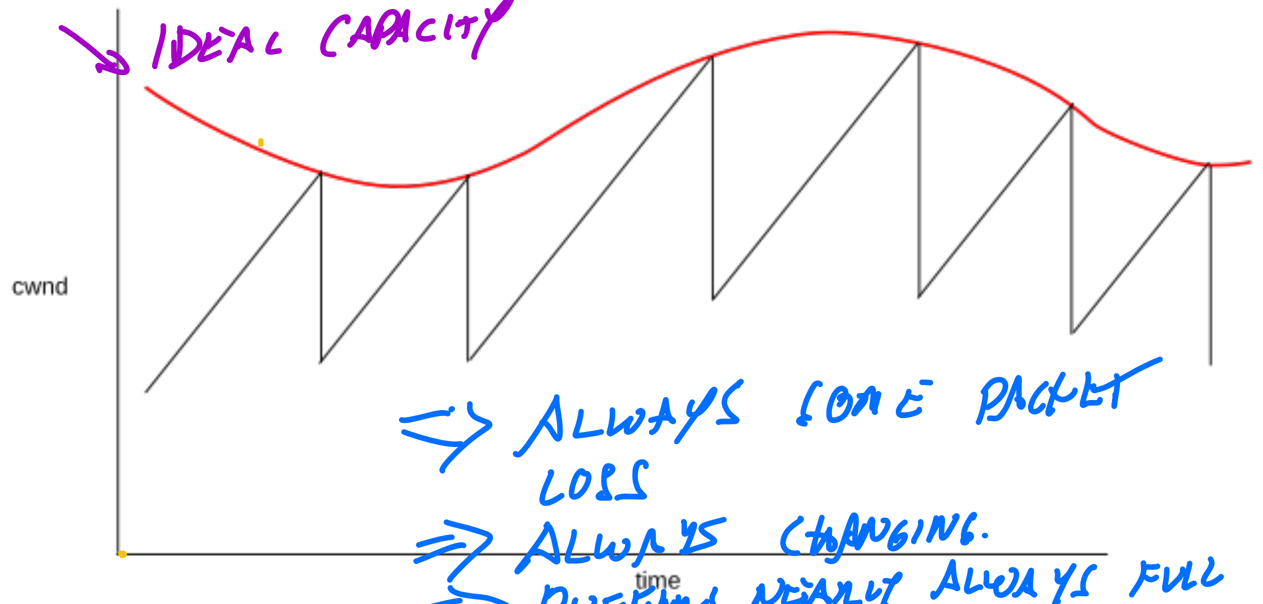
Congestion Avoidance

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 called 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

Turns out AIMD is really slow to start up. So do something faster at connection start…

INITIAL ~~STAT~~ CWND? 1?

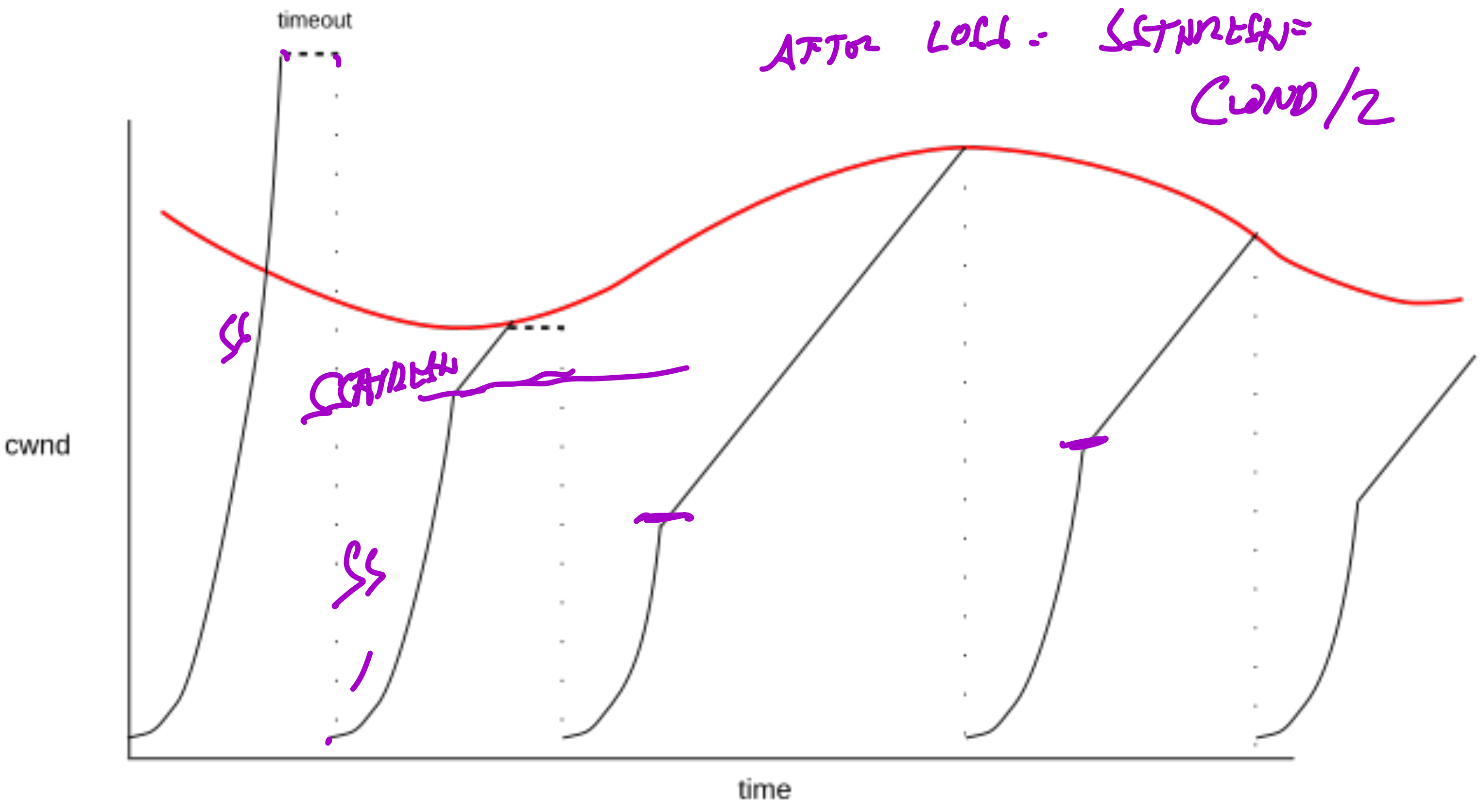
Slow Start

Turns out AIMD is really slow to start up. So do something faster at connection start…

After finishing a window

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

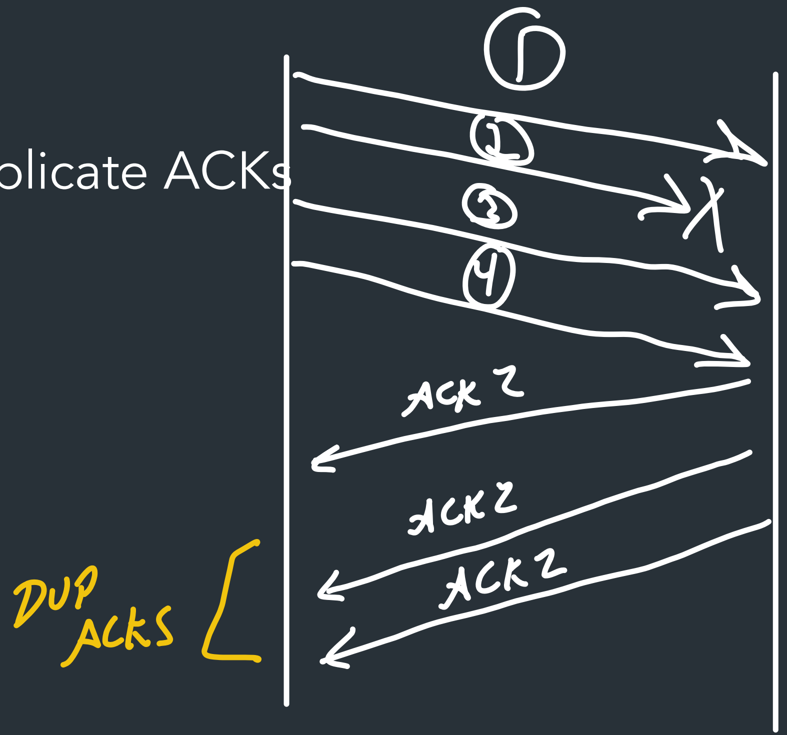
- After first loss, keep slow-start threshold (ssthresh):
  - If window < ssthresh: slow-start
  - If window > ssthresh: congestion avoidance
- After first loss: ssthresh = cwnd / 2



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 ⇒ REALLY LONG RTT
- Any other way?
  - Gap in sequence numbers at receiver
  - Receiver uses cumulative ACKs: drops => duplicate ACKs



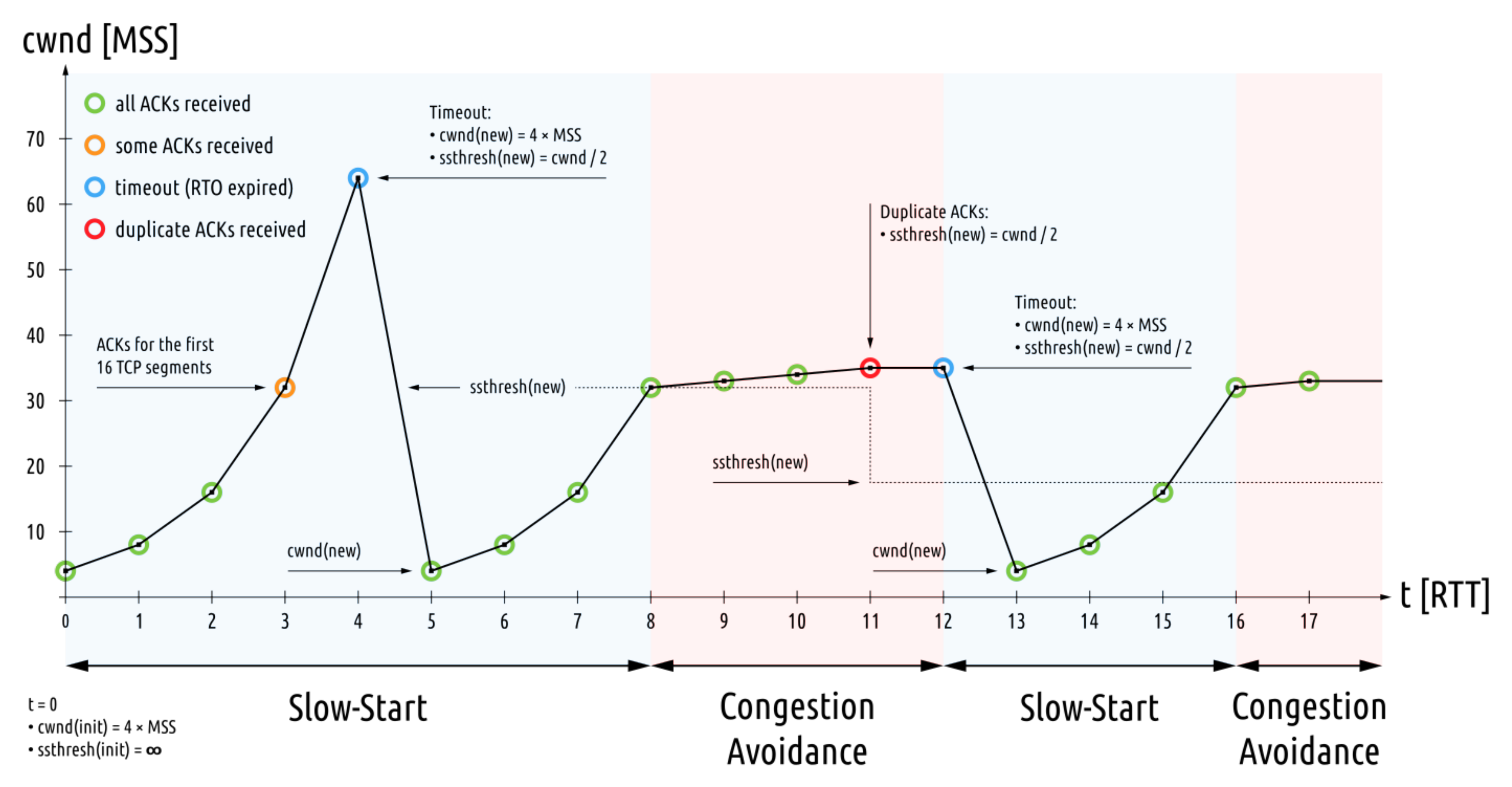
How to Detect Loss

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

How to Detect Loss

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

- Which one is worse?


cwnd [MSS]
all ACKs received
some ACKs received
timeout (RTO expired)
duplicate ACKs received
Timeout:
• cwnd(new) = 4 × MSS
• ssthresh(new) = cwnd / 2
Duplicate ACKs:
• ssthresh(new) = cwnd / 2
Timeout:
• cwnd(new) = 4 × MSS
• ssthresh(new) = cwnd / 2
ACKs for the first
16 TCP segments
ssthresh(new)
ssthresh(new)
cwnd(new)
cwnd(new)
70
60
50
40
30
20
10
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:

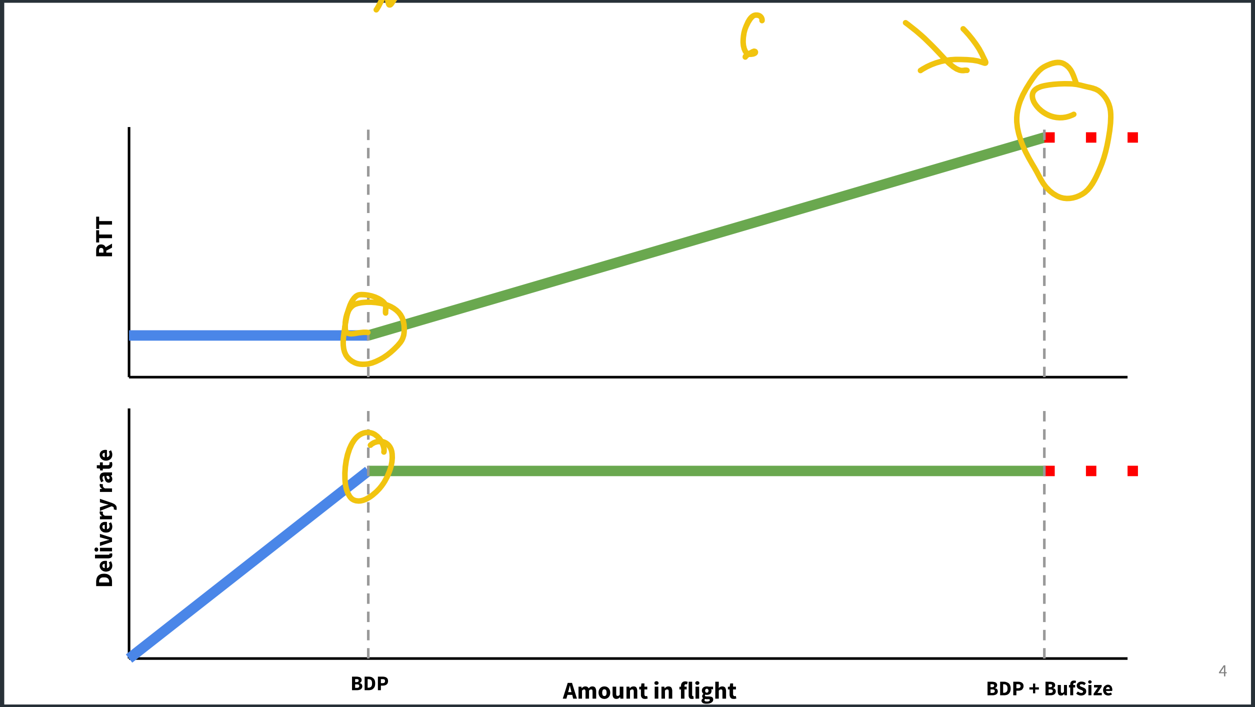
- Tahoe (1988)
- Reno (1990)

LOSS-BASED CC

- 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

WIN, MACOS.

BBR: what's different



Not based just based on packet loss

- Tries to measure both RTT and link capacity

Normal phase swaps between measuring link capacity (sending more) vs. measuring RTT (sends less, looks for RTT to go down)

=> Uses both of these things to figure out a sending rate

"BBR congestion control"

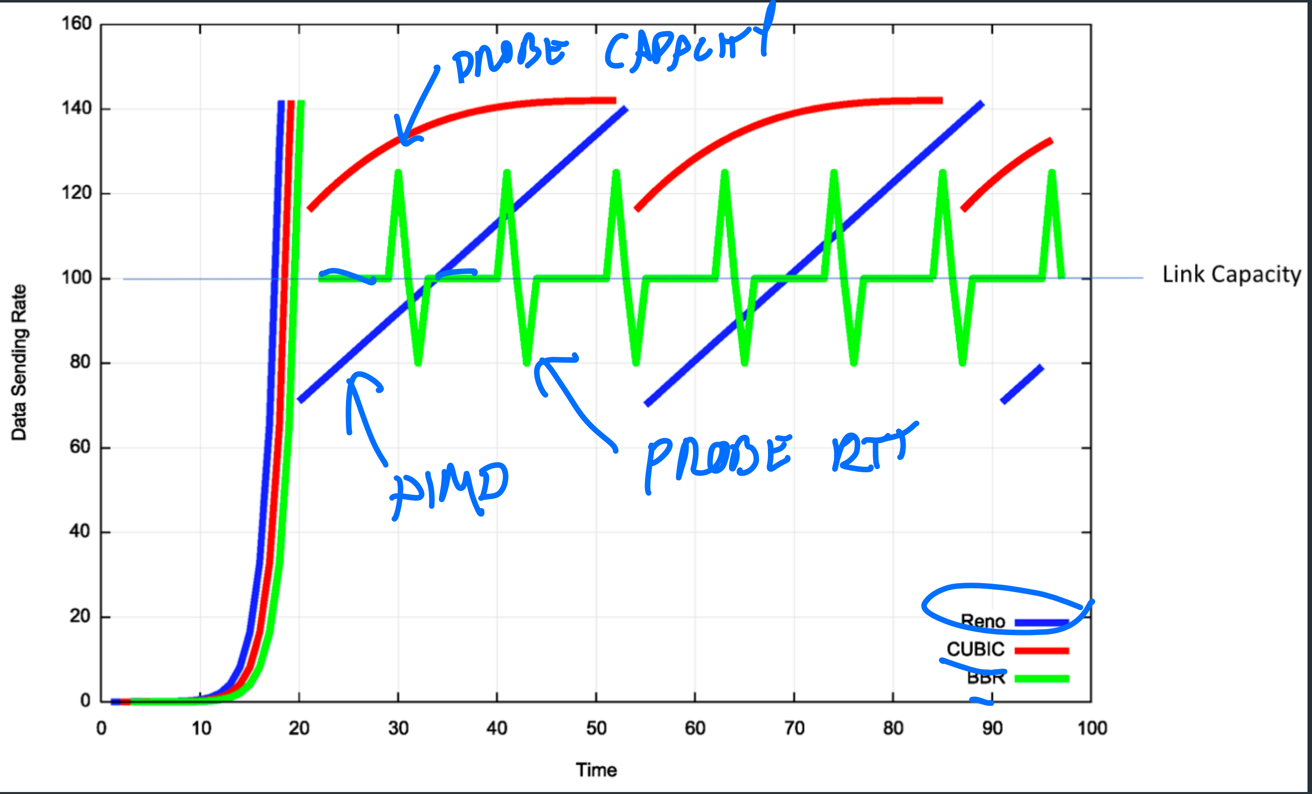
BBR

- 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 (2016)



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

Another way: ECN

EXTERNAL CONGESTION NOTIFICATION

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

→ NOTIFY BEFORE BUFFERS FILL UP.

ONLY WORKS IF ROUTERS COOPERATE (NOT ON INTERNET)

Special purpose example: DCTCP (2010)

IN DATACENTER < 200ms RTT

Designed for datacenter usage only

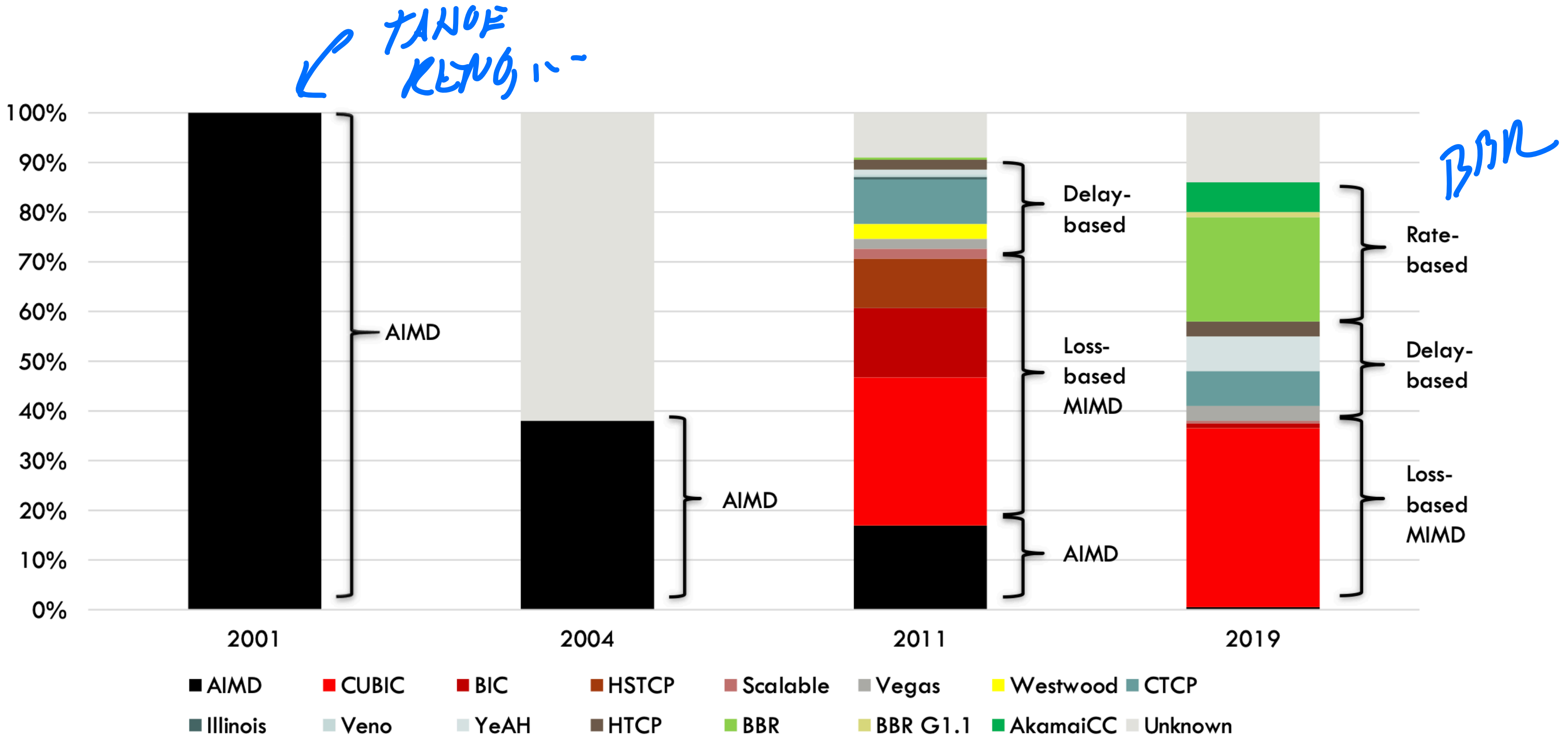
- Want to avoid queuing as much as possible
- Routers/switches mark packets with ECN bit in header
- When this happens, senders scale back dramatically

⇒ IN DATACENTER, PROVIDER CAN CONTROL WHOLE STACK

(SERVERS, OS, SWITCHES, ROUTERS, APPS...)

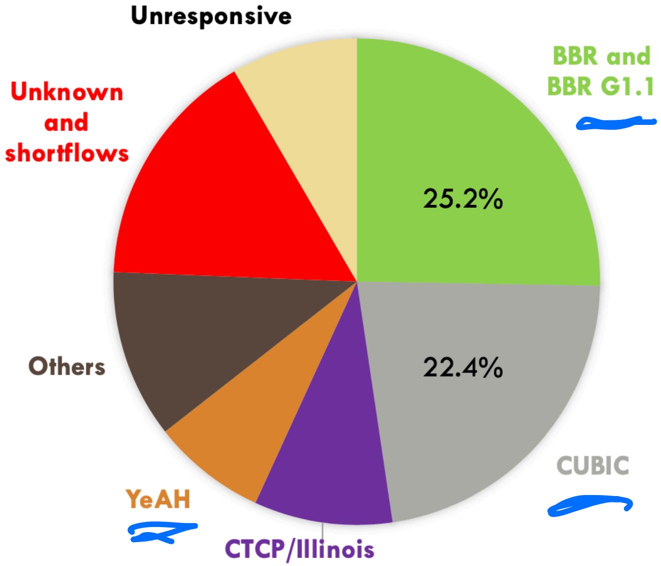
What happens in practice now?

THE EVOLUTION OF THE TCP ECOSYSTEM



"The Great Internet TCP congestion control census" (2019)

DISTRIBUTION BY **POPULARITY** AND **TRAFFIC SHARE**



Share of congestion control algorithms deployed by website count in the **Alexa Top 250** websites

- Among the top 250 Alexa websites, BBR has a larger share by website count than Cubic
- In terms of traffic share, BBR is now contributing to **more than 40%** of the downstream traffic on the Internet!

Site	Downstream traffic share	Variant
Amazon Prime	3.69%	CUBIC
Netflix	15%	CUBIC
YouTube	11.35%	BBR
Other Google sites	28%	BBR
Steam downloads	2.84%	BBR

(As measured on **static HTTP webpages**)

"The Great Internet TCP congestion control census" (2019)

LOOKING CLOSER AT THE UNCLASSIFIED VARIANTS

We had a total of **6,330 (31.65%)** of websites that were **unclassified**

We ran a variety of network profiles on these websites to infer something about their congestion control mechanism

Type	React to Packet Loss?	React to BDP?	Websites (share)
AkamaiCC	✗	✓	1,103 (5.52%)
Unknown Akamai	✗	?	157 (0.79%)
Unknown	✗	?	493 (2.47%)
	✓	?	1,782 (8.91%)
Short flows	✓	?	1,493 (7.47%)
Unresponsive	?	?	1,302 (6.51%)
Total			6,330 (31.65%)

"The Great Internet TCP congestion control census" (2019)