Flow Control; Congestion Control

Lecture 16

http://www.cs.rutgers.edu/~sn624/352-F24

Srinivas Narayana

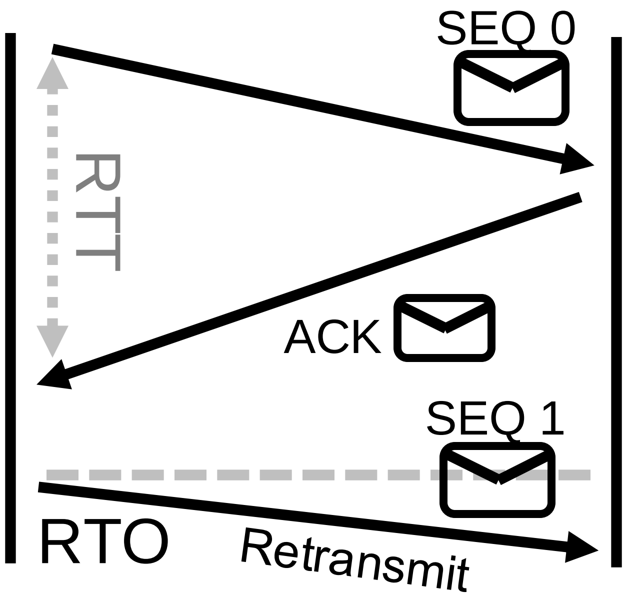
RUTGERS
UNIVERSITY | NEW BRUNSWICK

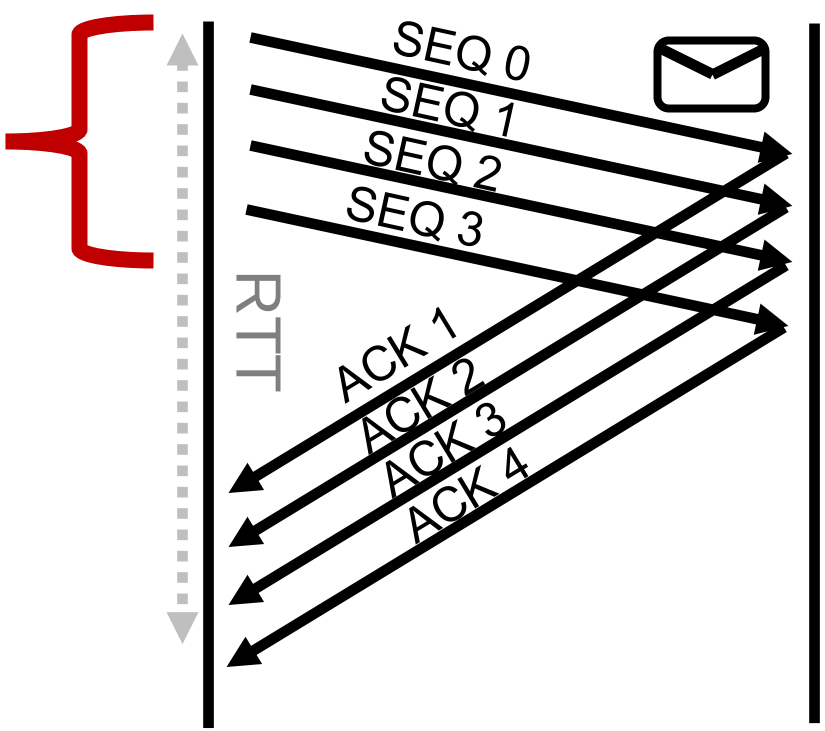
= window size

Proportional to throughput

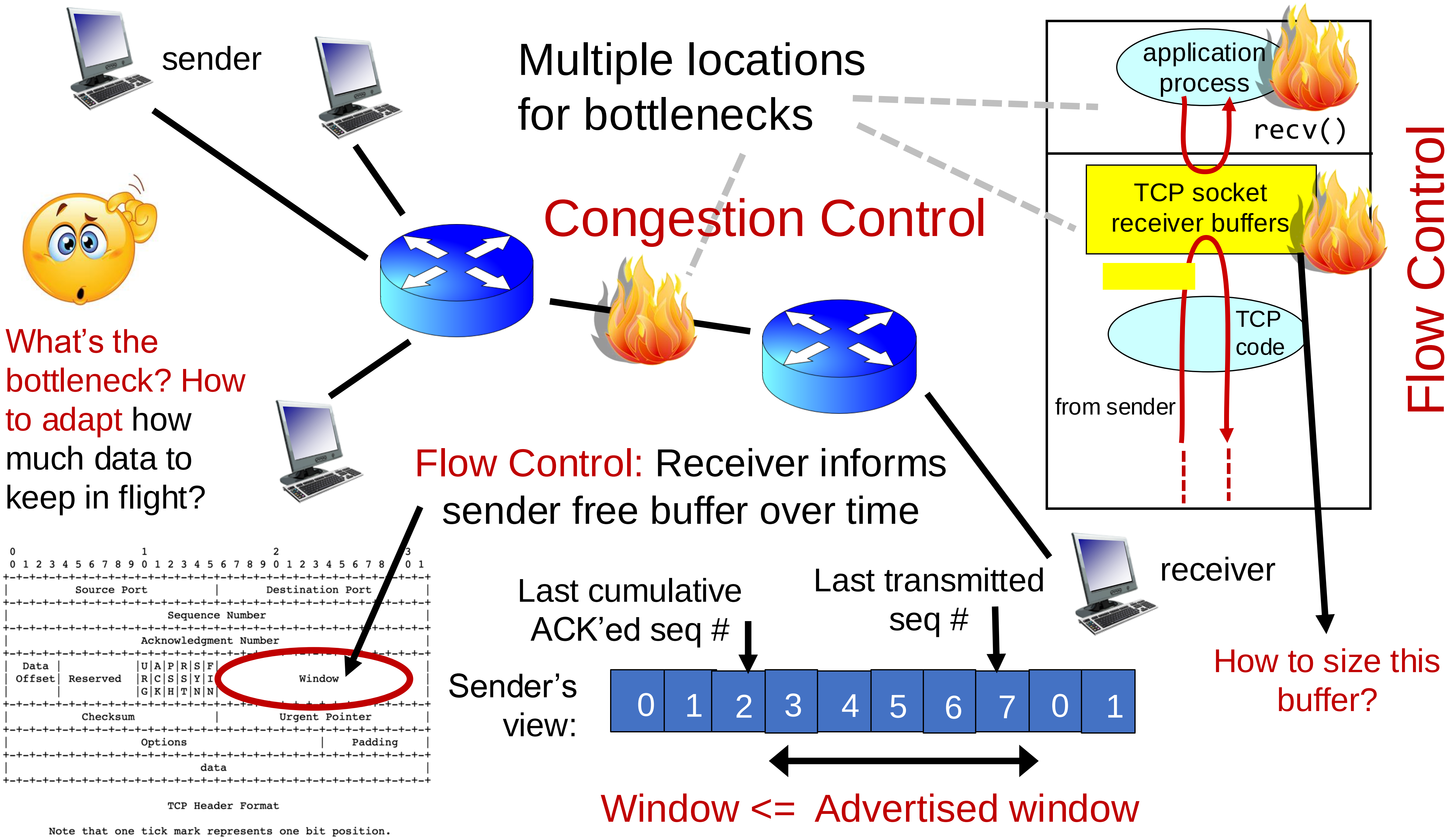
How much data to keep in flight?

Stop and Wait





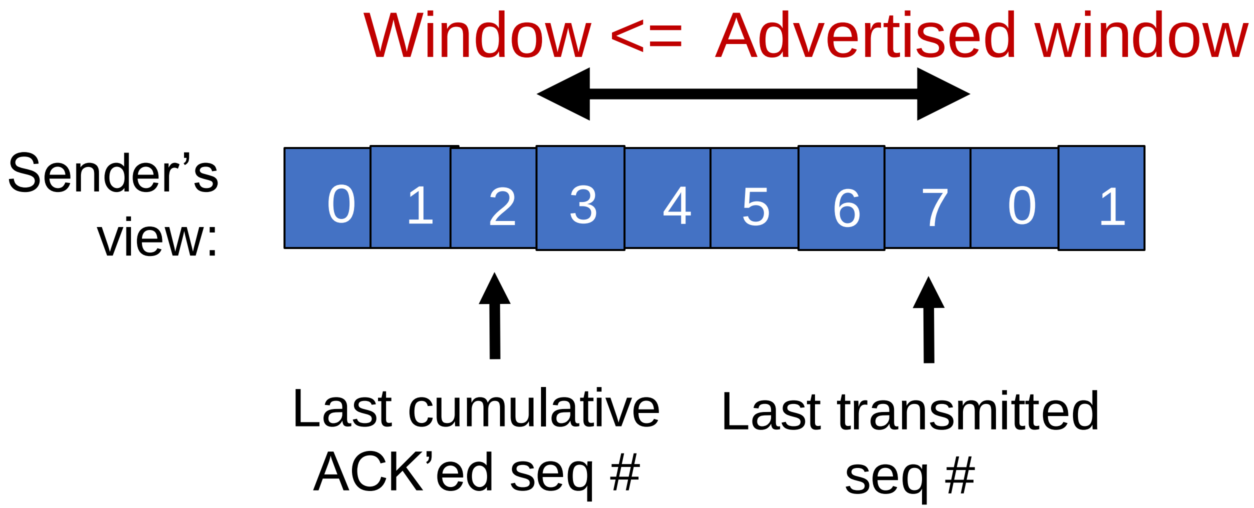
Pipelined Reliability

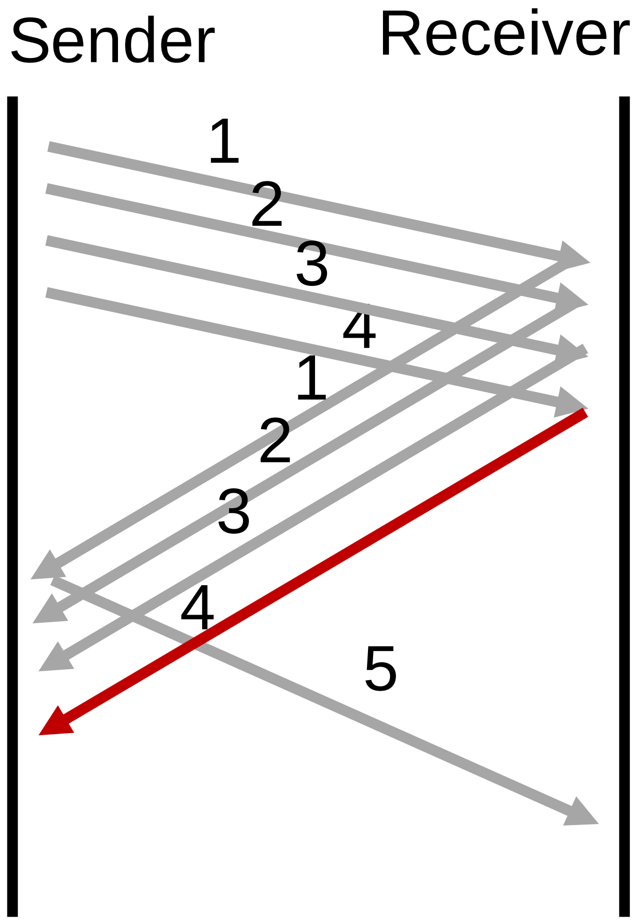


sender
Multiple locations
for bottlenecks
Congestion Control
What's the bottleneck? How to adapt how much data to keep in flight?
Flow Control: Receiver informs sender free buffer over time
application process
recv()
TCP socket receiver buffers
TCP code
from sender
Flow Control
receiver
How to size this buffer?
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 Offset
Reserved
U R G
A C K
P S H
R S T
S Y N
F I N
Window
Checksum
Urgent Pointer
Options
Padding
data
TCP Header Format
Note that one tick mark represents one bit position.
Last cumulative ACK'ed seq #
Last transmitted seq #
Sender's view:
0 1 2 3 4 5 6 7 0 1
Window <= Advertised window

TCP flow control

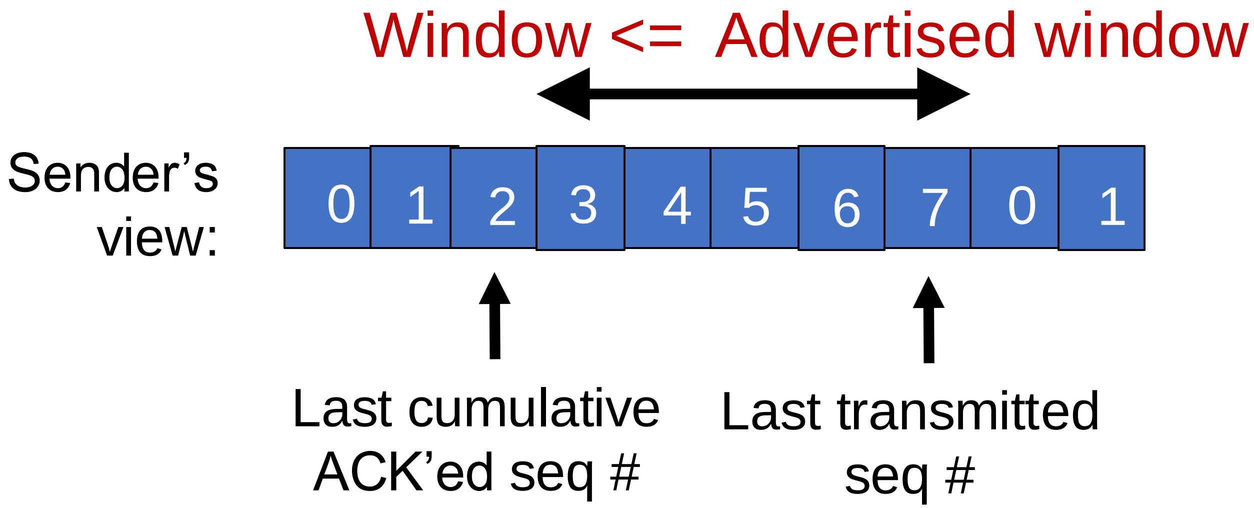
- If receiver app is too slow reading data:
  - receiver socket buffer fills up
  - => advertised window shrinks
  - => sender's window (sending rate) reduces
  - => sender's socket buffer fills up
  - => sender process put to sleep upon send()

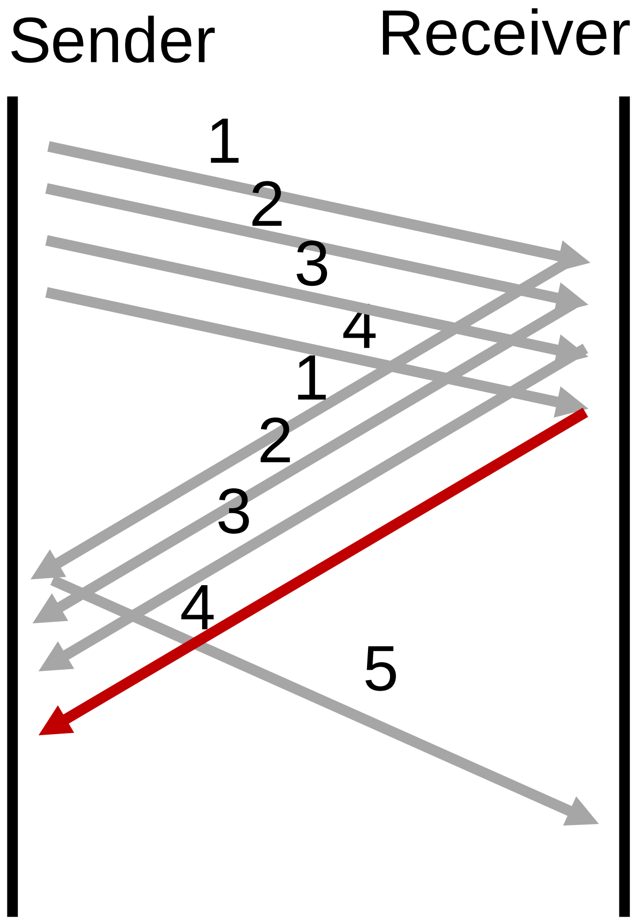




TCP flow control

Flow control matches the sending process’s write speed to the receiving process’s read speed.





Sizing the receiver's socket buffer

- Operating systems have a default receiver socket buffer size
  - Listed among `sysctl -a | grep net.inet.tcp` on MAC
  - Listed among `sysctl -a | grep net.ipv4.tcp` on Linux
- If socket buffer is too small, sender can't keep too many packets in flight ➔ lower throughput
- If socket buffer is too large, too much memory consumed per socket
- How big should the receiver socket buffer be?

Sizing the receiver's socket buffer

- Case 1: Suppose the receiving app is reading data too slowly:
- No amount of receiver buffer can prevent low throughput (for a long-lived connection).
- Flow control matches throughput to the receiving app's (low) speed

Sizing the receiver's socket buffer

- Case 2: Suppose the receiving app reads sufficiently fast *on average* to match the sender's writing speed.
  - Assume the sender desires a window of size W.
  - The receiver must use a buffer of size at least W. Why?
- Captures two cases:
- (1) When the first sequence #s in the window are dropped
  - *Selective repeat*: data in window buffered until the "hole" within the window can be filled by the sender. Advertised window reduces sender's window
- (2) When the sender sends a burst of data of size W
  - The receiver may not keep up with the *instantaneous* rate of the sender
- Set receiver socket buffer size > desired window size

Summary of flow control

- Keep memory buffers available at the receiver whenever the sender transmits data
- Buffers needed to hold for selective repeat, reassembling data in order, and until applications can read data
- Inform available buffer to sender on an ongoing basis, with each ACK
- Function: match sender speed to receiver speed

- Correct socket buffer sizing is important for TCP throughput
  - Throughput = window size / RTT <= receiver socket buffer / RTT

Info on (tuning) TCP stack parameters

- https://www.ibm.com/support/knowledgecenter/linuxonibm/liaag/wkvm/wkvm_c_tune_tcpip.htm

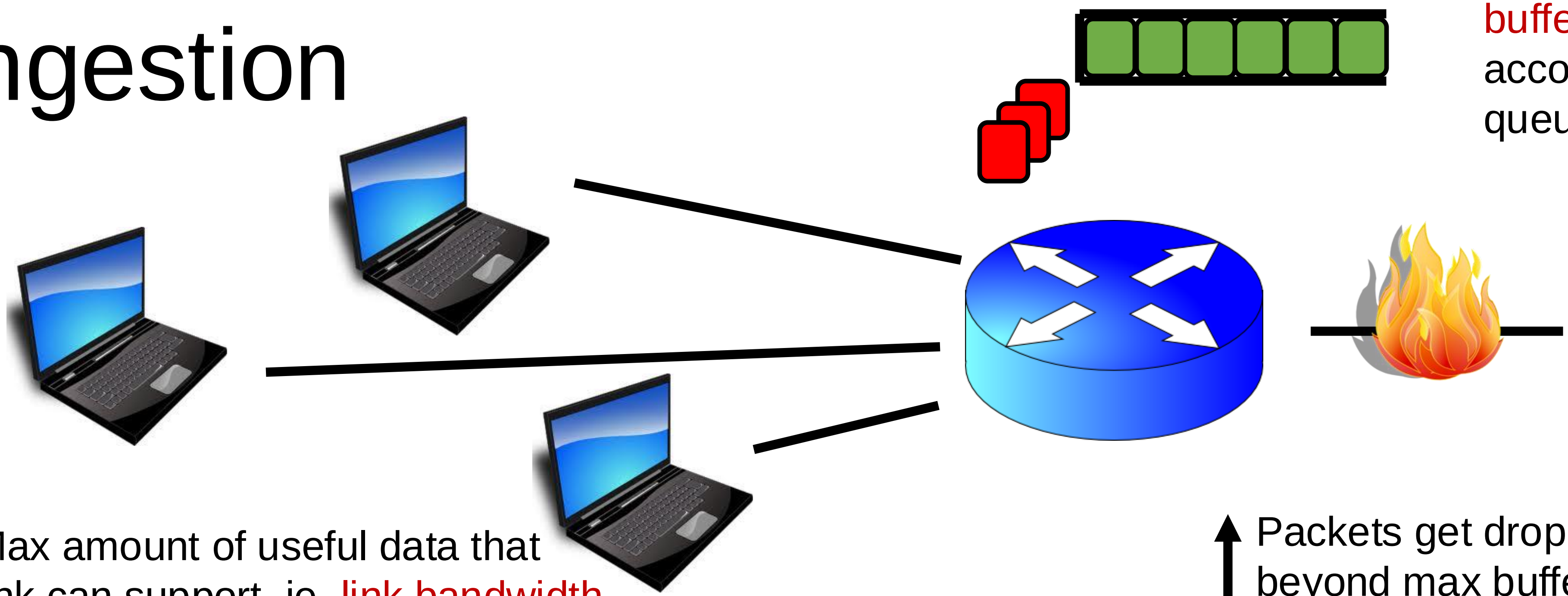
- https://cloud.google.com/solutions/tcp-optimization-for-network-performance-in-gcp-and-hybrid

Playing around with socket buffer sizes

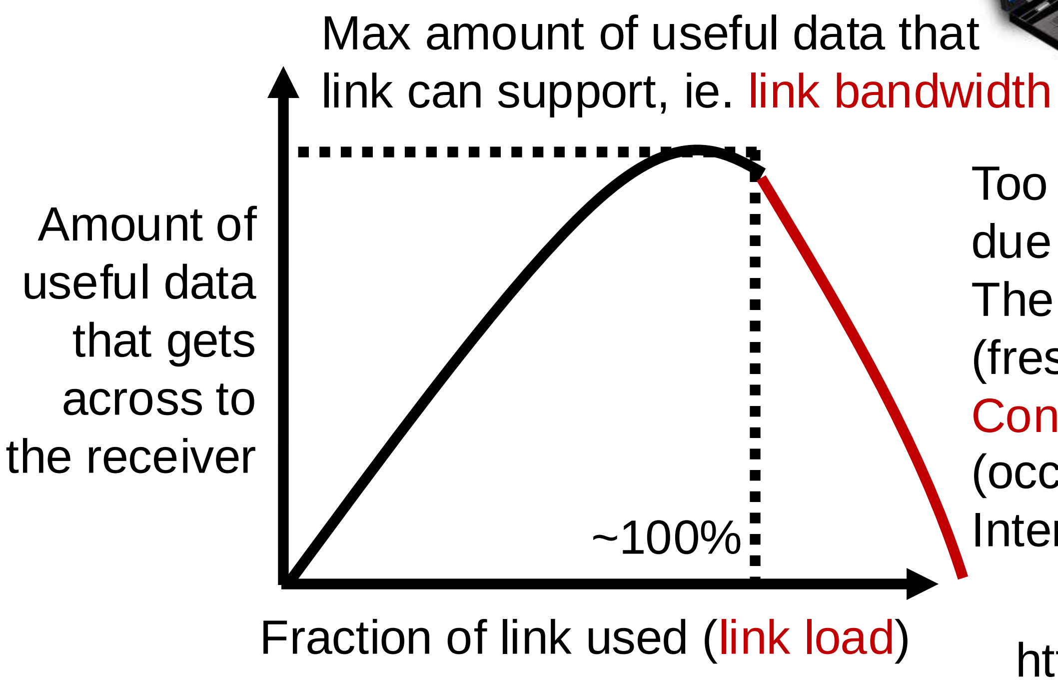
- iperf –s ; iperf –c localhost –i 1
- ping localhost
- sudo tc qdisc add dev lo root netem delay 100ms
- sudo sysctl net.ipv4.tcp_rmem # min, default, max
- Default buffer size 128KB; change e.g., 2.56MB by using
  - sudo sysctl net.ipv4.tcp_rmem=“4096 2621440 6291456”
- Clean up and restore to defaults
- sudo tc qdisc del dev lo root netem
  - If needed:
  - sudo sysctl net.ipv4.tcp_rmem=“4096 131072 6291456”

Congestion Control

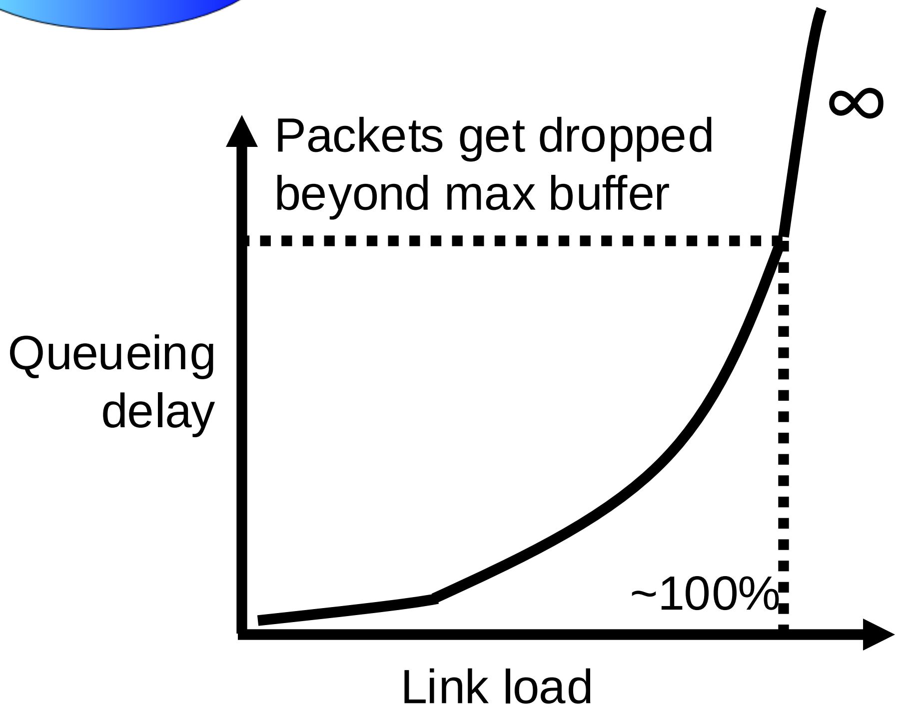
Congestion



Routers have buffers which accommodate queued packets.

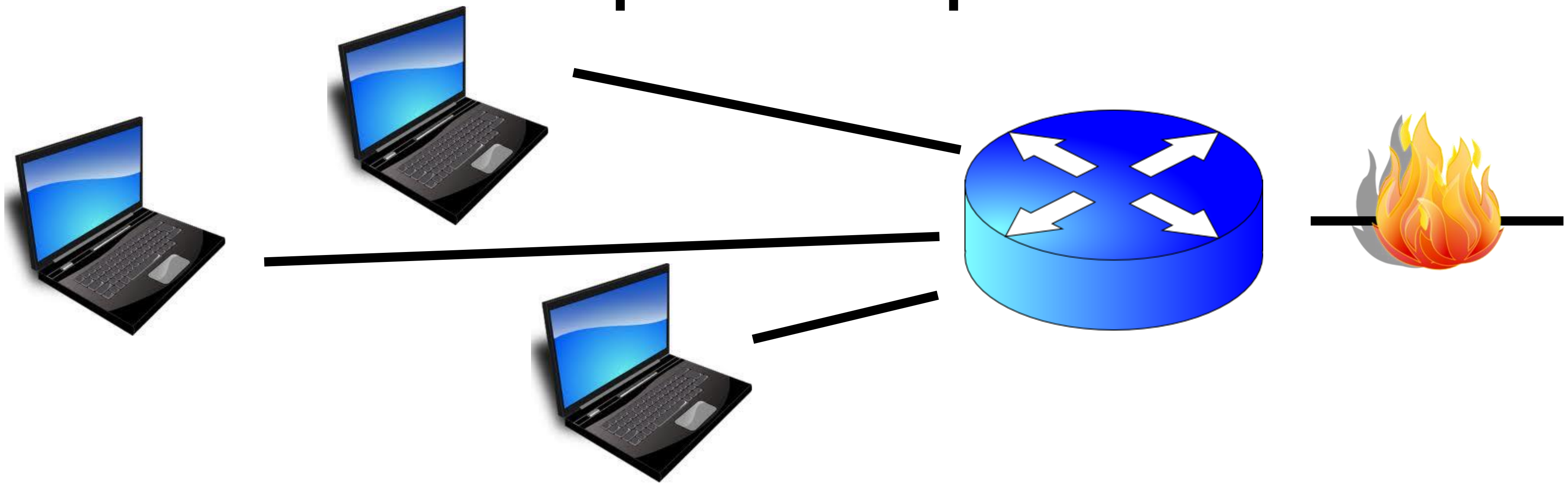


Too many retransmissions due to packet drops! The amount of useful (fresh) data plummets. Congestion collapse (occurred for real on the Internet in the last 80s)



https://en.wikipedia.org/wiki/Network_congestion#Congestive_collapse

How should multiple endpoints share net?



- It is difficult to know where the bottleneck link is
- It is difficult to know how many other endpoints are using that link
- Endpoints may join and leave at any time
- Network paths may change over time, leading to different bottleneck links (with different link rates) over time

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

No one can centrally view or control all the endpoints and bottlenecks in the Internet.

Every endpoint must try to reach a globally good outcome by itself: i.e., in a distributed fashion.

This also puts a lot of trust in endpoints.

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

If there is spare capacity in the bottleneck link, the endpoints should use it.

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

If there are N endpoints sharing a bottleneck link, they should be able to get equitable shares of the link's capacity.

For example: 1/N'th of the link capacity.

Flow Control vs. Congestion Control

- Avoid overwhelming the receiving application
- Sender is managing the receiver's socket buffer

- Avoid overwhelming the bottleneck network link
- Sender is managing the bottleneck link capacity and bottleneck router buffers

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

How to achieve this?

Approach: sense and react

Example: showering: Want “just right” water

Use a feedback loop with signals and knobs

Signals and Knobs in Congestion Control

- Signals
  - Packets being ACK’ed
  - Packets being dropped (e.g. RTO fires)
  - Packets being delayed (RTT)
  - Rate of incoming ACKs

Implicit feedback signals measured directly at sender. (There are also explicit signals that the network might provide.)

- Knobs
  - What can you change to “probe” the available bottleneck capacity?
    - Window size
  - Suppose the receiver socket buffer size is unbounded
    - Congestion window: window size used for congestion control
  - Increase window/sending rate: e.g., add x or multiply by a factor of x
  - Decrease window/sending rate: e.g., subtract x or reduce by a factor of x

Sense and react, sure…but how?

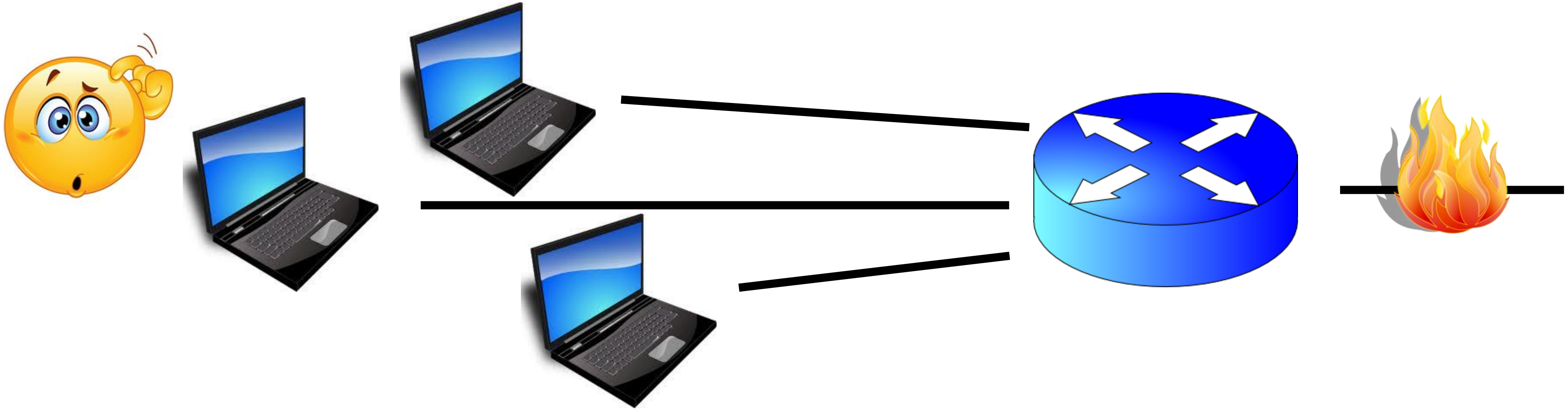
- Where do you want to be?
  - The steady state
- How do you get there?
  - Congestion control algorithms
- Sense accurately & react accordingly

The Steady State

Efficiency for a single TCP connection

What does efficiency look like?

- Suppose we want to achieve an efficient outcome for one TCP connection by observing network signals from the endpoint



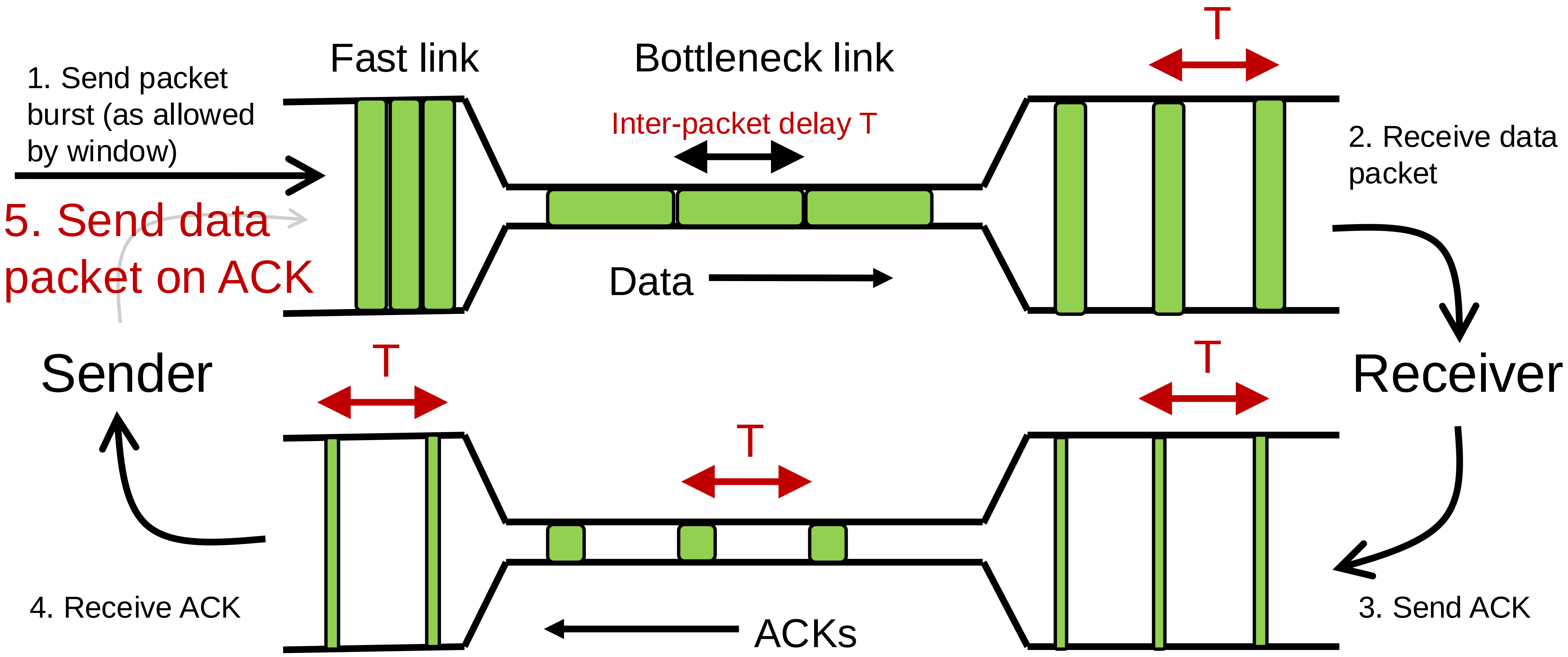
- Q: How should the endpoint behave at steady state?
- Challenge: bottleneck link is remotely located

Steady state: Ideal goal

- High sending rate: Use the full capacity of the bottleneck link
- Low delay: Minimize the overall delay of packets to get to the receiver
  - Overall delay = propagation + queueing + transmission
  - Assume propagation and transmission components fixed
- “Low delay” reduces to low queueing delay
- i.e., don’t push so much data into the network that packets have to wait in queues

- Key question: When to send the next packet?

When to send the next packet?



Rationale

- When the sender receives an ACK, that's a signal that the previous packet has left the bottleneck link (and the rest of the network)
- Hence, it must be safe to send another packet without congesting the bottleneck link
- Such transmissions are said to follow packet conservation
- ACK clocking: "Clock" of ACKs governs packet transmissions

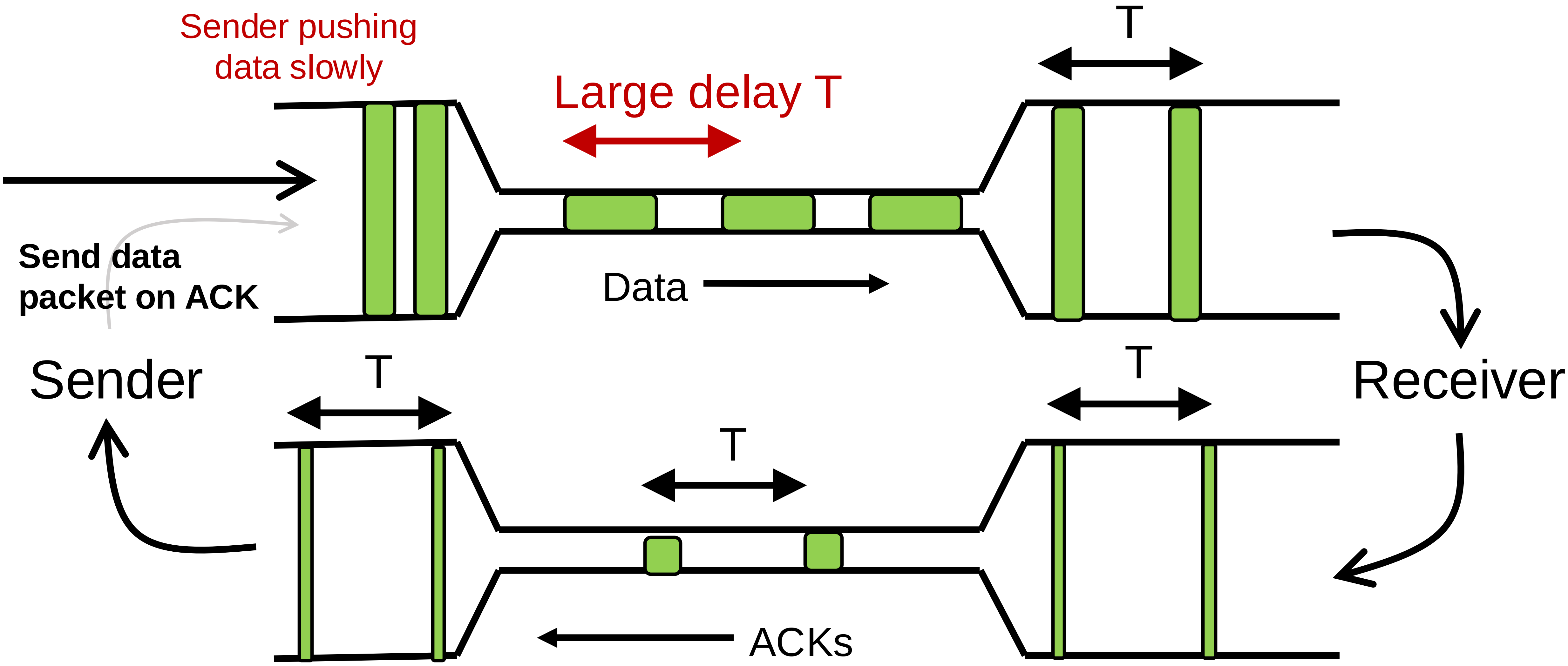
ACK clocking: analogy

- How to avoid crowding a grocery store?

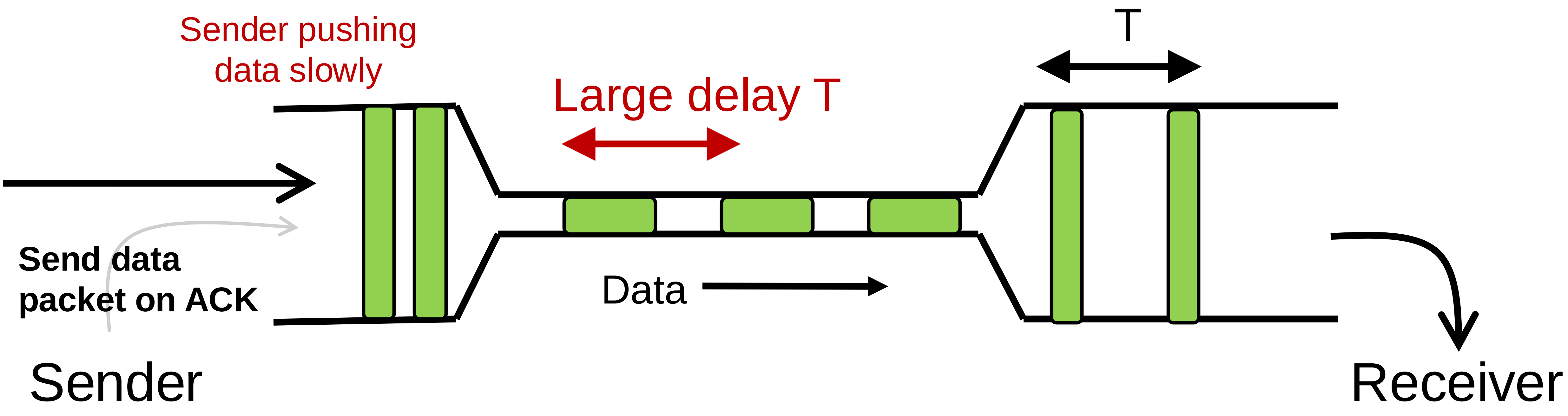
- Strategy: Send the next waiting customer exactly when a customer exits the store

- However, this strategy alone can lead to inefficient use of resources…

ACK clocking alone can be inefficient



ACK clocking alone can be inefficient



The sending rate should be high enough to keep the “pipe” full

Analogy: a grocery store with only 1 customer in entire store

If the store isn’t “full”, you’re using store space inefficiently

Steady State of Congestion Control

- Send at the highest rate possible (to keep the pipe full)
- while being ACK-clocked (to avoid congesting the pipe)

- So, how to get to steady state?

Finding the Right Congestion Window

Let’s play a game

- Suppose I’m thinking of a number (positive integer). You need to guess the number I have in mind.
- Each time you guess, I will tell you whether your number is smaller or larger than (or the same as) the one I’m thinking of
- My number can be very large or small
- How would you go about guessing the number?

Finding the right congestion window

- TCP congestion control algorithms solve a similar problem!
- There is an unknown bottleneck link rate that the sender must match
- If sender sends more than the bottleneck link rate:
  - packet loss, delays, etc.
- If sender sends less than the bottleneck link rate:
  - all packets get through; successful ACKs

Quickly finding a rate: TCP slow start

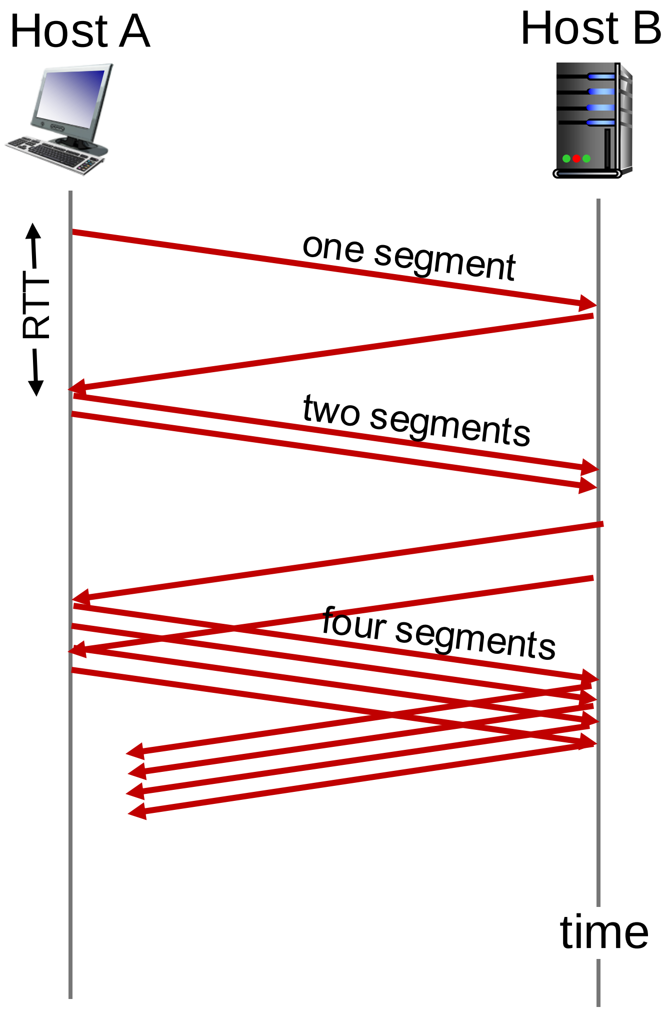
- Initially `cwnd` = 1 MSS
  - MSS is “maximum segment size”

- Upon receiving an ACK of each MSS, increase the `cwnd` by 1 MSS

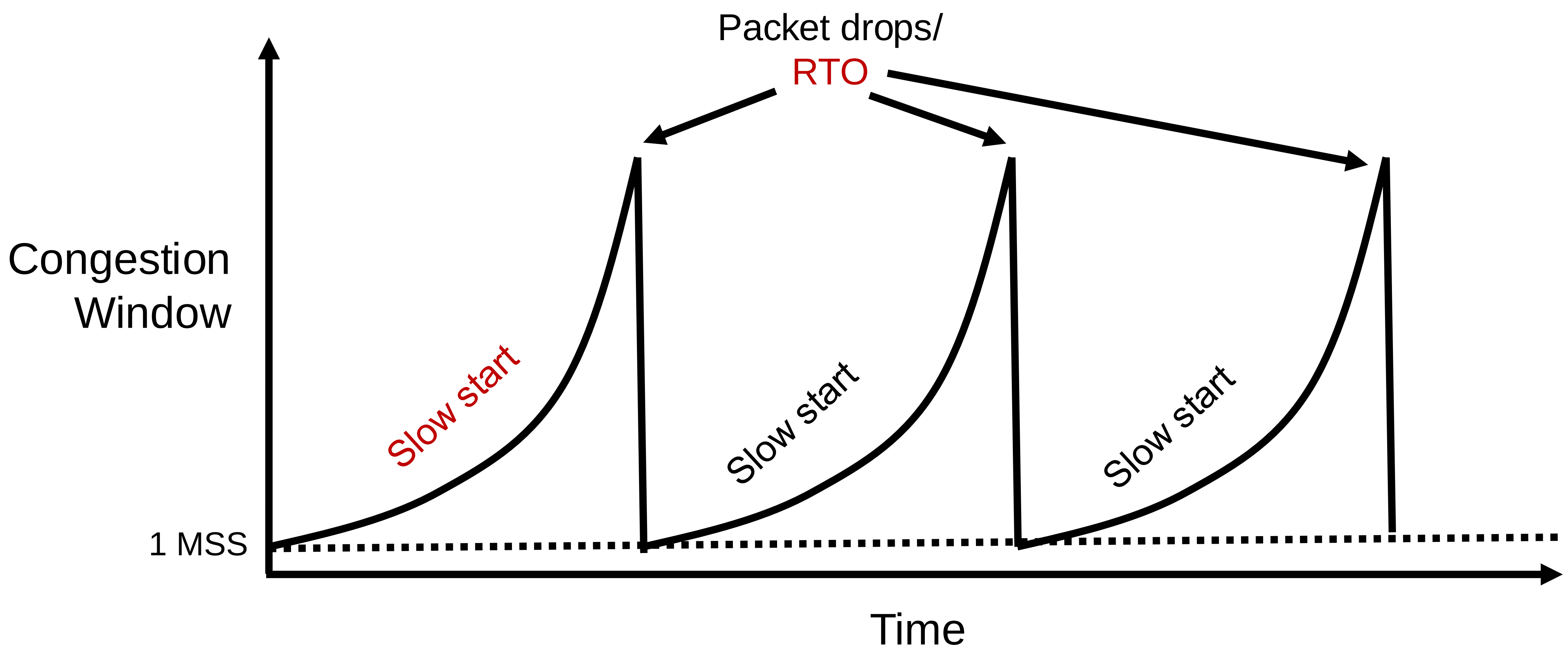
- Effectively, double `cwnd` every RTT

- Initial rate is slow but ramps up exponentially fast

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



Behavior of slow start



Slow start has problems

- 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

- Instead, perform finer adjustments of `cwnd` based on signals

Use slow start mainly at the beginning

- You might accelerate your car a lot when you start, but you want to make only small adjustments after.
  - Want a smooth ride, not a jerky one
- Slow start is a good algorithm to get close to the bottleneck link rate when there is little info available about the bottleneck, e.g., the beginning of a connection
- Once close enough to the bottleneck link rate, use a different set of strategies to perform smaller adjustments to the congestion window
  - Called TCP congestion avoidance