EECS-340 Introduction to Computer Networking

Lecture 7: TCP Congestion Control

Steve Tarzia

Many diagrams & slides are adapted from those by J.F Kurose and K.W. Ross

Last Lecture: TCP

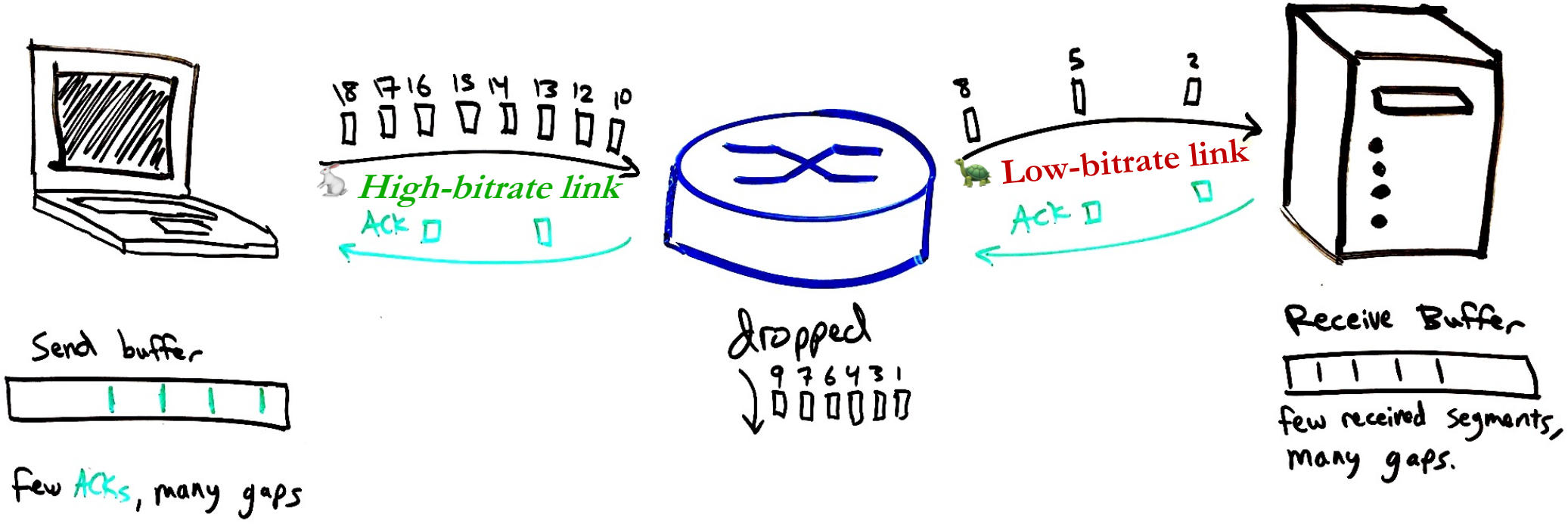
- Uses **cumulative ACKs**.
- Any data segment can carry an ACK.
- Receivers buffer out-of-order (early) segments for later reassembly.
- ACK timeout can be appropriately set with Exponentially-Weighted Moving Average (**EWMA**) of recent RTT and recent **jitter**.
- # in-flight packets (thus throughput) is determined by *window size*.
- TCP throughput should be regulated so as not to overwhelm:
  - the receiver -- **Flow control** is implemented with explicit Receive Window.
  - the network – **Congestion control** (today's topic).
- Connection setup requires a *3-way handshake*.
  - Sets initial sequence numbers and receive windows in both directions.
  - Teardown requires sending and acknowledging **FIN** messages.

Congestion Control

- Congestion is when the **network** is overloaded
  - Router queues are full, so packets are dropped
  - or length of queues leads to long nodal queuing delay, causing timer to expire.
- Dropped packets lead to *inefficiency* and can compound the problem:
  - Congestion → Packet loss → Retransmission → *More congestion!* → *More loss!*
- Goal is to prevent the self-destructive feedback cycle above.
- ***Better to wait*** than to send a packet likely to be dropped before reaching its destination.
- We have **end-to-end observations** of network performance, but the precise internal cause of a network problem is difficult to know.
  - State of routers along path is unknown – end hosts know only their state.

What would happen without congestion control?

- Problems arise even in a simple network with just one connection/flow:
- If links have different bitrates, and hosts send as fast as possible, packet loss will occur:

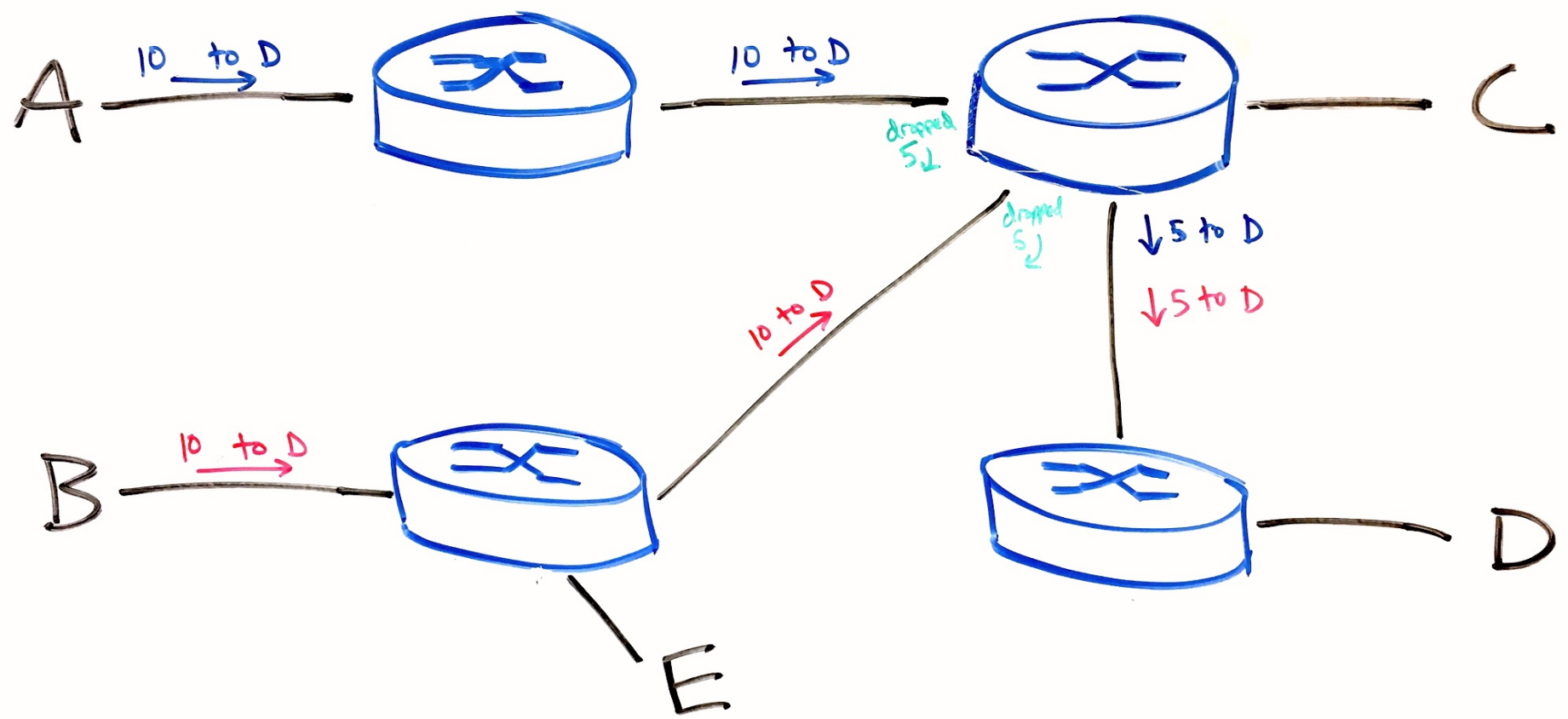


- Very large buffers would be needed, with many gaps for lost packets.
- Cumulative ACKs (as in TCP) would not be efficient:
- "Early" segments would be common, and these would all be retransmitted.

Shared congestion causes more inefficiency

It's wasteful to drop packets dropped halfway through their trip.

- Below, assume all links have a capacity of 10.
- E cannot communicate with C simply because B is wasting bandwidth on the middle diagonal link. Half of the B to D packets are dropped anyway.



Two general types of congestion control

End-to-end (TCP)

- No explicit feedback from routers.
- Congestion is inferred from observed packet loss.

Network assisted

- Routers signal congestion:
  - Sets a certain bits in TCP (or IP) header (called ECN)
  - List precise bitrate desired for sender.
- This has become popular within cloud/datacenter networks.
  - Requires routers to be properly configured and trusted.

Observing congestion

- Recall that flow control (receive window) prevents loss at receiver.
- If packets are dropped (timers expire before ACK), two possibilities:
  - Timer interval is too small (EWMA will adjust itself), or
  - The network is congested, and packets are being dropped by routers.

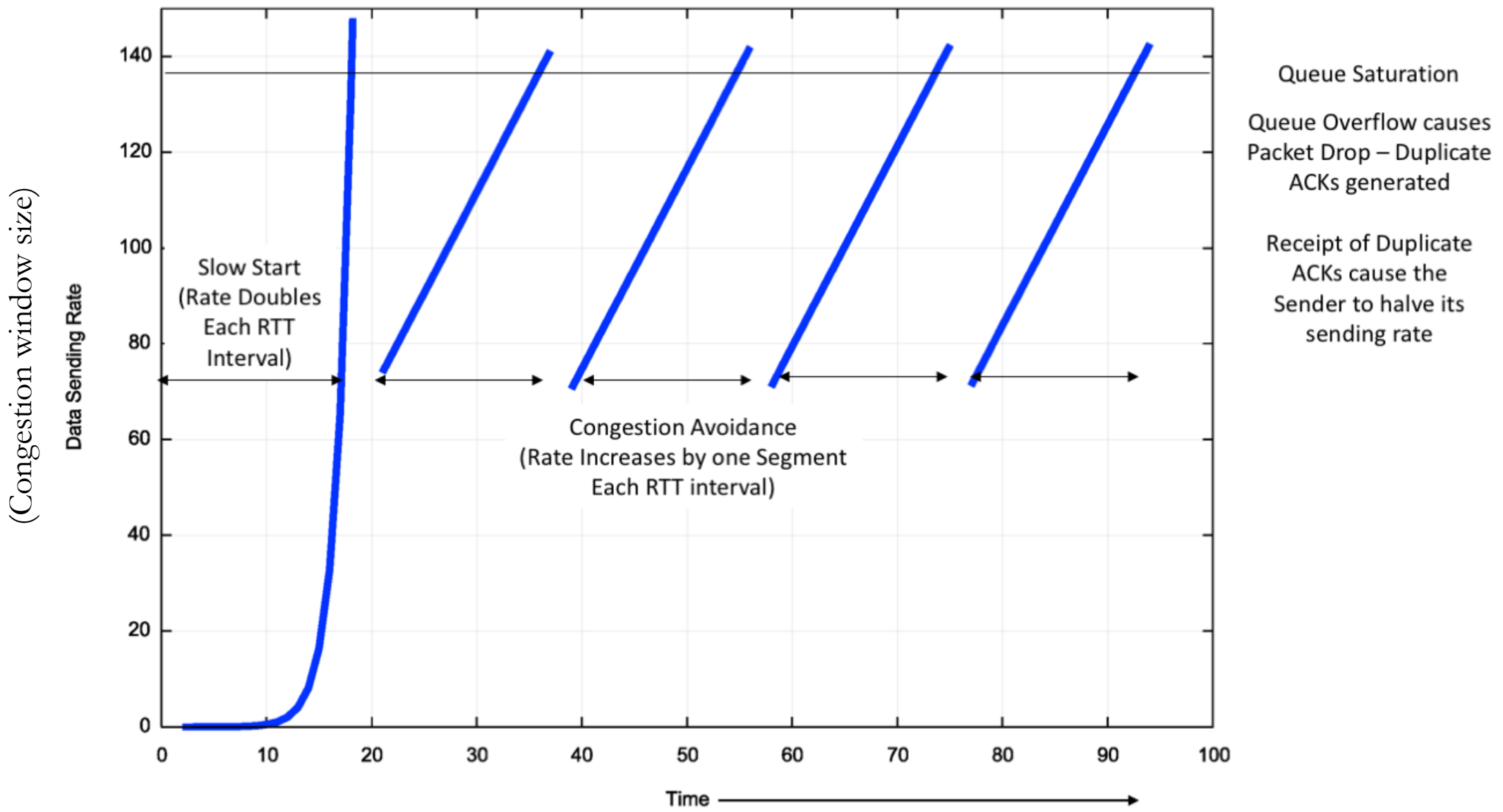
Key Challenge: sender's choice of congestion window size

Basic idea of TCP congestion control:

- Sender tries to *estimate* an appropriate *congestion window* (cwnd)
- # in-flight packets ≤ MIN(cwnd, rwnd) ← rwdn is flow control's "receive window"
- Larger cwnd → greater throughput, greater load on network
- When senders observes packet loss, decrease cwnd.

Precise details are open to debate, RFCs make recommendations.

TCP "Reno"

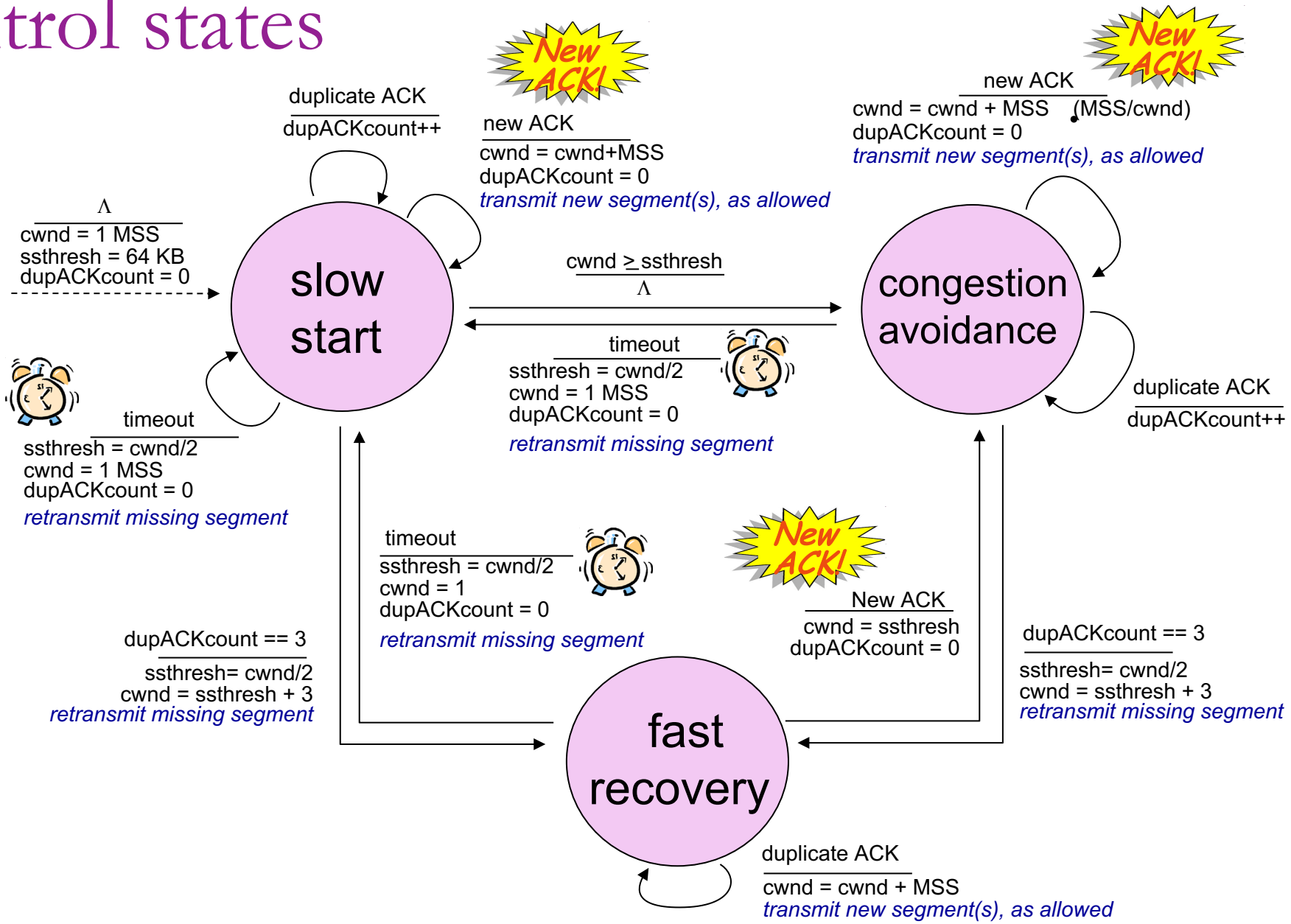

(Congestion window size)
Data Sending Rate
140
120
100
80
60
40
20
0
0
10
20
30
40
50
60
70
80
90
100
Time
Slow Start
(Rate Doubles
Each RTT
Interval)
Congestion Avoidance
(Rate Increases by one Segment
Each RTT interval)
Queue Saturation
Queue Overflow causes
Packet Drop – Duplicate
ACKs generated
Receipt of Duplicate
ACKs cause the
Sender to halve its
sending rate

TCP Reno congestion control operates in phases

1. **Slow start**: initially, network capacity is unknown, so start with cwnd=MSS (maximum segment size ~= 1460 bytes), send 1 packet.
   - *Double* cwnd each time an ACK is received (every RTT).
     This leads to *exponential growth* in cwnd.
   - **Loss by timeout** → *ssthresh* = cwnd/2
   - If cwnd > ssthresh, move to Congestion Avoidance mode.
2. In **Congestion Avoidance** mode:
   - ACK received → cwnd += MSS
     There is *linear growth* in cwnd.
   - **Loss by timeout** → *reset* cwnd = MSS, and move back to *slow start* mode
     Timeout indicates a serious network problem
   - On **loss by triple-duplicate ACK**, move to third phase:
3. **Fast Recovery Mode**:
   - Some packets are still getting through, so don't over-react. Cut cwnd in *half.*
   - cwnd = cwnd/2
   - ACK received → move to *congestion avoidance* mode

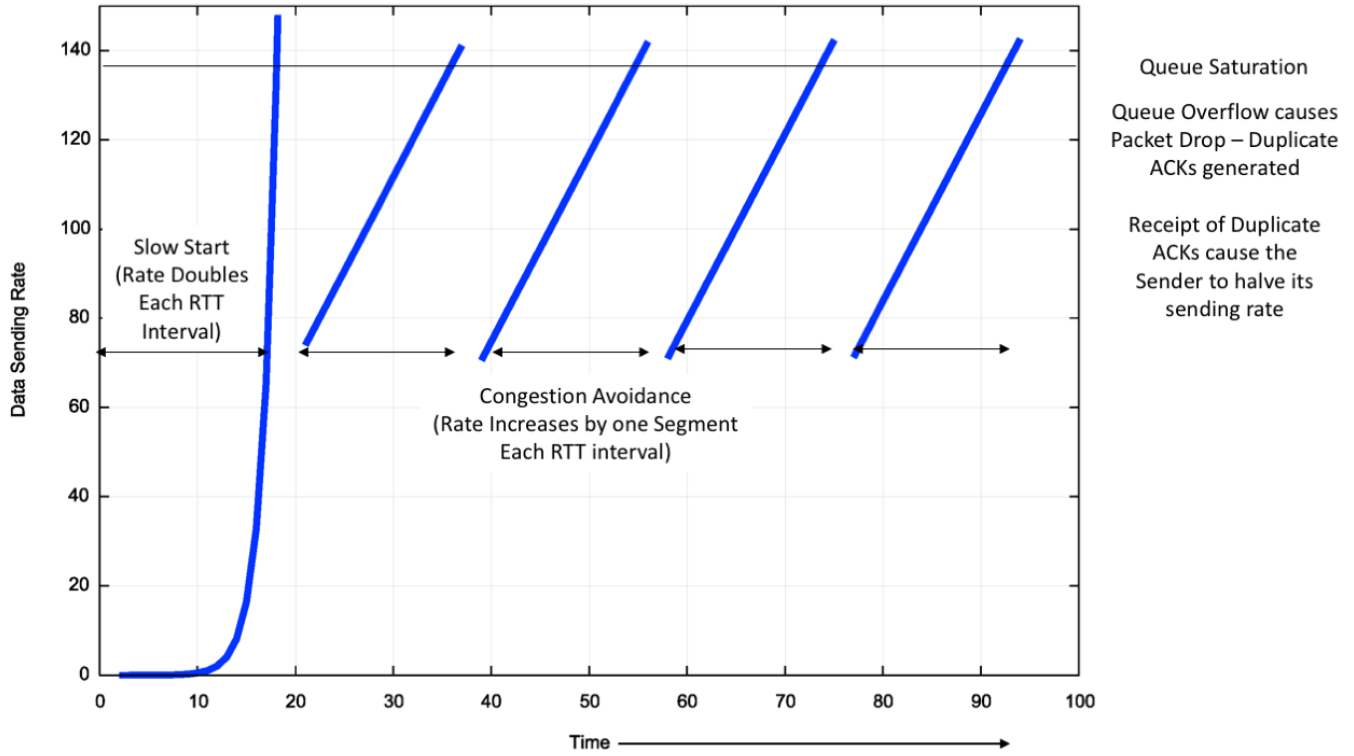
Congestion control states

- Previous slide is missing some details.



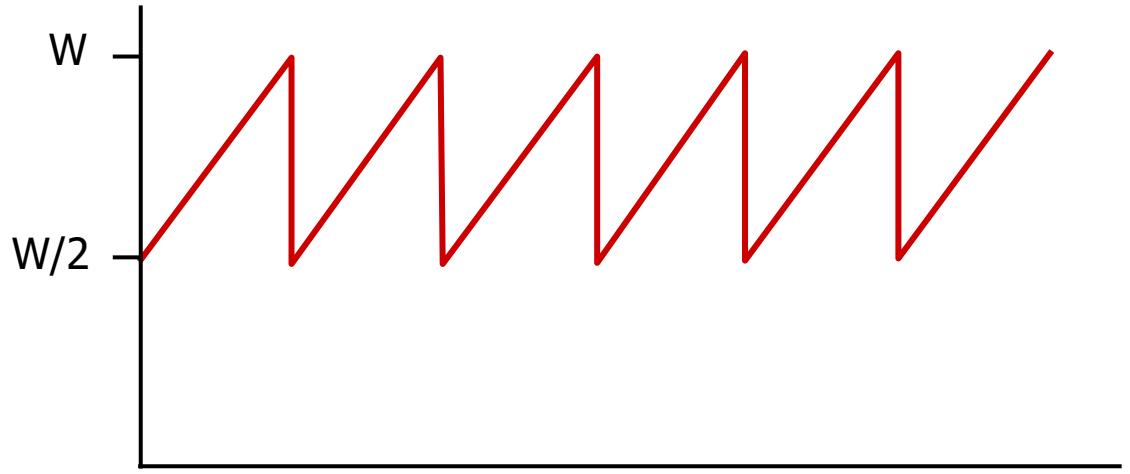
TCP Reno throughput

- Slow start, then congestion avoidance:



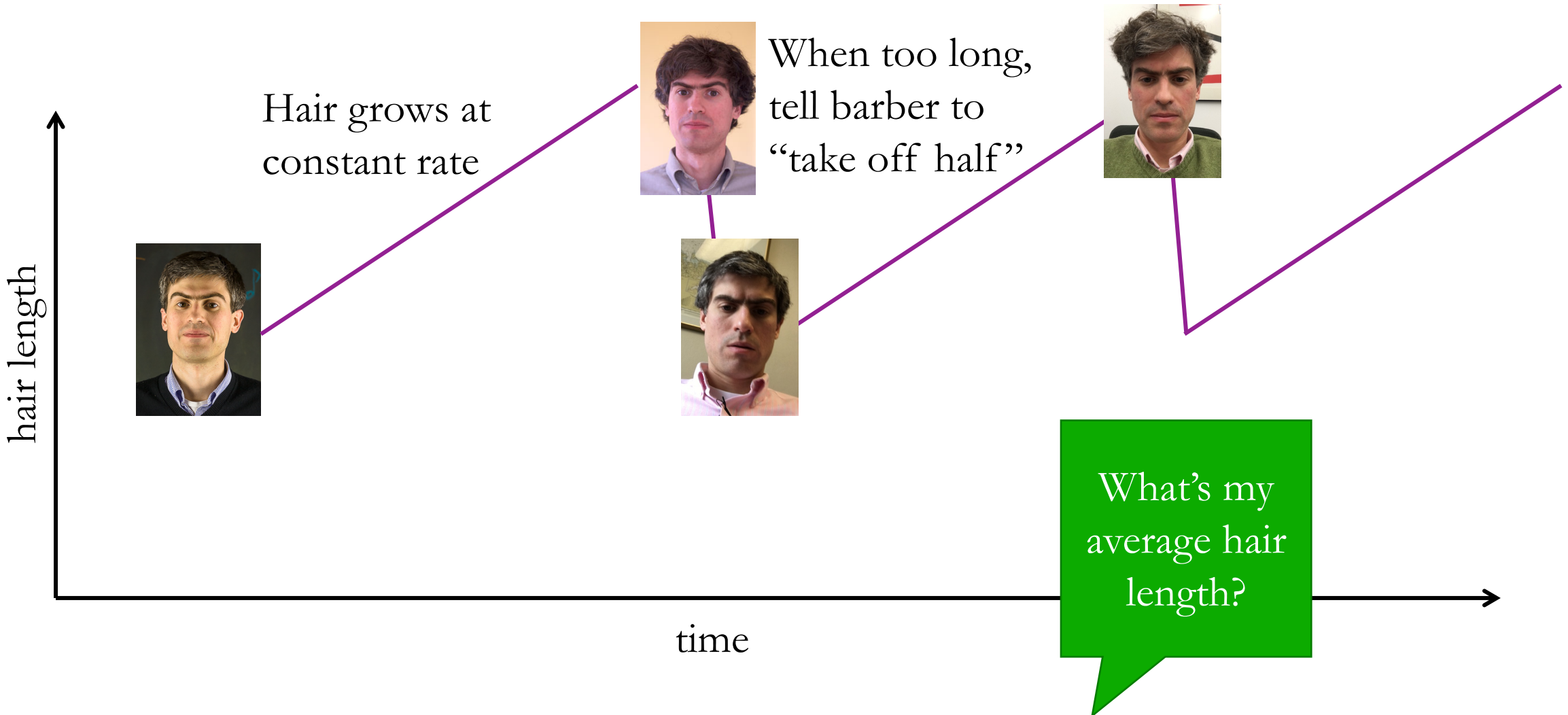
What's Reno's steady-state throughput?

- In congestion avoidance steady state, throughput is approximately **¾ of the max**.
  - Throughput increases linearly until congestion occurs, and it's halved.



Steve's Haircut Algorithm

A TCP congestion control analogy



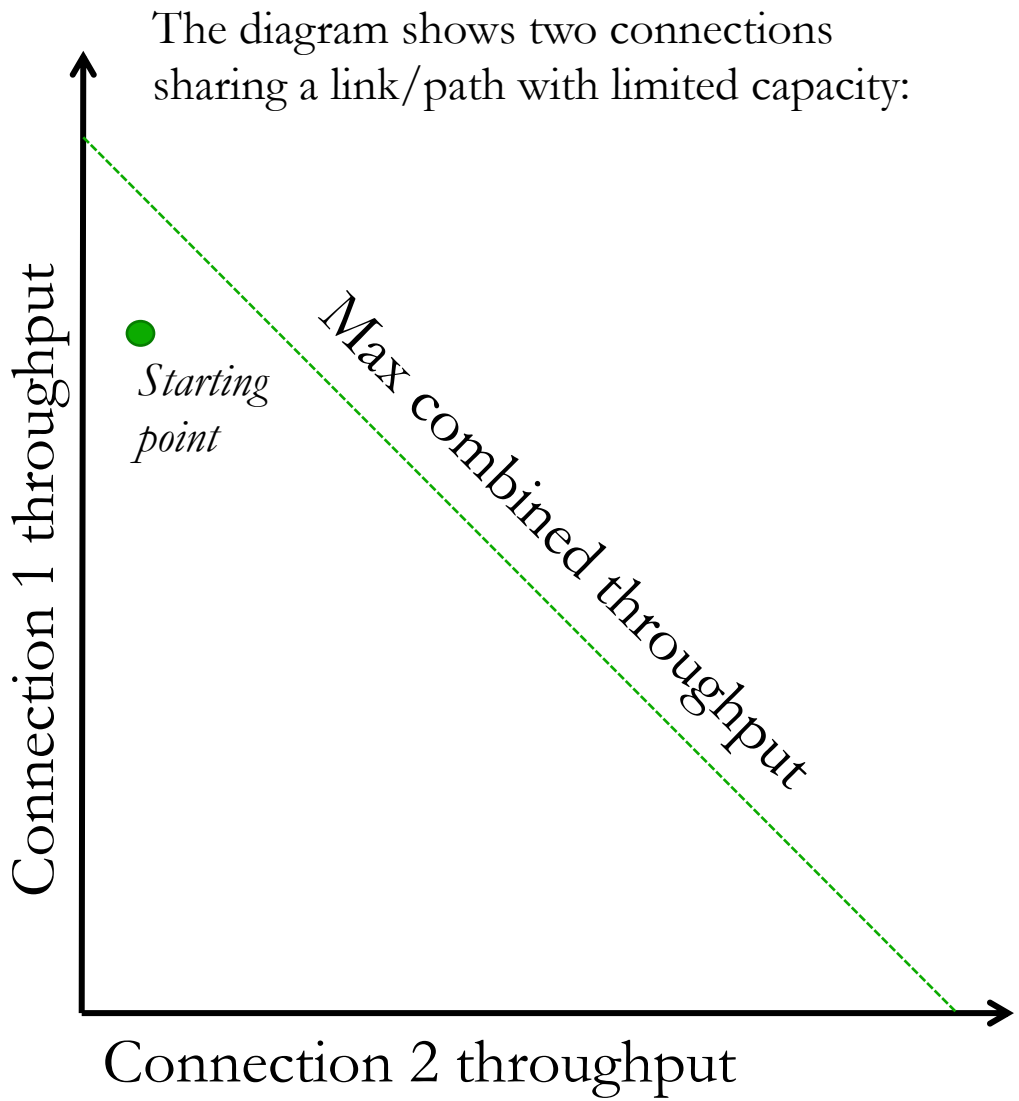
Incentive to follow TCP rules?

Following the rules benefits everyone, but also benefits yourself.

- If I don't throttle myself during congestion, then I will have poor performance in cases when I am the only host causing the congestion.
  - This is is often the case when the first or last hop is the "bottleneck."
- Hosts cannot observe the network directly, just observe packet losses.
  - I cannot tell whether others' traffic is contributing to congestion or whether it's just my own traffic causing the congestion (and thus packet loss).
- Thus, everyone will reduce throughput when congestion occurs, just in case they are solely responsible for the congestion.

- Also, there are only a few TCP implementations, supplied by operating systems, so they generally follow an RFC's recommendations.

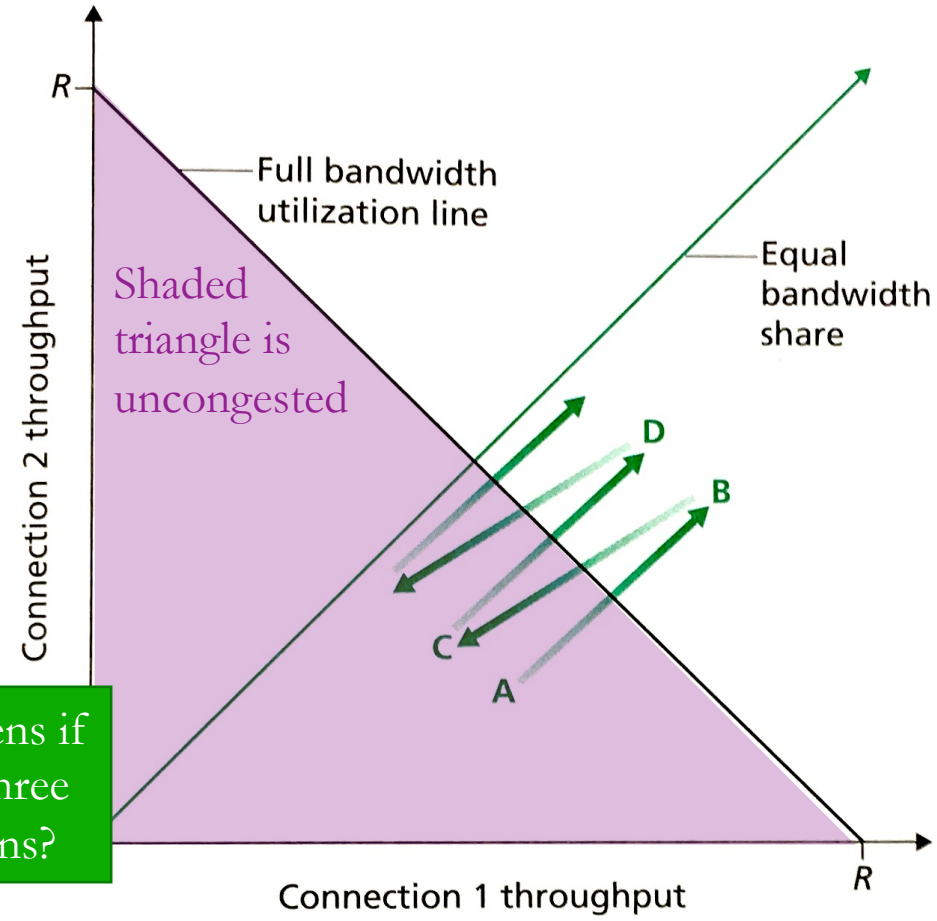
TCP Reno congestion control iteration



- How does the throughput of connections 1 and 2 evolve over time?

- Remember that the two connections follow the TCP rules independently.
- Each controls its own throughput (congestion window size), but is unaware that the other connection.
  - Don't know other connection's cwnd.
  - Don't even know other connection exists!
- Initially 1 is faster than 2, how does this change?
  - Where does the point move?

Reno eventually leads to equal share of bandwidth



What happens if there are three connections?

Figure 3.55 ♦ Throughput realized by TCP connections 1 and 2

- During congestion, packets are dropped and bandwidth (window sizes) are cut in half.
  - Connection with more bandwidth share ***always loses more*** when window is cut in half.
  - Move halfway toward the origin in the plot.
- During congestion avoidance, bandwidth of both connections increases at the same rate, moving at 45° angle up-right.

Intermission

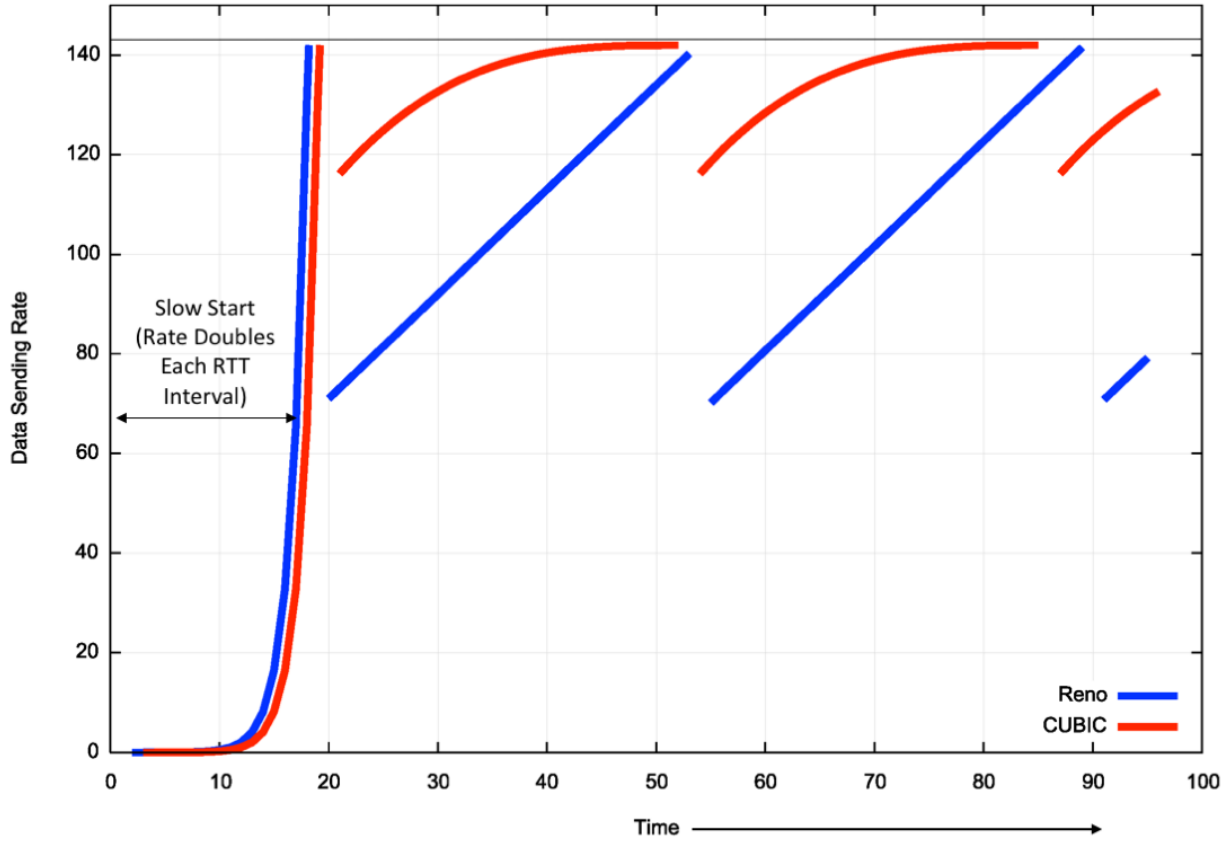
Driving past Reno

- TCP congestion control in the Linux kernel can use many algorithms.
  - They can be dynamically switched by the user. For example:
    ```
    $ echo "cubic" > /proc/sys/net/ipv4/tcp_congestion_control
    ```
- TCP congestion algorithm in open source OSes has evolved like so:
  - **Tahoe**: BSD 4.3 Tahoe (1988)
  - **Reno**: BSD 4.3 Reno (1990)
  - **New Reno with SAC**: Linux 2.4?–2.6.17 (2001?–2004)
  - **BIC**: Linux 2.6.8–2.6.18 (2004–2006)
  - **CUBIC**: Linux 2.6.19–3.2 (2006–2012)
  - **PRR**: Linux 3.2–4.9 (2012–2016)
  - **BBR**: Linux 4.9–present (2016–present)

Problems with TCP Reno

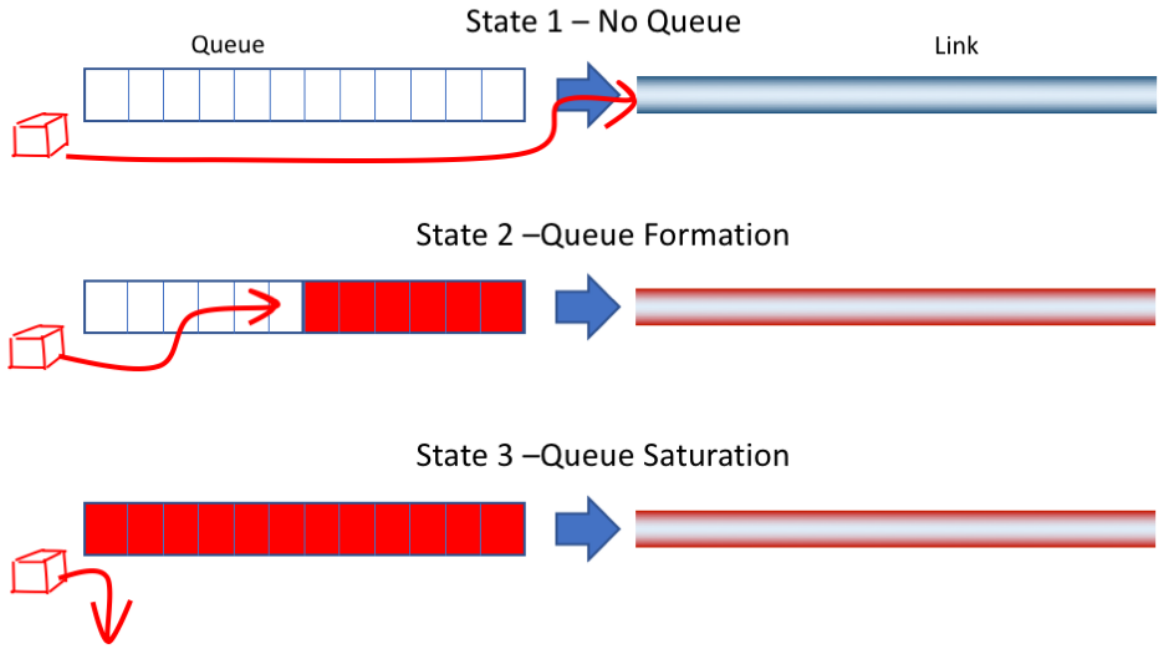
- In congestion avoidance, linear increase is too slow on fast, high latency links (long fat pipes)
- Consider a 10Gbps link with 30ms RTT.
  - Increasing window linearly (1500 bytes per RTT) would require **3.5 hours** to move from 5 to 10 Gbps.
  - This is especially bad if we lose packets due to bit corruptions:
    - In 3.5 hours 100 trillion bits will be sent!
    - Perfect reliability on that scale is unlikely.
- **Solution**: grow faster!

- Binary Increase (BIC) and CUBIC congestion control use a kind of binary search to *exponentially* grow the window:



Packet loss means full queues and wasted time!

- Reno and CUBIC both rely on packet loss to detect congestion.
- They keep the links busy by moving between states 2 and 3:
- **Bufferbloat** is the unnecessary delay added to RTT because router buffers are too large.
- It would be better to operate between states 1 and 2.
- Throughput would still be high, but without queueing delays.

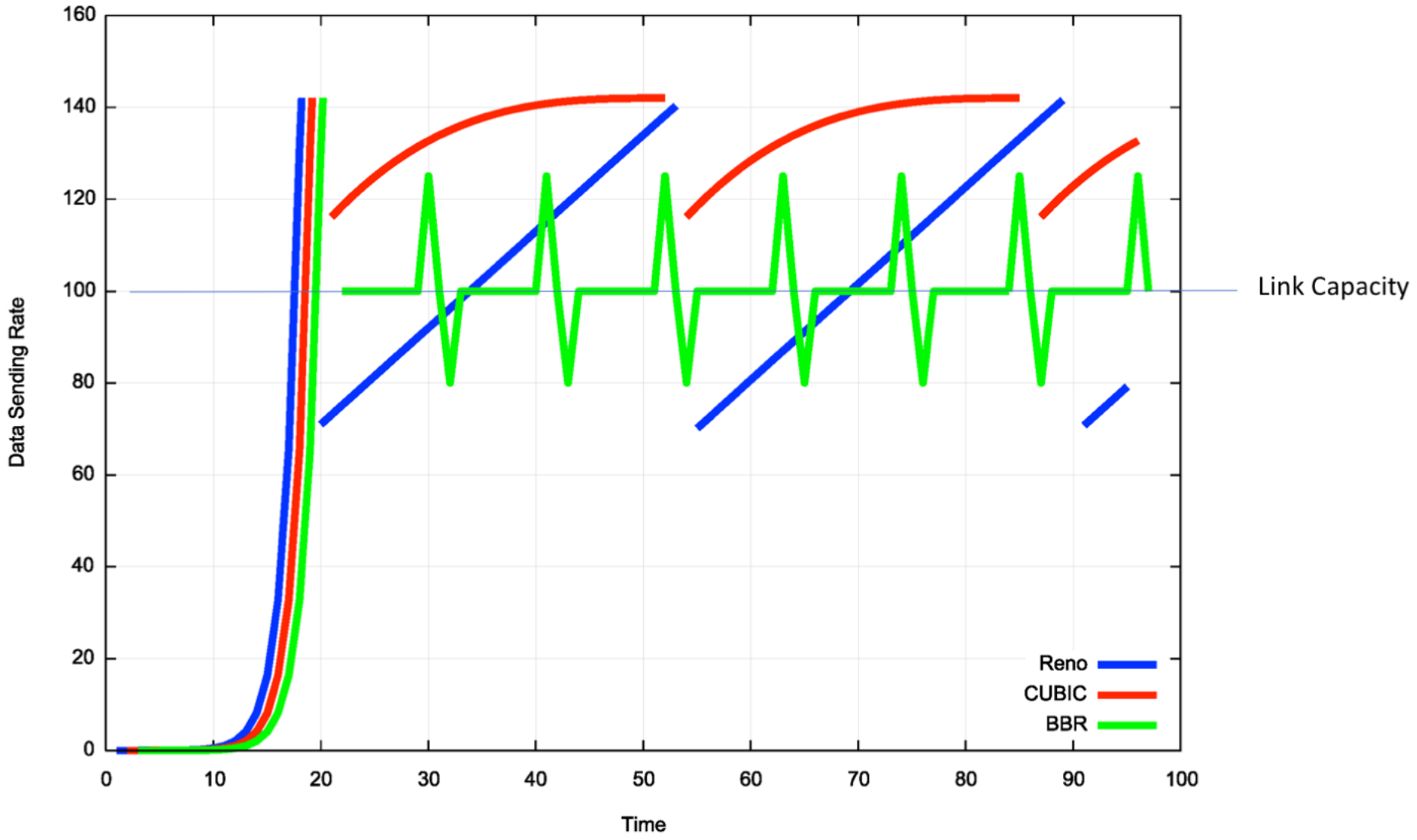


- There is no packet loss until we get to state 3, so what else can we monitor to detect state 2?

Vegas and **BBR** Congestion Control

- The RTT of packets can give information about queue states.
- TCP Vegas introduced the idea of looking for increases in RTT (instead of packet loss) as a signal of congestion.
  - If we're below the max throughput, RTT should be fixed.
  - When we send above the max throughput, queues start to fill and RTT rises.
  - Vegas achieves throughput near the maximum while keeping router buffers nearly empty. **RTT is optimized,** not just throughput.
  - Unfortunately, Vegas flows do not get a fair share of capacity when competing with Reno and CUBIC flows.
- **BBR** (Bottleneck Bandwidth and Round-trip propagation time) fixes this shortcoming and seems to be the best approach so far.
  - We will not cover its details. It was published by Google in 2016.

BBR in action



A few final tweaks for TCP

Nagle's algorithm merges small packets

- An application may write a series of small message to a TCP stream.
  - Eg: `write("OK\n"); write("READY\n"); write("GO\n");`
- A simple implementation of TCP would send segments for each write.
- But each TCP packet has 40 bytes of ***header overhead***.
  - Merging small packets into one larger packet would reduce network load:

Total size includes three headers before merging:

After merge:

$$(40+3) + (40+6) + (40+3) \rightarrow (40+12)$$

$$132 \rightarrow 52 \text{ bytes}$$

- Wait until segment is full before sending, *unless* there are no un-ACK'ed segments outstanding (eg., send first segment immediately).

Interactive applications

- Interactive apps and bulk-transfer apps prefer different TCP behavior.
- **Socket options** give applications some control of the underlying TCP:
- TCP_NODELAY socket option disables Nagle's algorithm.
  - Every *write* → segment(s) are sent immediately (if allowed by window).
  - Nagle's algorithm adds extra latency which may hurt performance of applications that send small, *time-sensitive* data. (eg., GUI events).
- TCP_NOPUSH is even more aggressive than standard Nagle.
  - Wait until send buffer is full before sending segment(s).
  - Also, don't set PSH bit (to maximize buffering on the receiver's side as well).
- Usually the **PSH bit** will be set on the last segment in a *write* call.
  - PSH tells the receiving TCP implementation to alert the receiving process that that data is ready.

TCP Keepalive

- An ***idle*** TCP connection involves no data exchange.
- Optionally, a TCP host may occasionally send an empty data segment, called a **keepalive message**, just to test whether an ACK will return.
  - Keepalive has SEQ # one less than expected, to trigger an ACK response.
  - Low frequency, ~once per minute.
- Keepalives are disabled by default and only used in special situations:
  - SSH clients give the option to enable TCP keepalives.
  - This forces NAT routers to keep the port mapping alive.
- Some application-level protocols have their own keepalive msgs.

TCP Recap

- **Congestion control** is implemented with a dynamic *congestion window*, controlled by *heuristics*. **Reno** congestion control operate in phases:
  - *Slow start* – exponential growth to find approximate network capacity.
  - *Congestion avoidance* – linear growth, slowly trying to increase throughput.
  - *Fast recovery* – If one packet is lost, then cut window in half.
- **Additive Increase, Multiplicative Decrease** ensures fair sharing.
- **CUBIC** and **BBR** are newer alternatives that work better in fast networks, particularly long fat pipes. (Read this for more info.)
- TCP behavior can be controlled with *socket options*:
  - Nagle's algorithm merges small packets to reduce *header overhead*.
  - TCP *keepalive* message can be periodically sent.