CS 352
Congestion Control: Intro

CS 352, Lecture 12.1

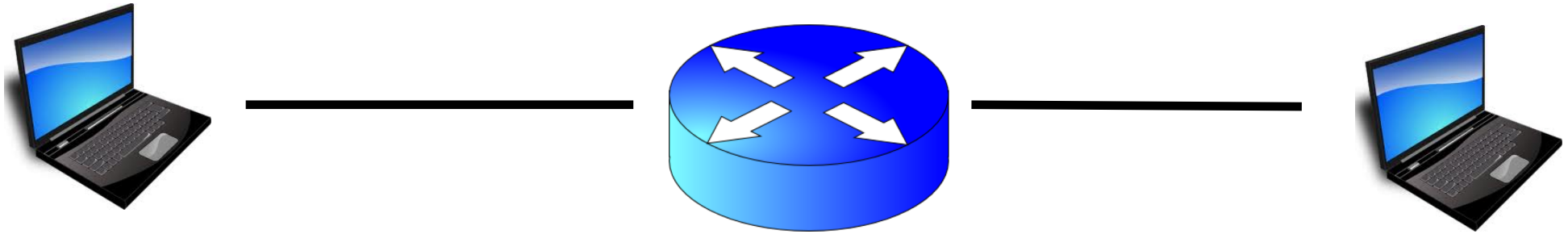
http://www.cs.rutgers.edu/~sn624/352

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

How do apps get perf guarantees?

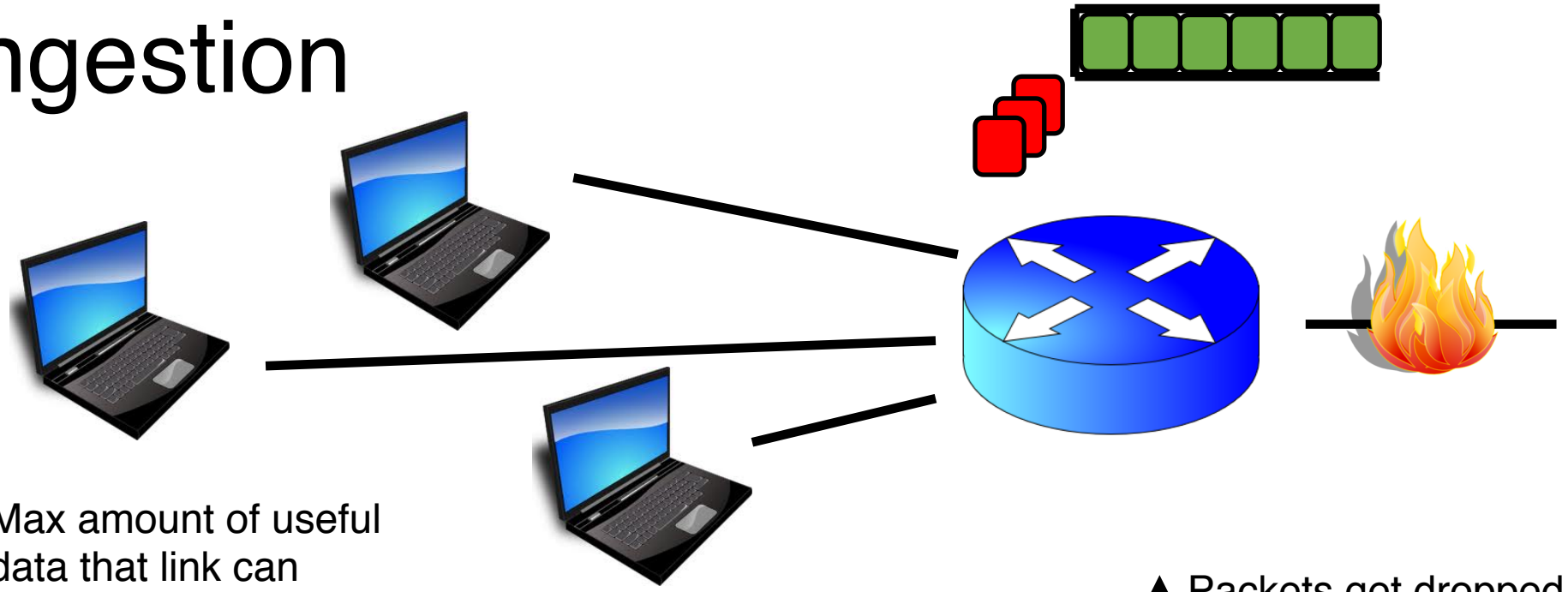
- The network core provides no guarantees on packet delivery

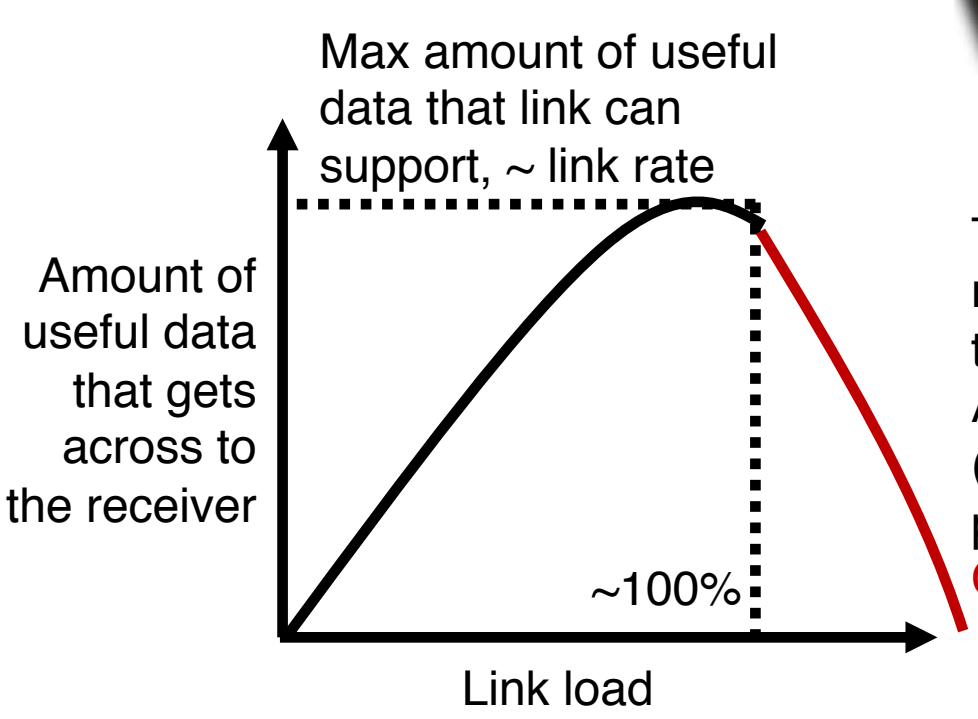


- Transport software on the endpoint oversees implementing guarantees on top of a best-effort network
- Three important kinds of guarantees
  - Reliability
  - Ordered delivery
  - Resource sharing in the network core

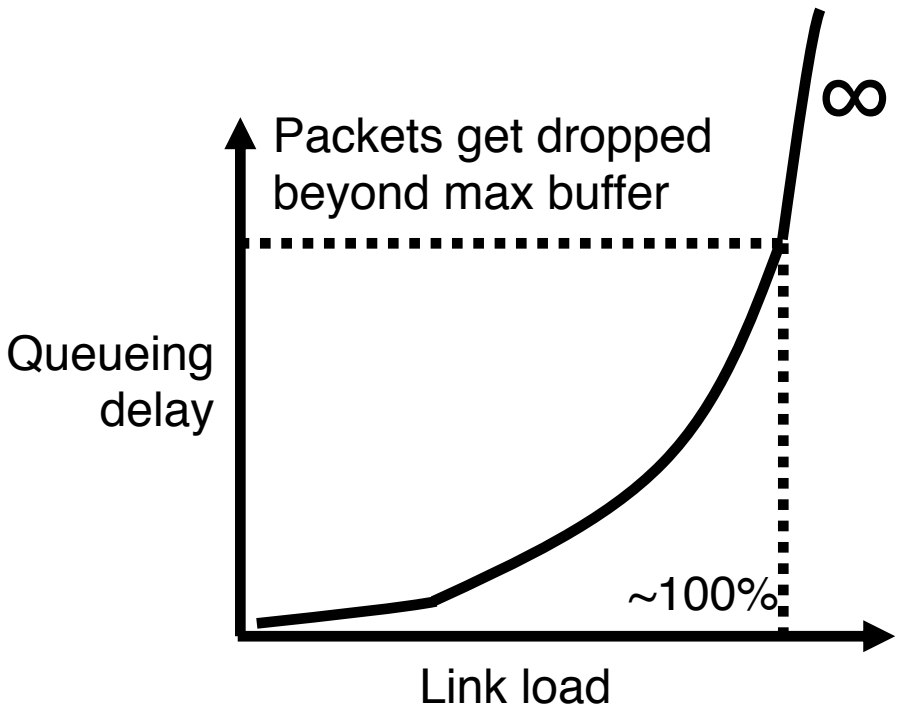
} Transmission Control Protocol (TCP)

Congestion

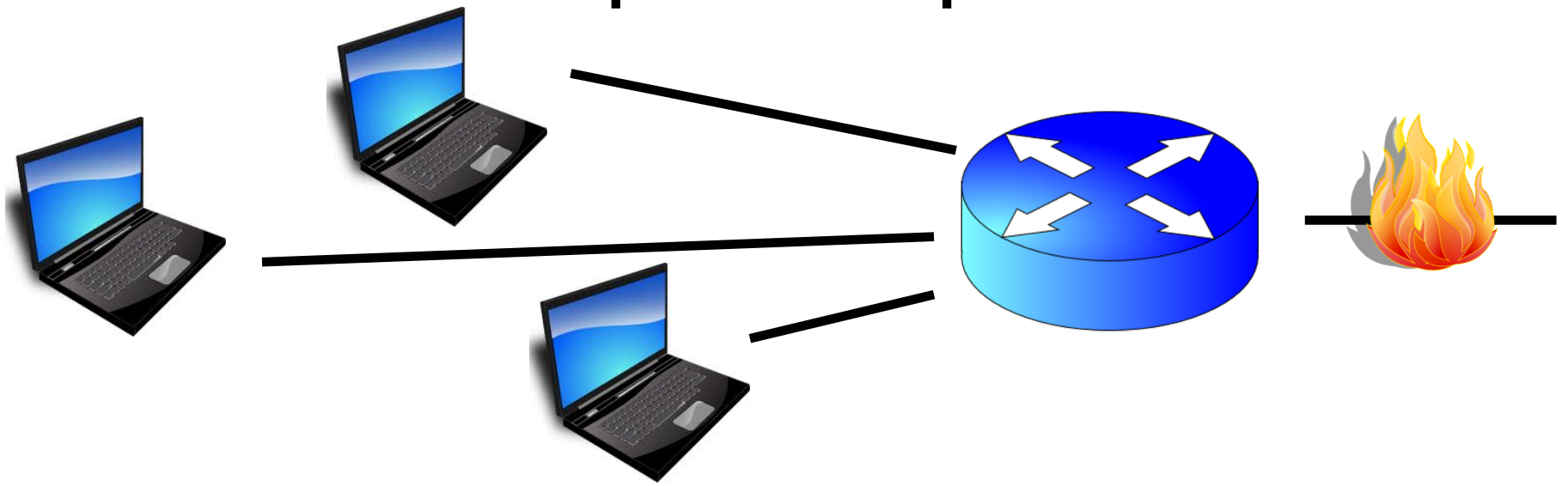




Too many retransmissions due to packet drops!
Amount of useful (fresh) data plummets.
Congestion collapse



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: taking a shower
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?
  - Suppose receiver buffer is unbounded:
  - 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

Subsequent modules/lectures

- Feedback loops used by 2 important TCPs
  - TCP New Reno and TCP BBR
- Strategies to react “proportionately” to network signals like loss
  - Fast retransmit and fast recovery
- Strategies to measure loss accurately
  - How to predict the RTT of a packet successfully received (in the future)?

CS 352
The Steady State

CS 352, Lecture 12.2

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

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

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

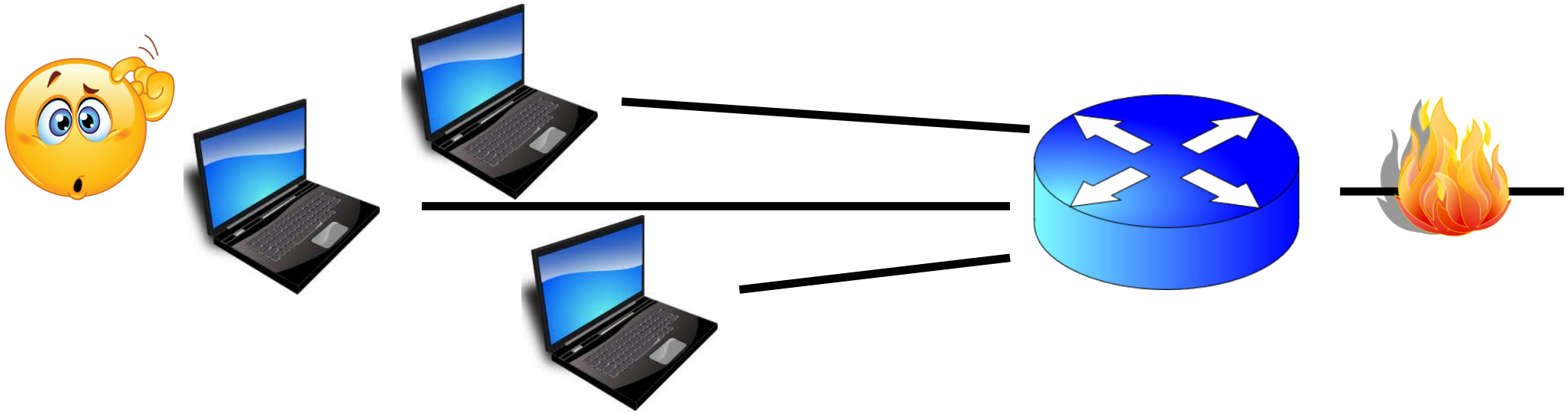
How to achieve this?

Design a feedback loop: measure signals, apply knobs, and repeat.

Efficiency with a Single Conversation

What does efficiency look like?

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



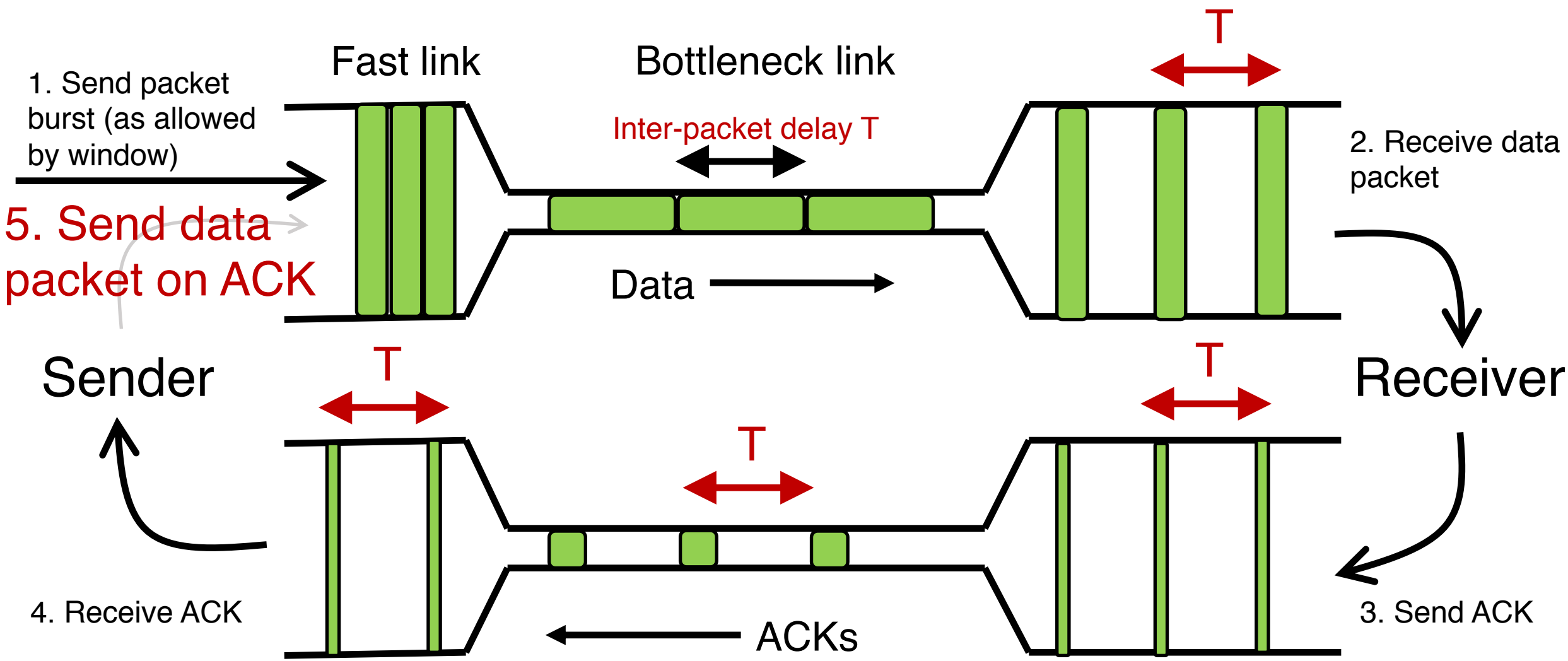
- Q1: How should the endpoint behave at steady state?
- Q2: How should the endpoint get to steady state? (next module)
- Challenge: bottleneck link is remotely located

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 other 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

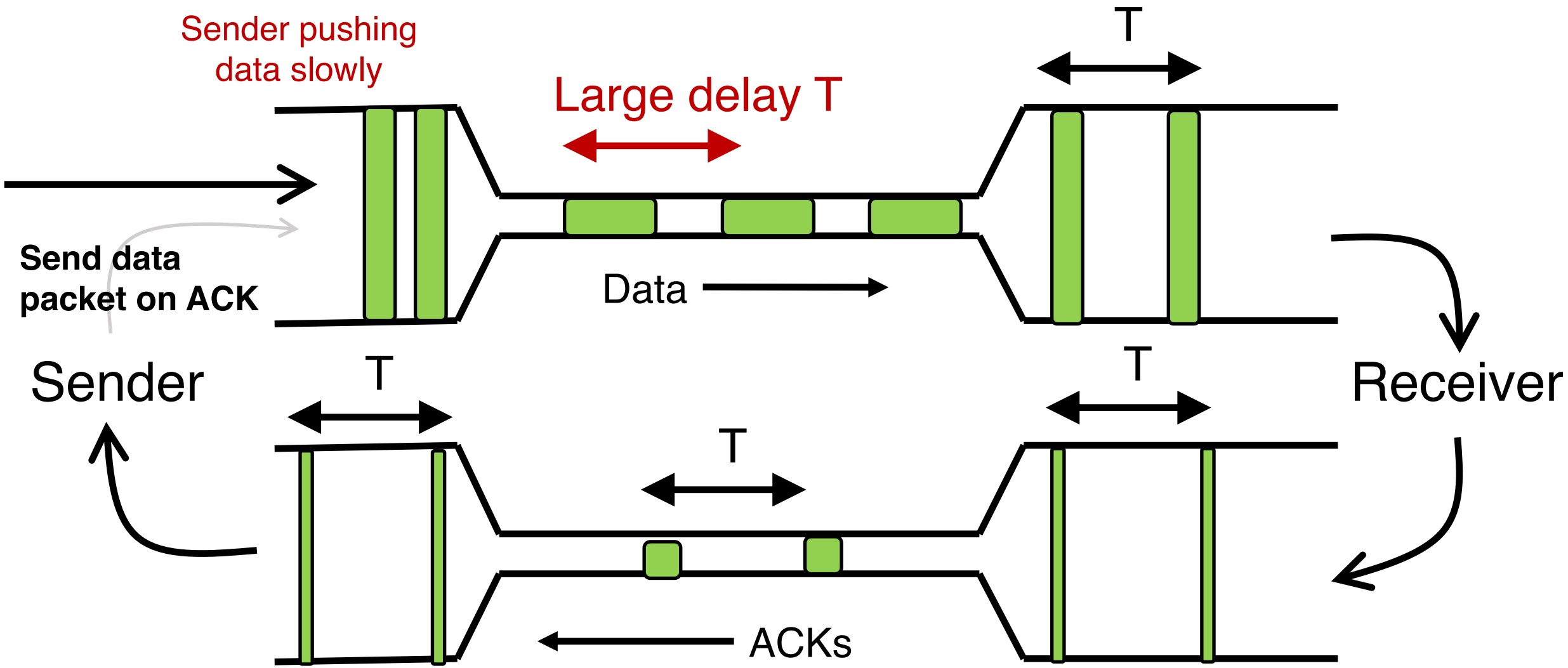
ACK clocking: analogy

- How to avoid crowding a grocery store?

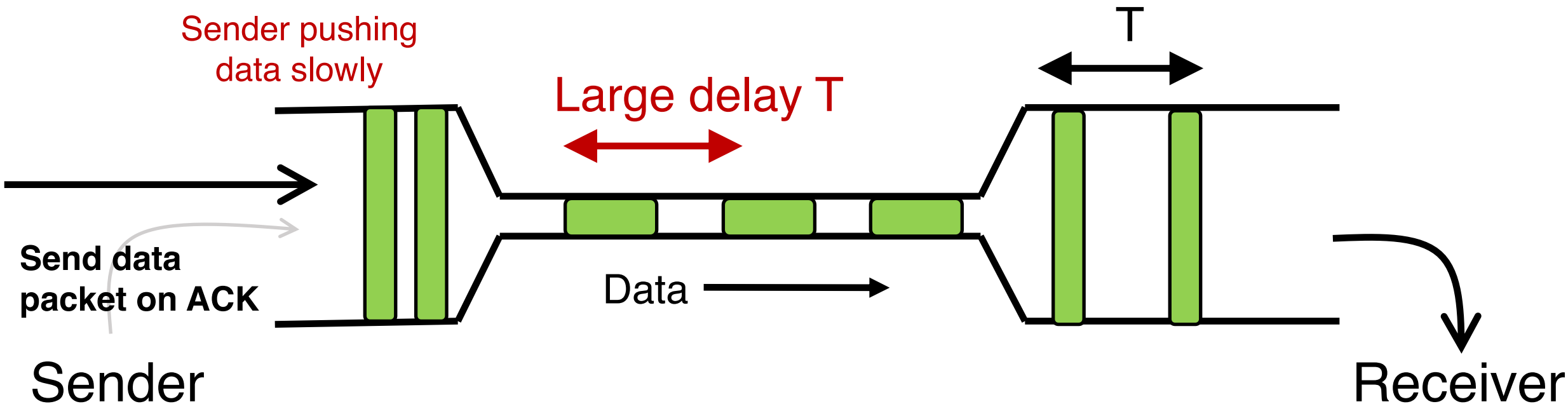
- Strategy: Send the next waiting customer exactly when a customer exits the store

- 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)

- Q: How to get to steady state? (subject of next module)

CS 352
Getting to Steady State

CS 352, Lecture 12.3

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

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

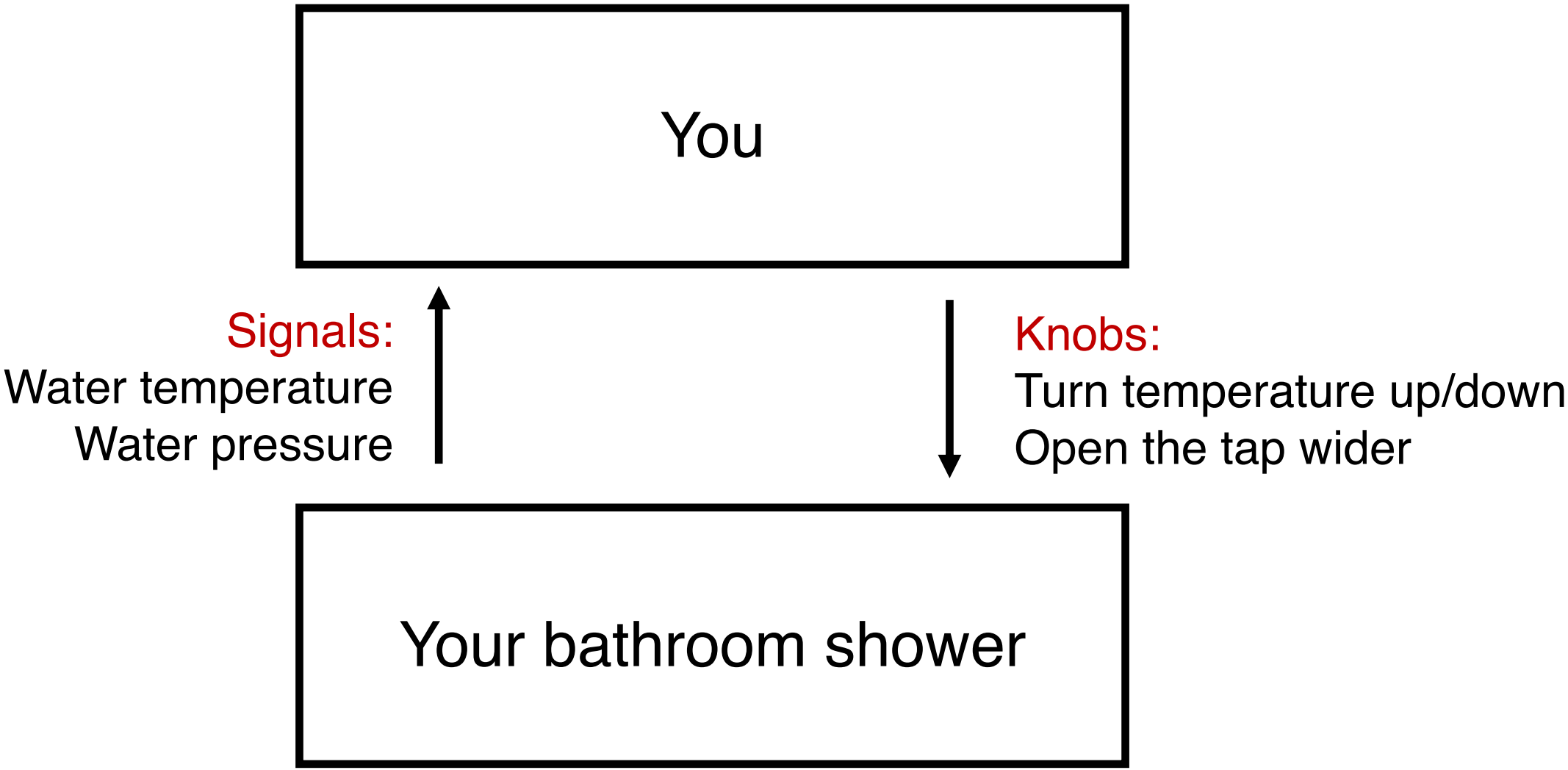
Congestion control

- Goal: at the steady state, send at the highest rate possible (to keep the pipe full) while being ACK-clocked (to avoid congesting the pipe)

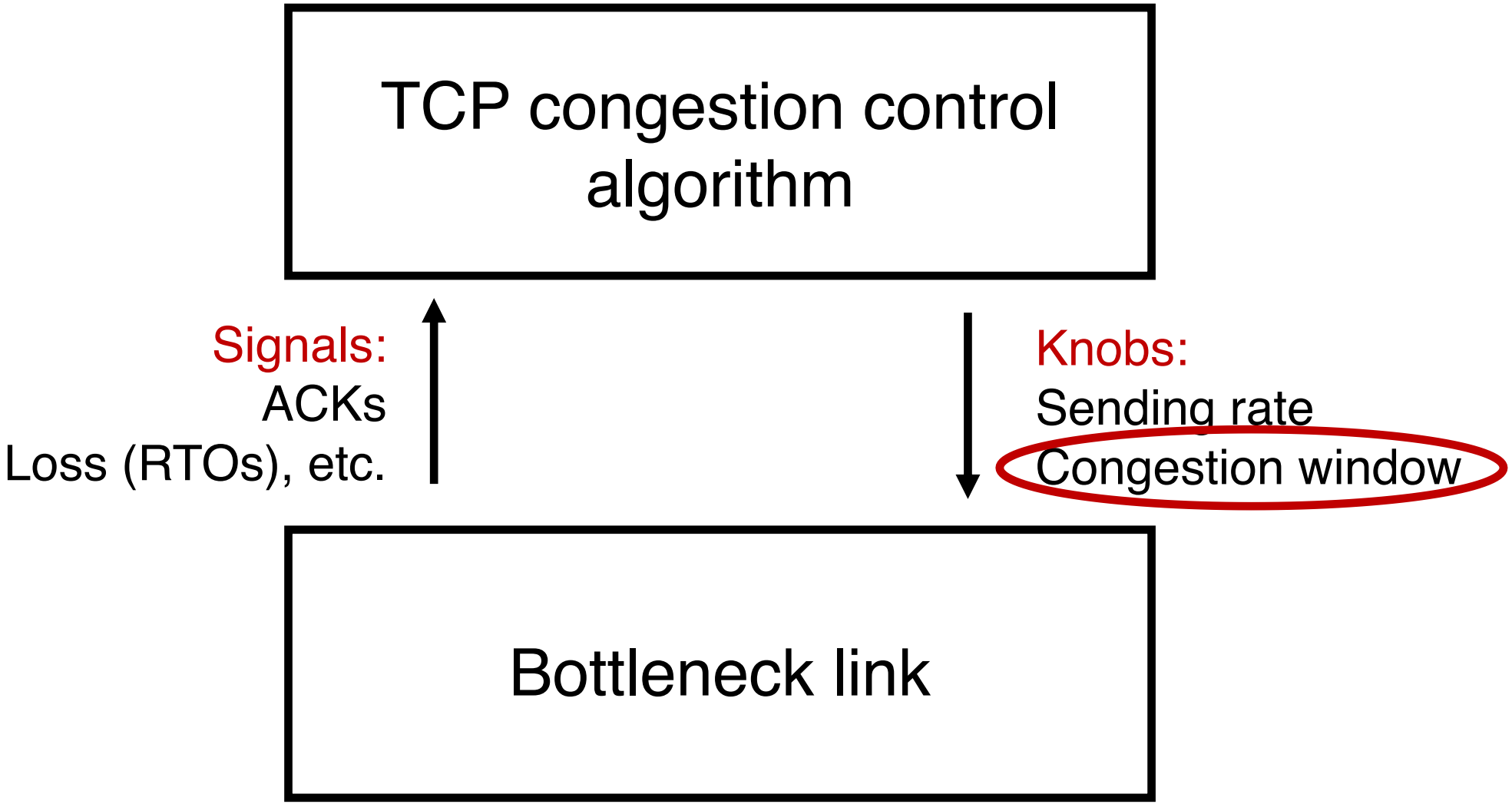
- So, how to get to steady state?

- TCP uses a feedback loop

An example of a feedback loop



The congestion control feedback loop



Congestion window

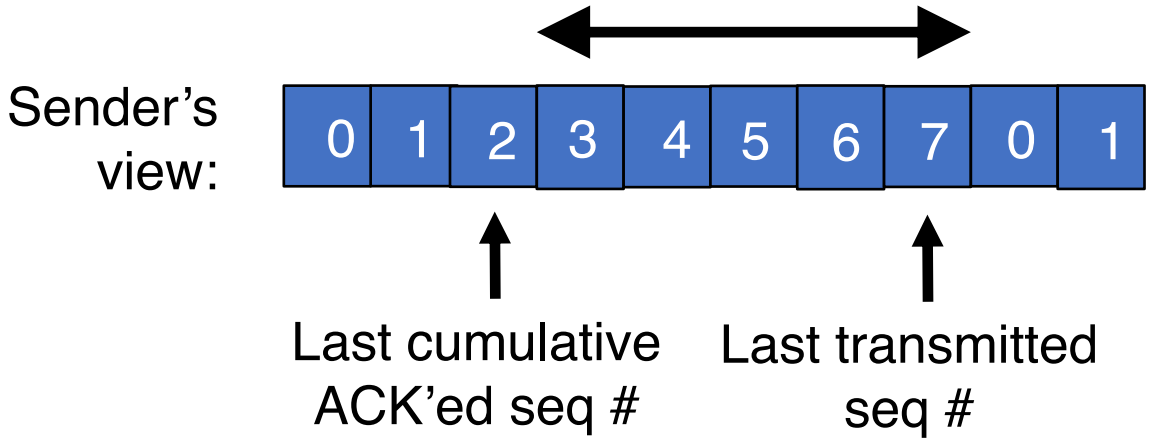
- The sender maintains an estimate of the amount of in-flight data needed to keep the pipe full without congesting it.
- This estimate is called the congestion window (cwnd)
- There is a relationship between the sending rate and the sender's window: sender transmits a window's worth of data over an RTT duration
  - rate = window / RTT

Interaction b/w flow & congestion control

- Use window = min(congestion window, receiver advertised window)
- Overwhelm neither the receiver nor network links & routers

Window <= Congestion window (congestion control)

Window <= Advertised window (flow control)



Finding the Right Congestion Window

Let’s play a game

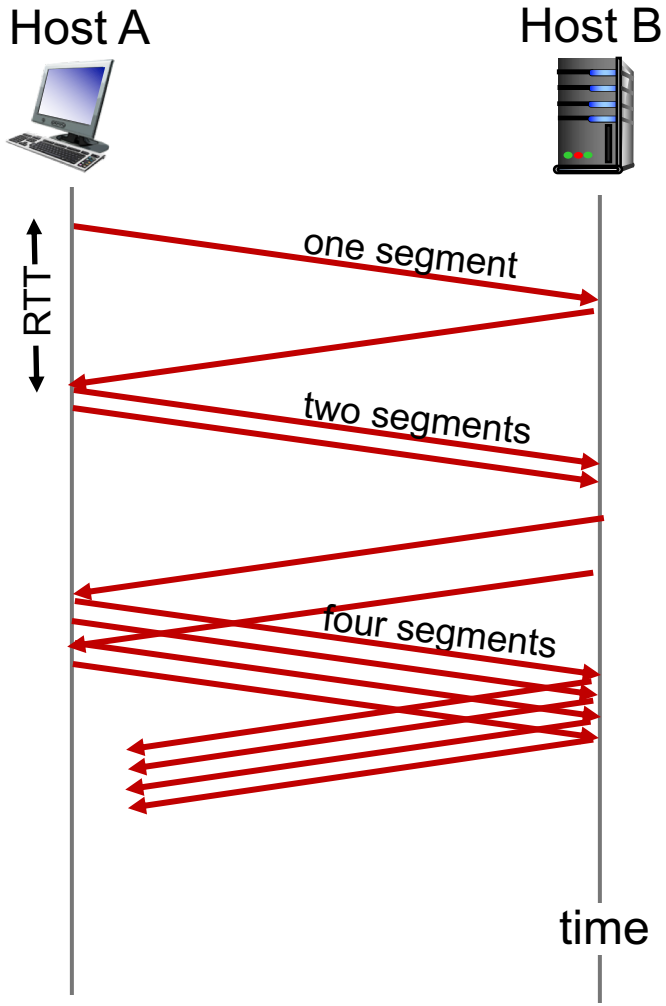
- Suppose I’m thinking of a positive integer. You need to guess the number I have in mind.
- Each time you guess, I will tell you whether your number is smaller or larger than (or the same as) the one I’m thinking of
- Note that my number can be very large
- How would you go about guessing the number?

Finding the right congestion window

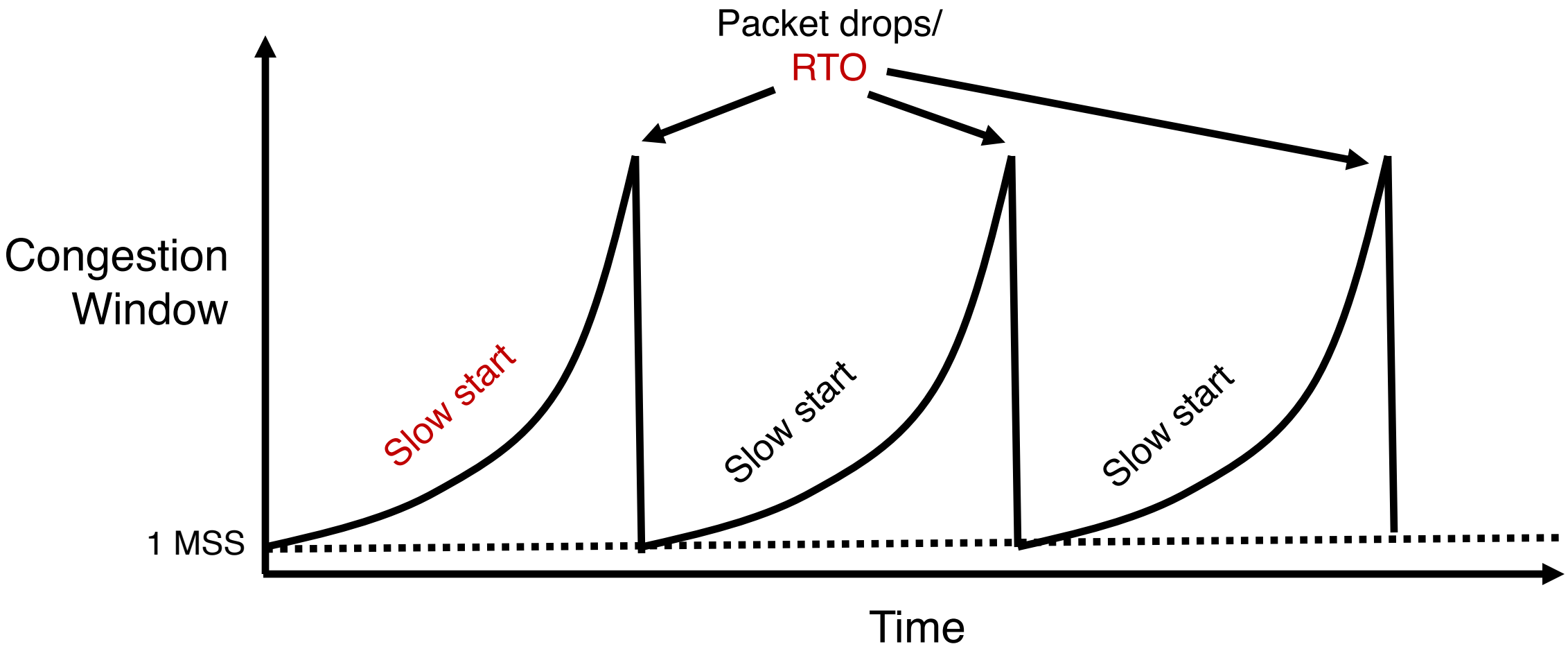
- TCP congestion control algorithms solve a similar problem!
- There is an unknown bottleneck link rate that the sender must match
- If sender sends more than the bottleneck link rate:
  - packet loss, delays, etc.
- If sender sends less than the bottleneck link rate:
  - all packets get through; successful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

- Once close enough to the bottleneck link rate, use a different set of strategies to perform smaller adjustments to `cwnd`
  - Called TCP congestion avoidance

TCP Congestion Avoidance

Two congestion control algorithms

TCP New Reno

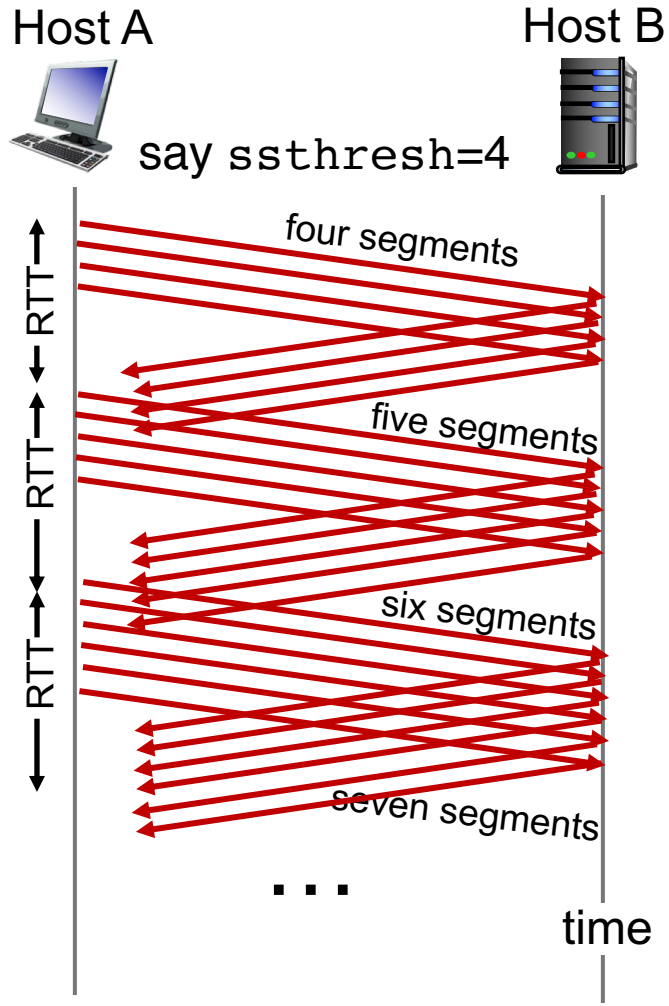
- The most studied, classic "textbook" TCP algorithm
- The primary knob is congestion window
- The primary signal is packet loss (RTO)
- Adjustment using additive increase

TCP BBR

- Recent algorithm developed & deployed by Google
- The primary knob is sending rate
- The primary signal is rate of incoming ACKs
- Adjustment using gain cycling and filters

TCP New Reno: Additive Increase

- Remember the recent past to find a good estimate of link rate
- The last good `cwnd` without packet drop is a good indicator
  - TCP New Reno calls this the slow start threshold (`ssthresh`)
- Increase `cwnd` by 1 MSS every RTT after `cwnd` hits `ssthresh`
  - Effect: increase window additively per RTT



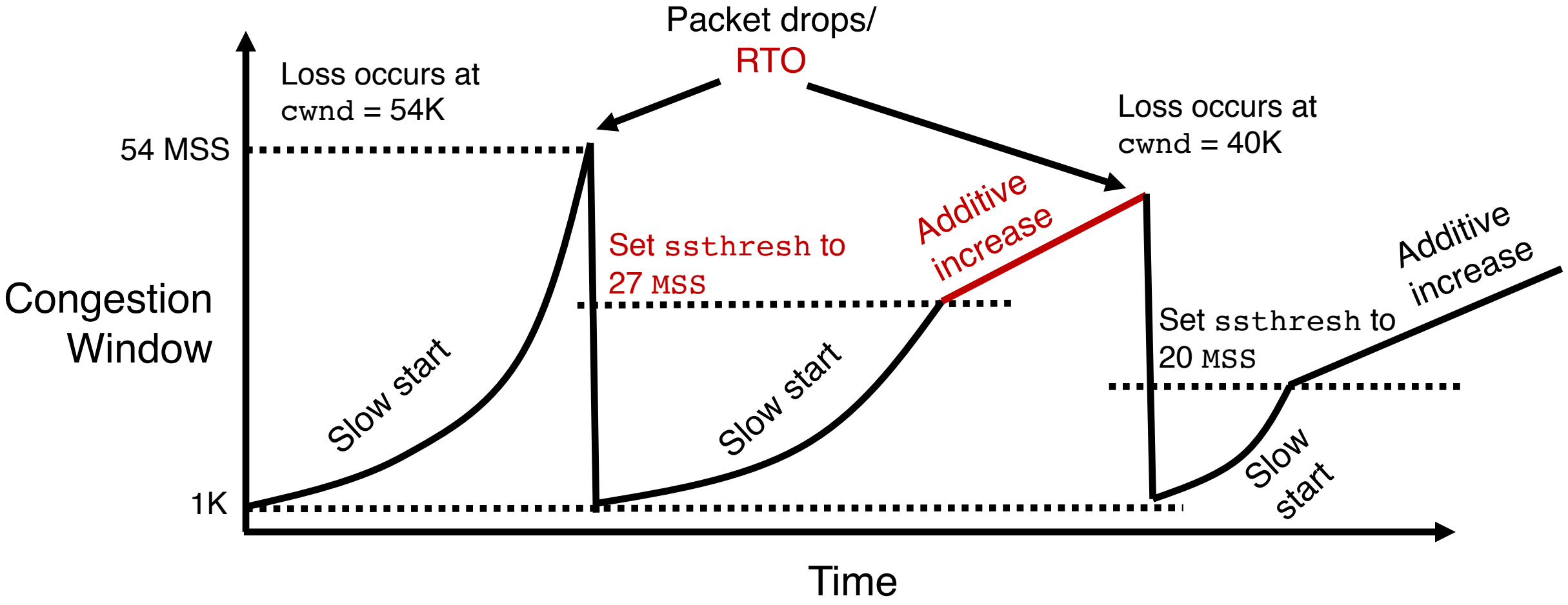
TCP New Reno: Additive increase

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

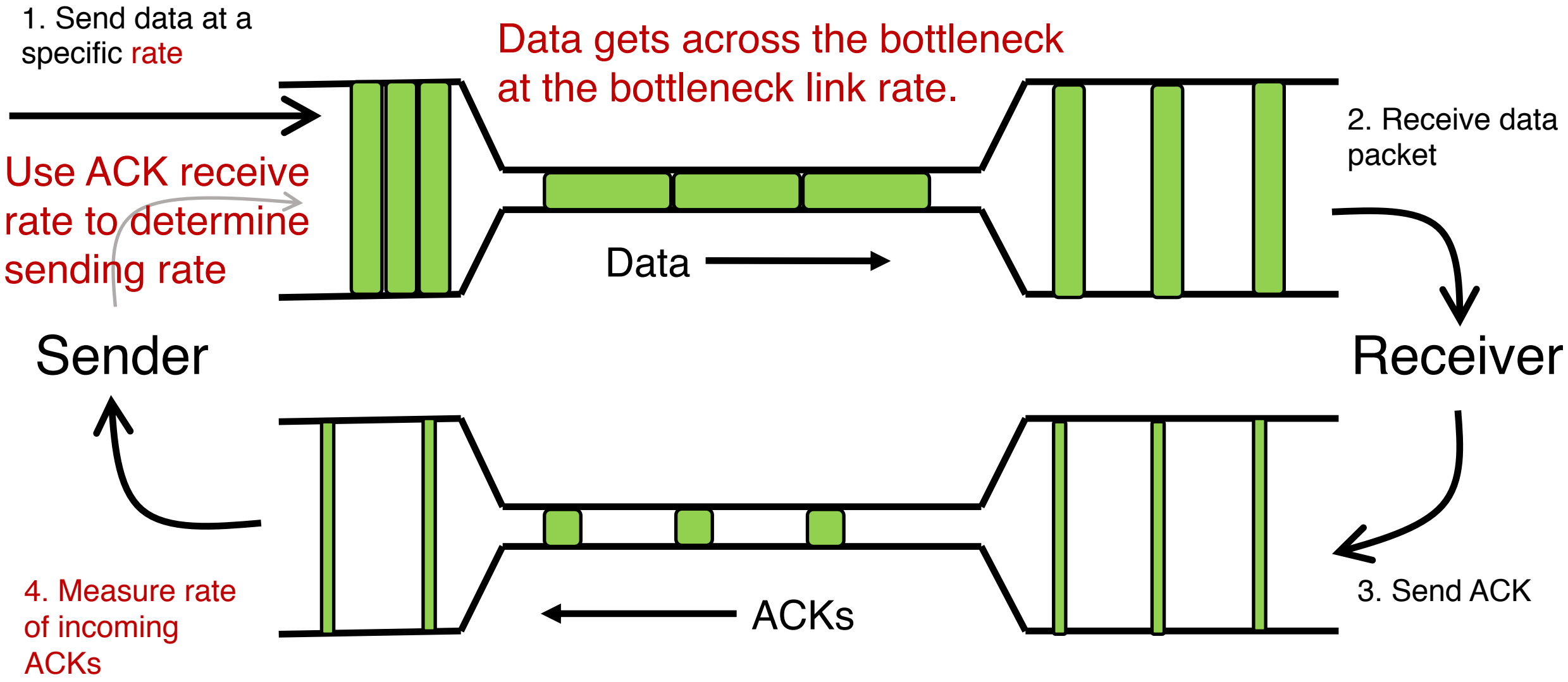
Behavior of Additive Increase

Say `MSS` = 1 KByte

Default `ssthresh` = 64KB = 64 `MSS`



TCP BBR: finding the bottleneck link rate



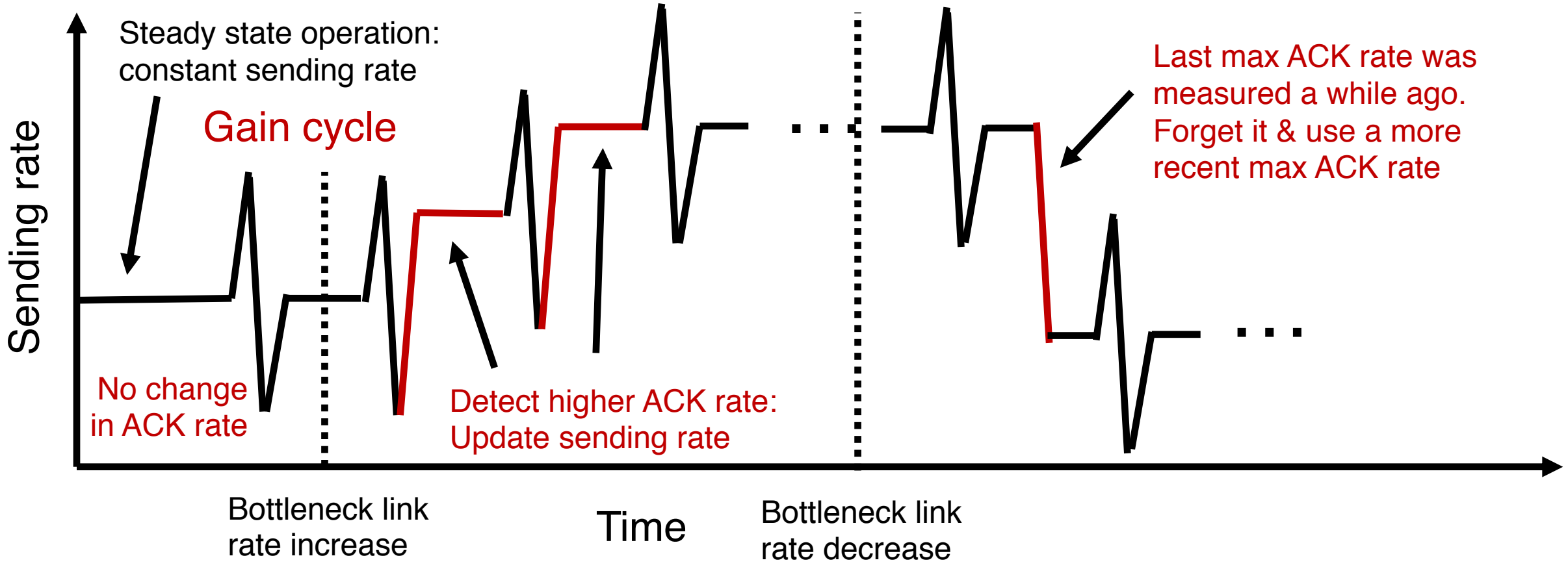
TCP BBR: finding the bottleneck link rate

- Assuming that the link rate of the bottleneck
  - == the rate of data getting across the bottleneck link
  - == the rate of data getting to the receiver
  - == the rate at which ACKs are generated by the receiver
  - == the rate at which ACKs reach the sender
- Measuring ACK rate provides an estimate of bottleneck link rate

- BBR: Send at the maximum ACK rate measured in the recent past
  - Update max with new bottleneck rate estimates, i.e., larger ACK rate
  - Forget estimates last measured a long time ago
  - Incorporated into a rate filter

TCP BBR: Adjustments by gain cycling

- BBR periodically increases its sending rate by a gain factor to see if the link rate has increased (e.g., due to a path change)



Summary: Getting to Steady State

- Want to get to highest sending rate that doesn't congest the bottleneck link

- Slow start: Exponential increase towards a reasonable estimate of link rate

- Congestion avoidance: milder adjustments to get close to correct link rate estimate.
- TCP New Reno: additive increase
- TCP BBR: gain cycling and filters