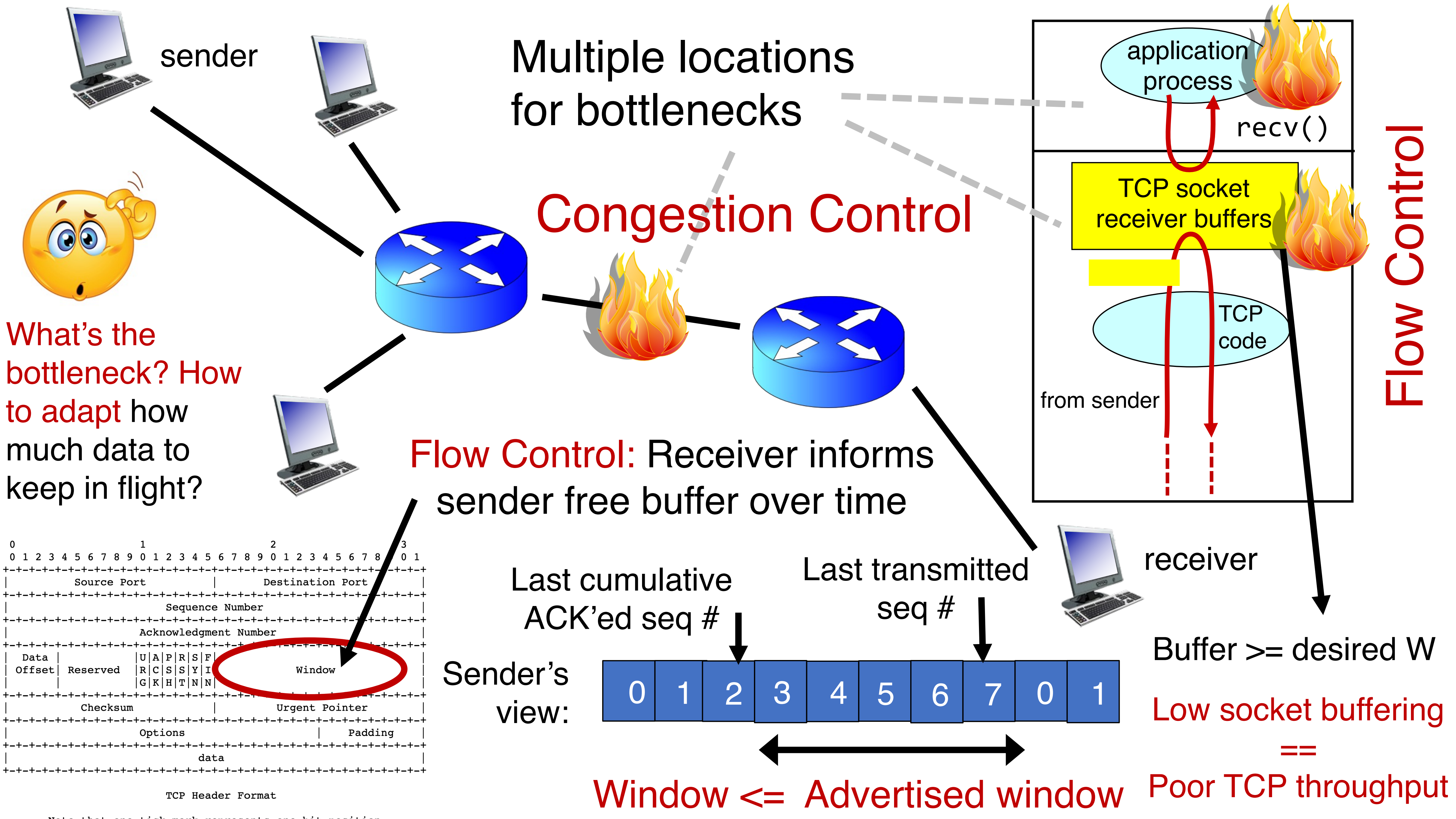
# CS 352
# Congestion Control (Part 1)

Lecture 16

http://www.cs.rutgers.edu/~sn624/352-F22


Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK




```
 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                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                    TCP Header Format

      Note that one tick mark represents one bit position.
```

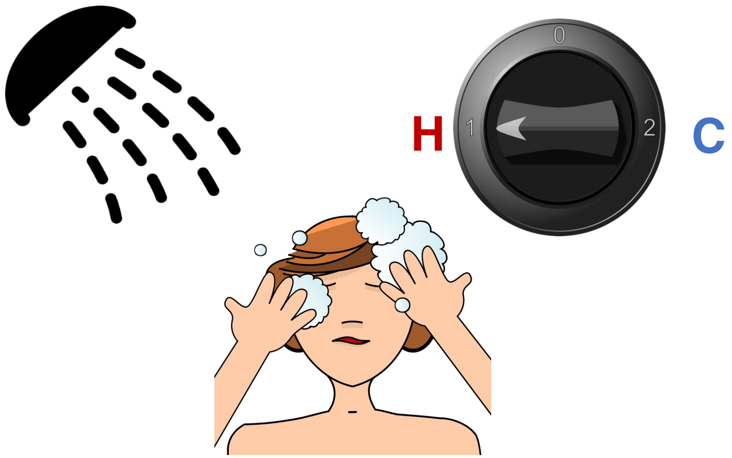
# Congestion control

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

Each endpoint acts by itself. No central vantage point or control.

Use whatever bottleneck capacity available, even with a single TCP connection.

Share bottleneck capacity equitably




**Sense and React**

# 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

# Sense and react, sure…but how?

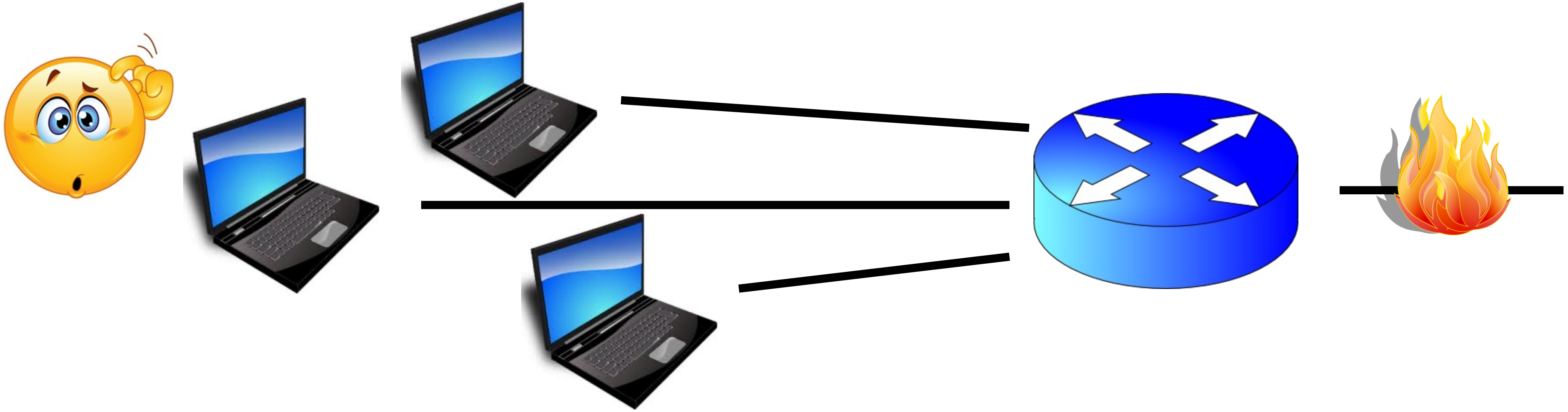
- Where do you want to be?
  - The steady state
- How do you get there?
  - Congestion control algorithms
- Sense accurately
- React proportionately

# The Steady State

## Efficiency of a single TCP 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



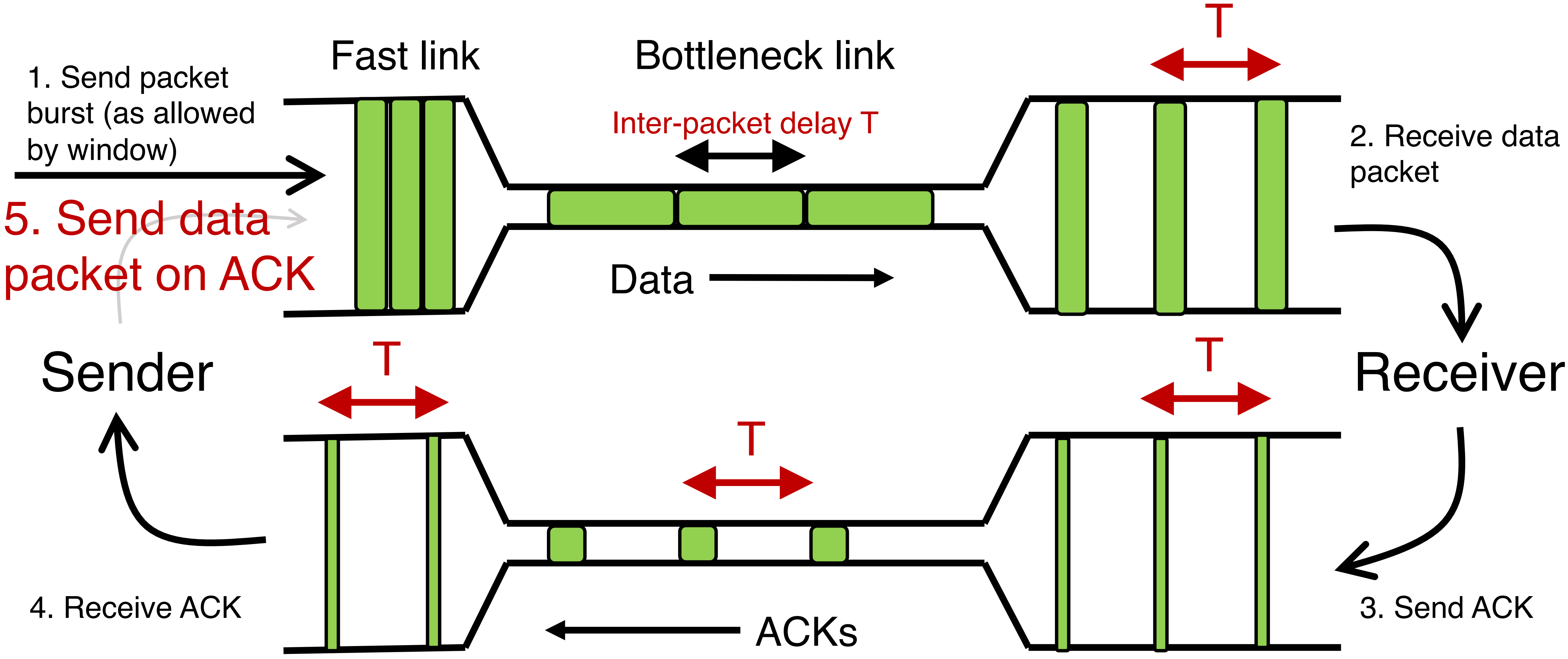
- Q: How should the endpoint behave at steady state?
- Challenge: bottleneck link is remotely located

# 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

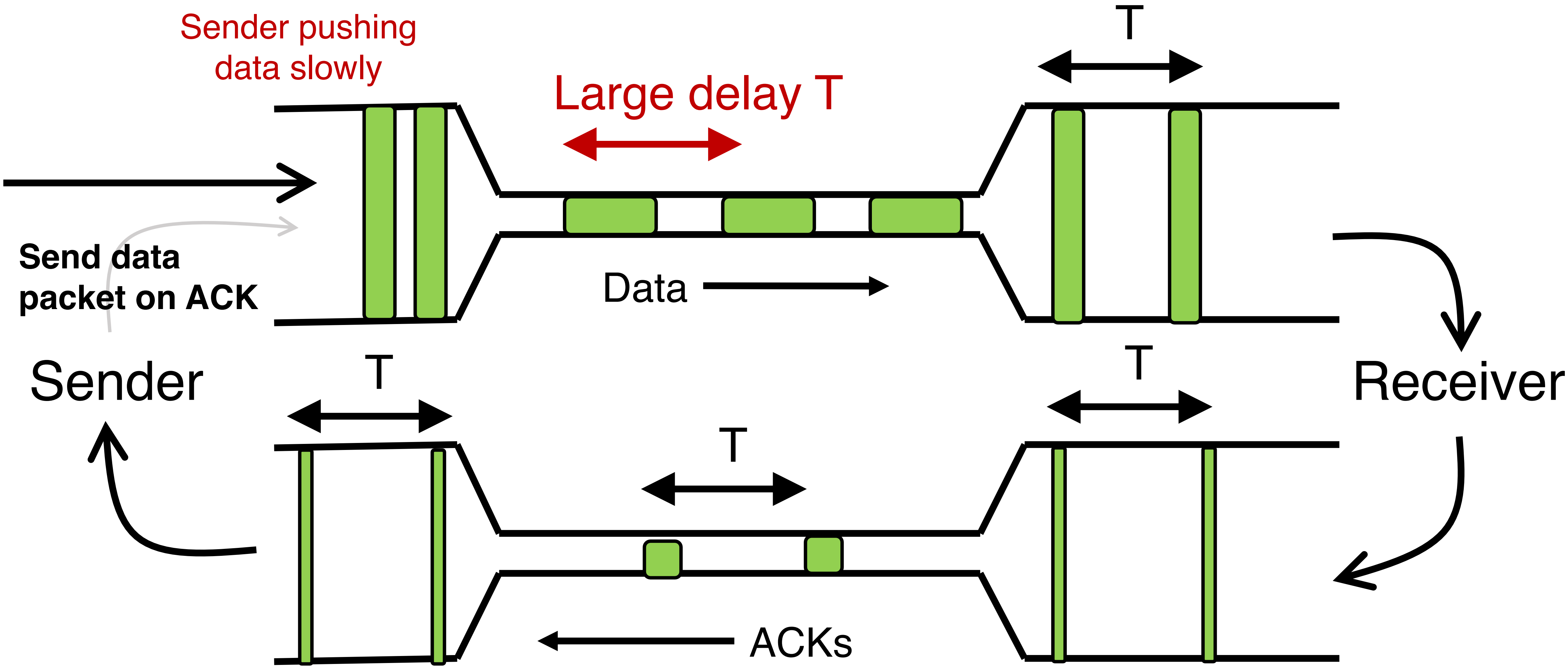
# ACK clocking: analogy

- How to avoid crowding a grocery store?

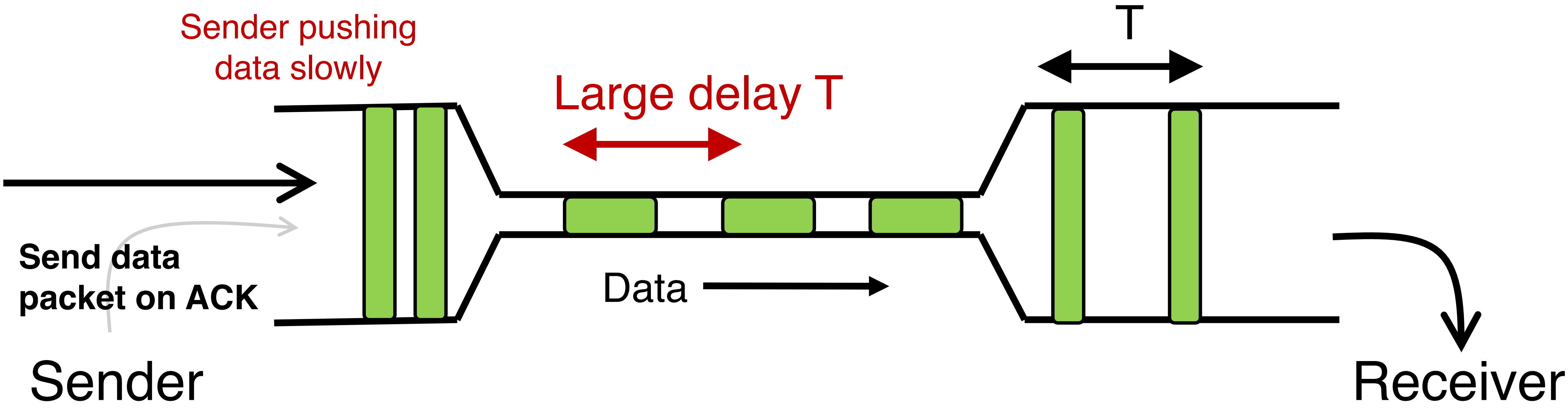
- Strategy: Send the next waiting customer exactly when a customer exits the store

- 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 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

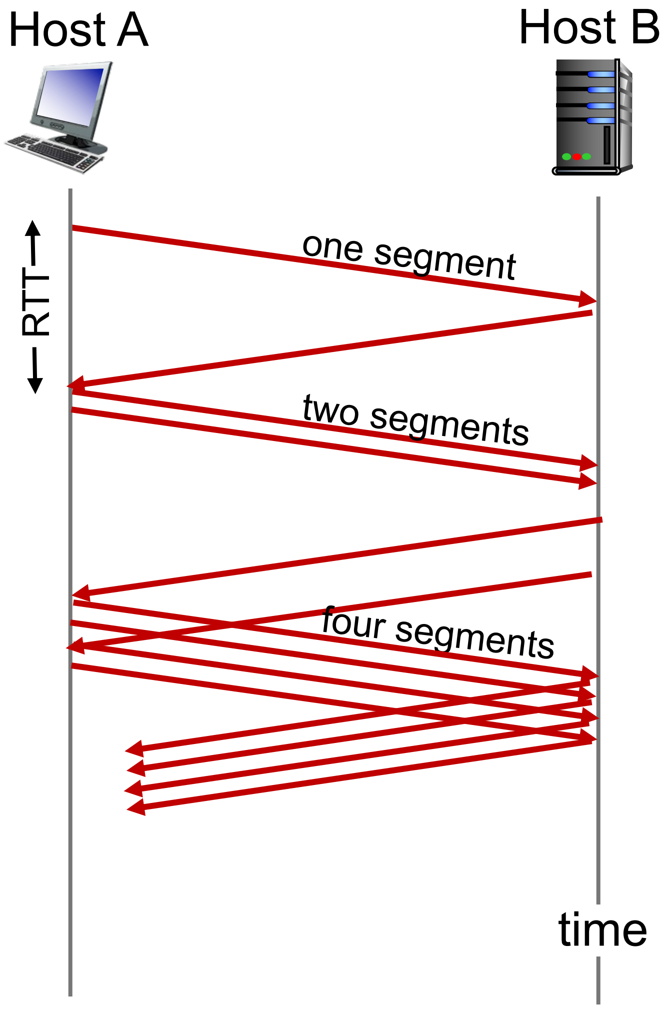
- How would you go about guessing the number?

# Finding the right congestion window

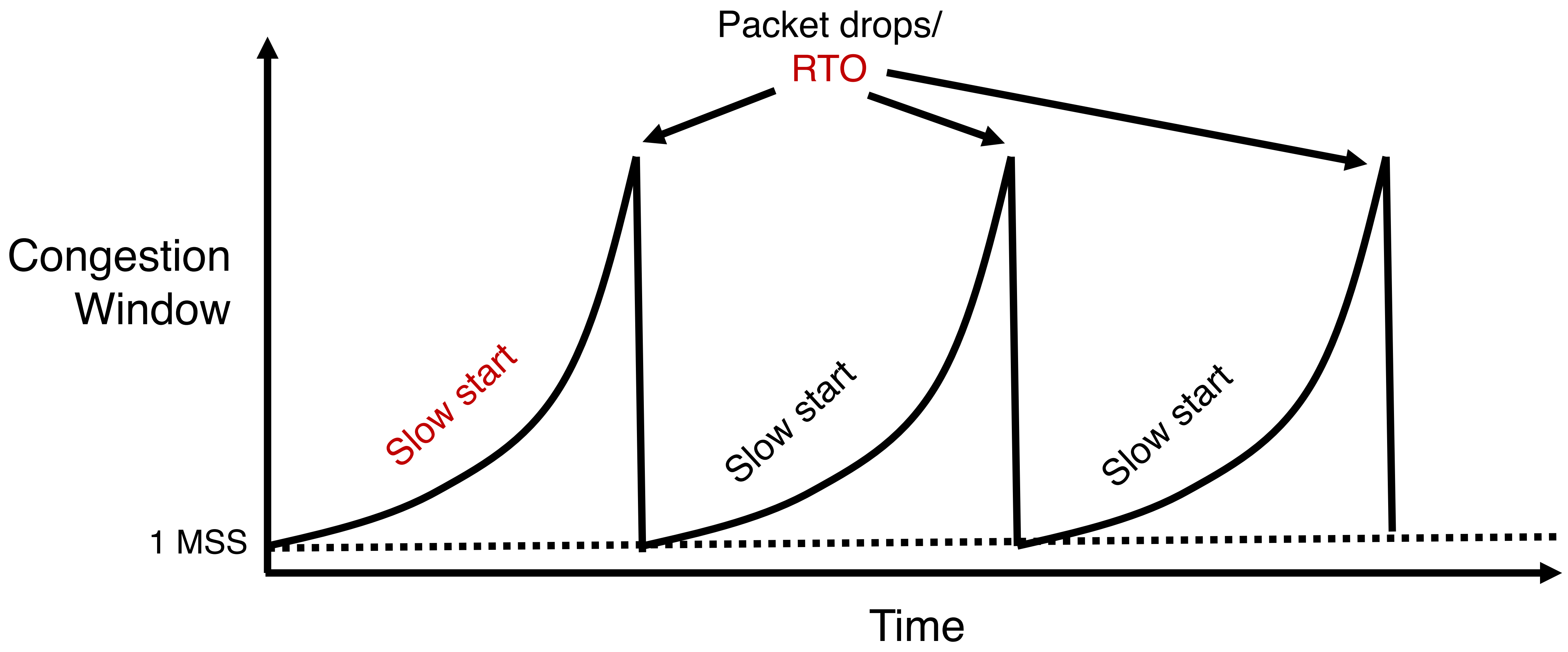
- TCP congestion control algorithms solve a similar problem!
- There is an unknown bottleneck link rate that the sender must match
- If sender sends more than the bottleneck link rate:
  - packet loss, delays, etc.
- If sender sends less than the bottleneck link rate:
  - all packets get through; successful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

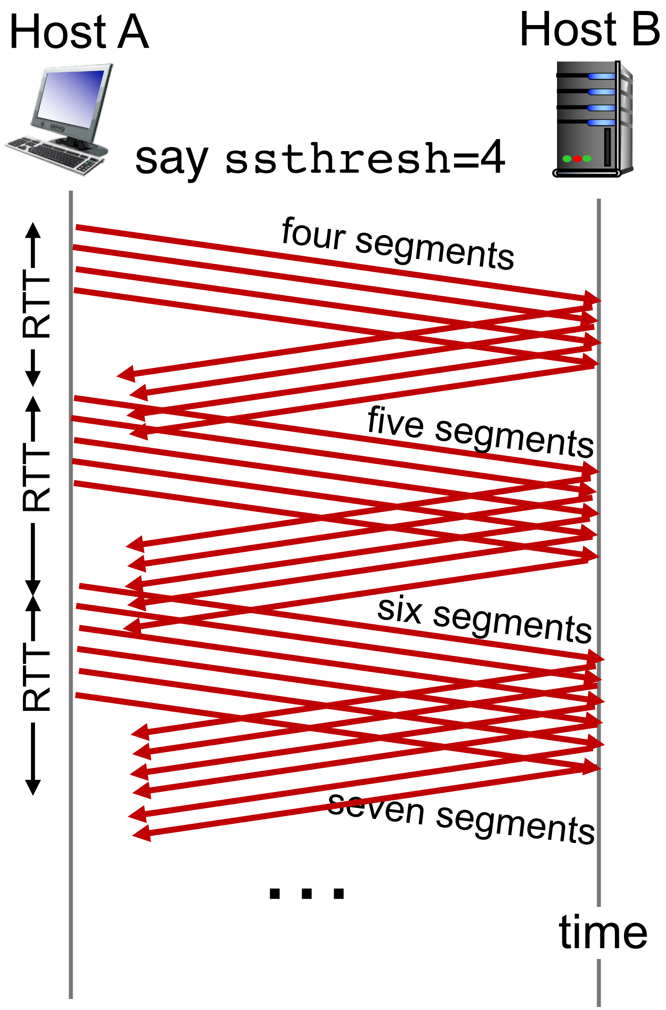
- The most studied, classic "textbook" TCP algorithm
- The primary knob is congestion window
- The primary signal is packet loss (RTO)
- Adjustment using additive increase

## 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



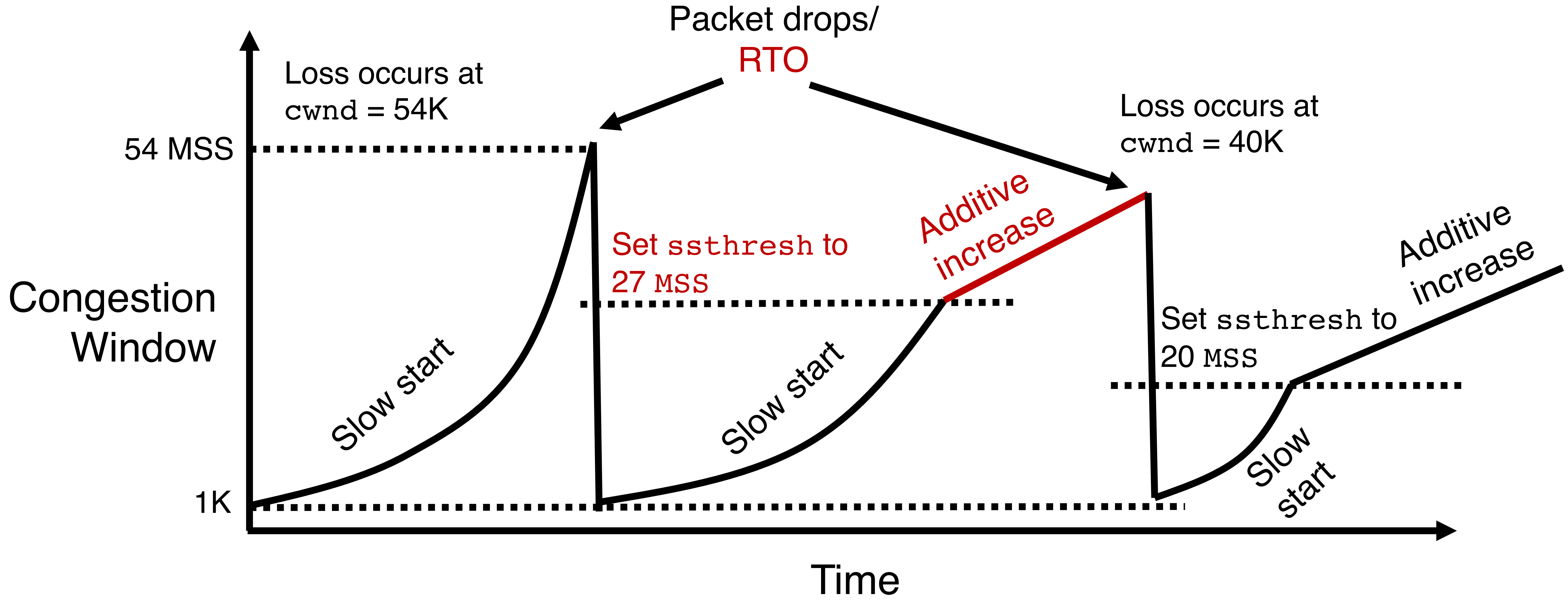
# TCP New Reno: Additive increase

- 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`

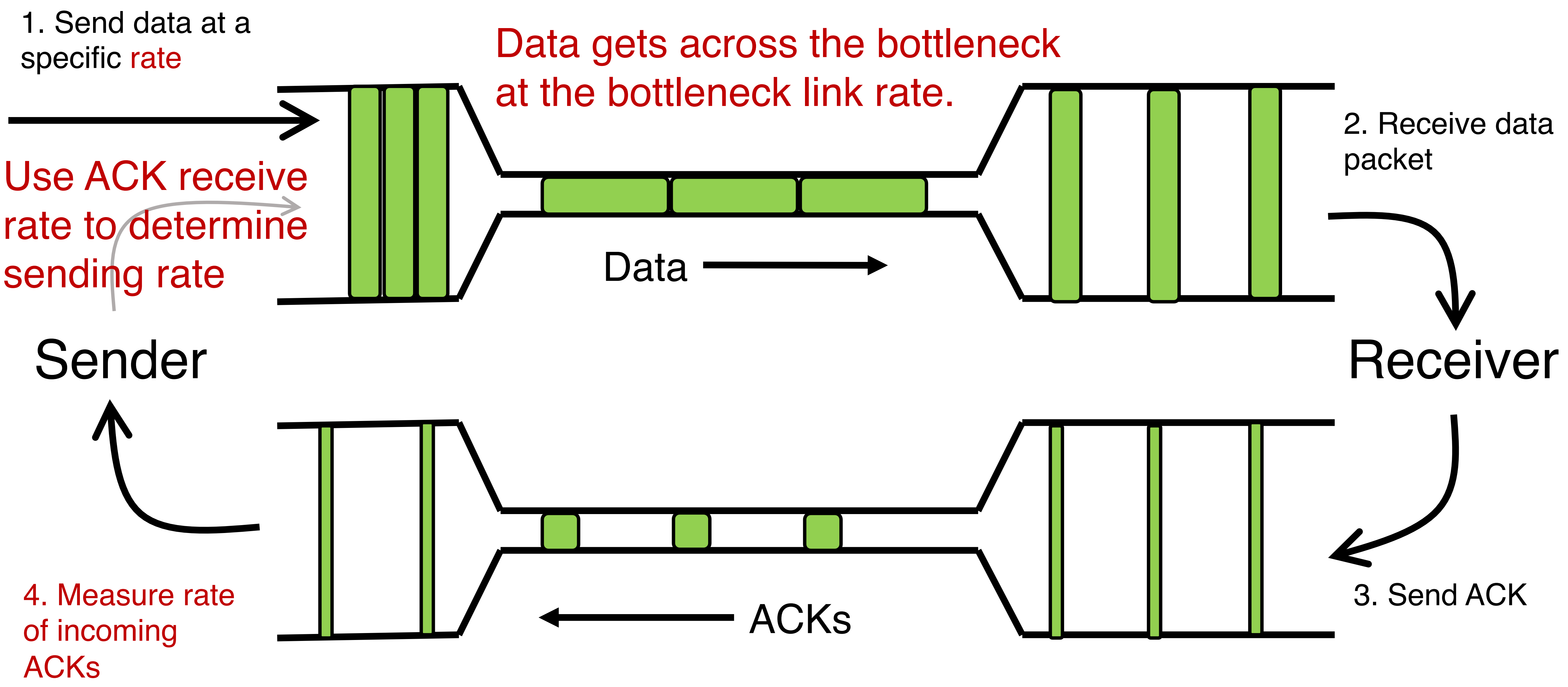
# Behavior of Additive Increase

Say `MSS` = 1 KByte

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



# TCP BBR: finding the bottleneck link rate



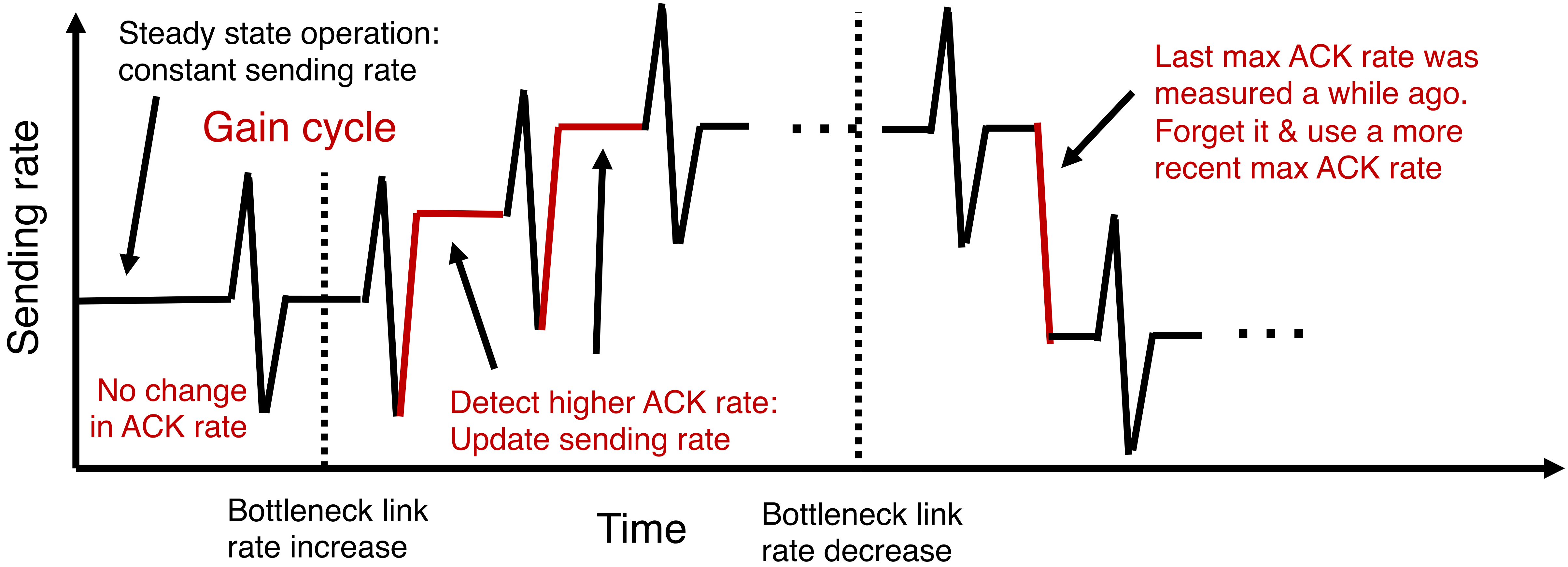
# TCP BBR: finding the bottleneck link rate

- 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