Congestion Control (Part 2)

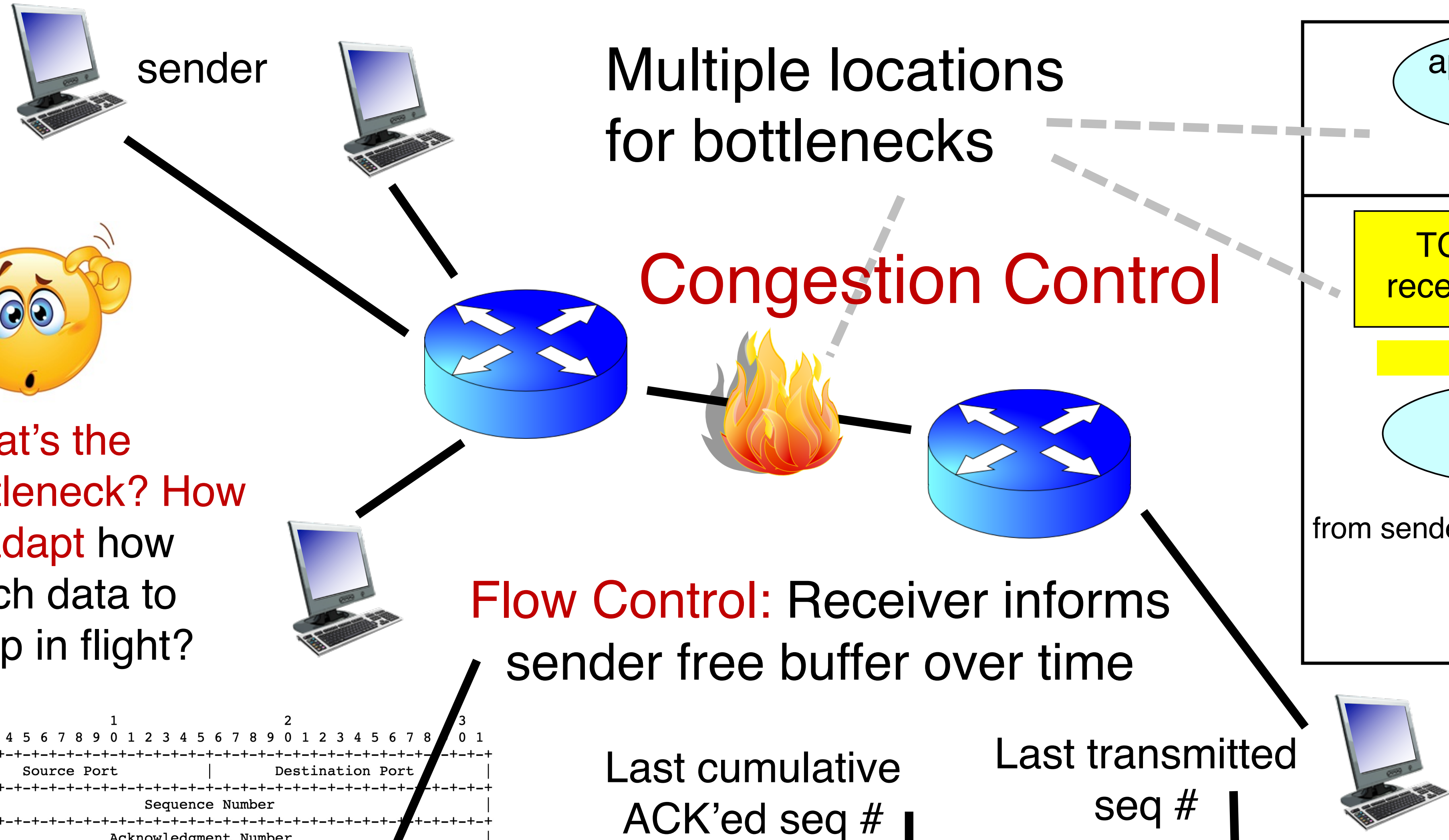
Lecture 15

http://www.cs.rutgers.edu/~sn624/352-S22

Srinivas Narayana

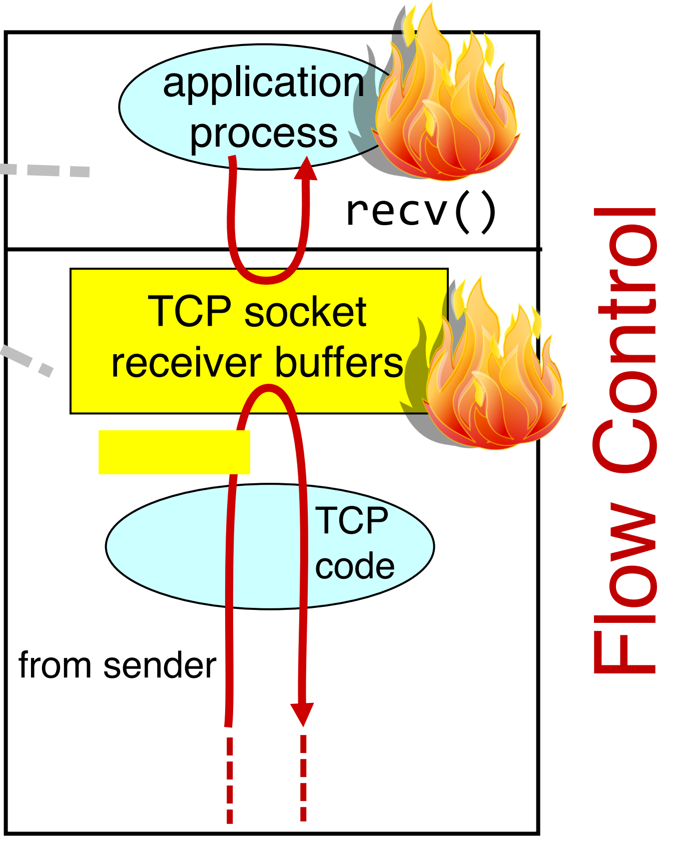
RUTGERS
UNIVERSITY | NEW BRUNSWICK

Multiple locations for bottlenecks



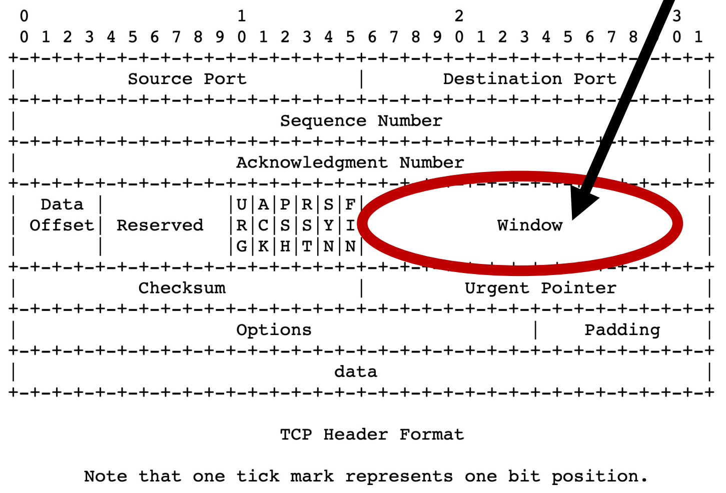
Congestion Control

What's the bottleneck? How to adapt how much data to keep in flight?



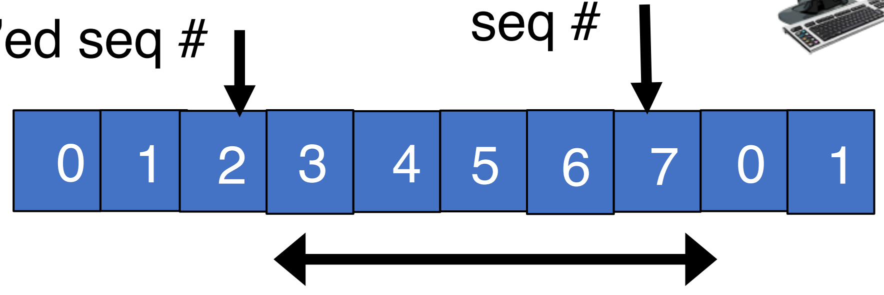
Flow Control

Flow Control: Receiver informs sender free buffer over time



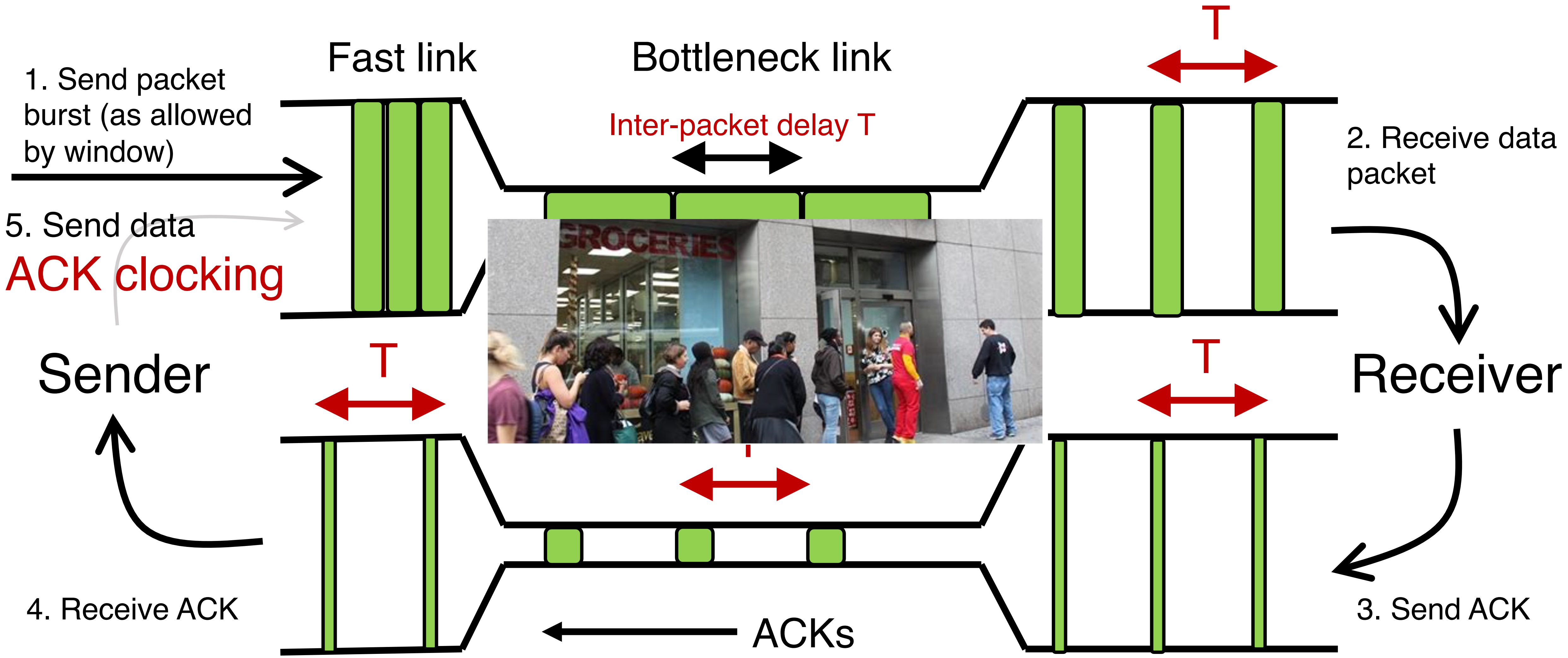
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        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.
```

Sender's view:



Window <= Advertised window

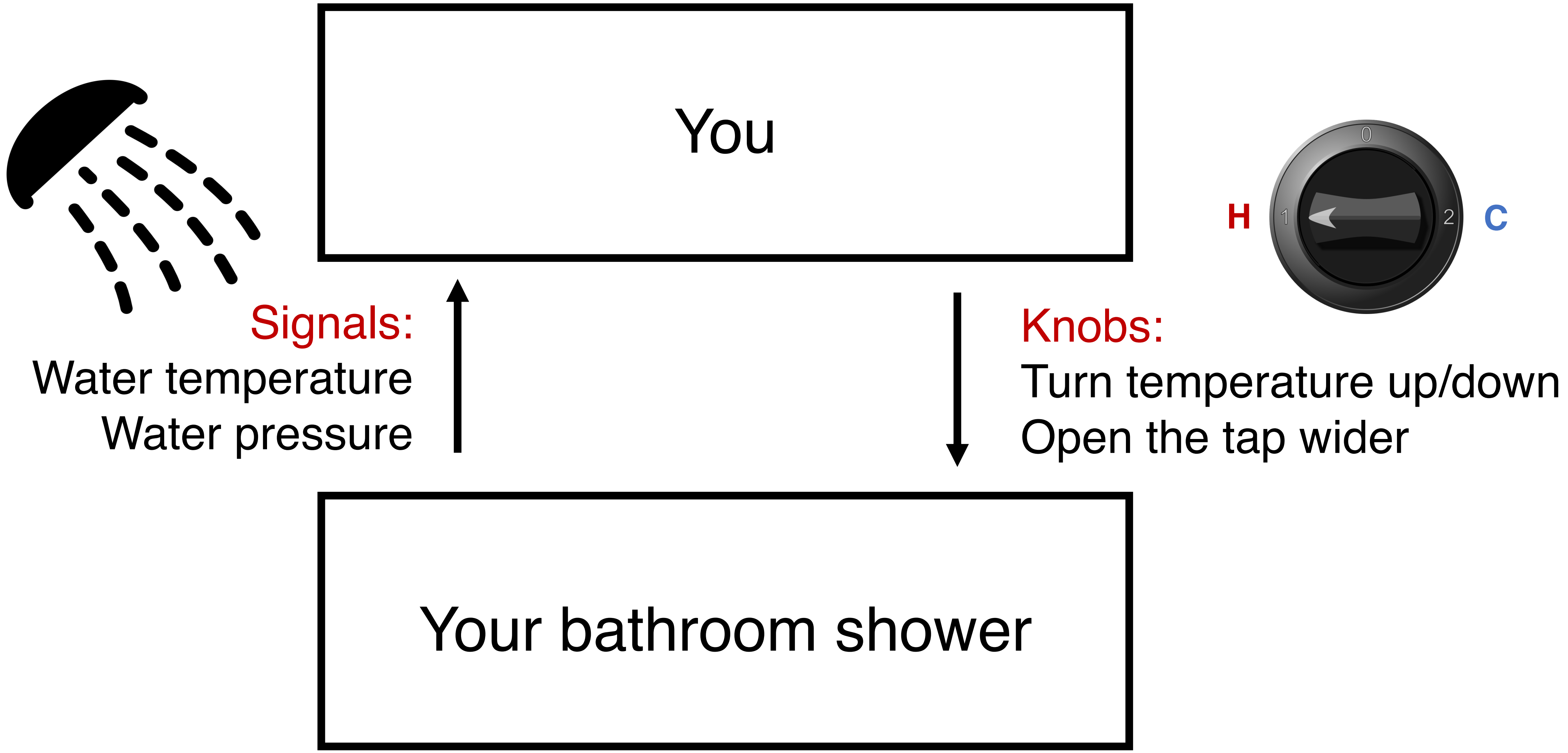
The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.



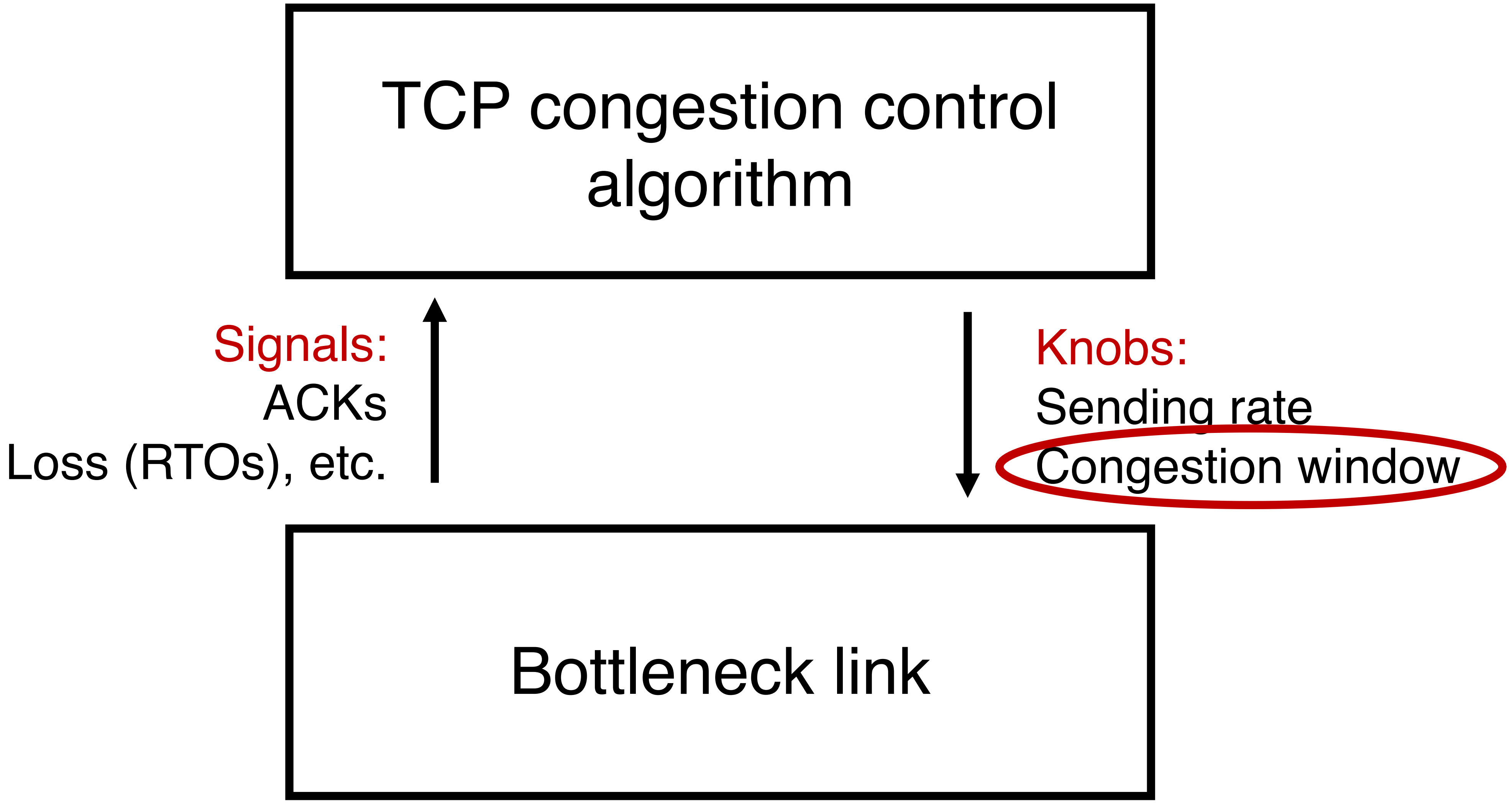
Getting to a fast ACK clock

The Feedback Loop

An example of a feedback loop



The congestion control feedback loop



Congestion window

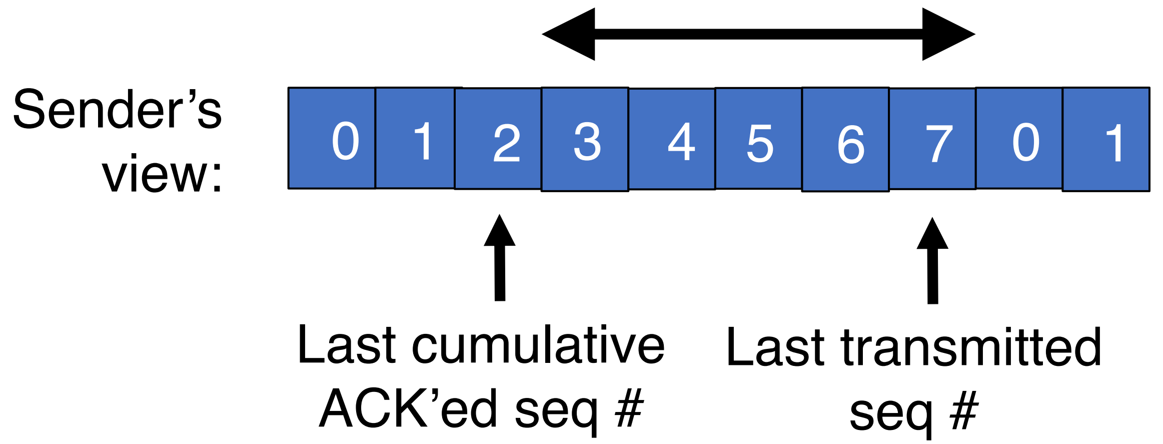
- The sender maintains an estimate of the amount of in-flight data needed to keep the pipe full without congesting it.
- This estimate is called the congestion window (cwnd)
- Recall: There is a relationship between the sending rate (throughput) and the sender’s window: sender transmits a window’s worth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

Interaction 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!

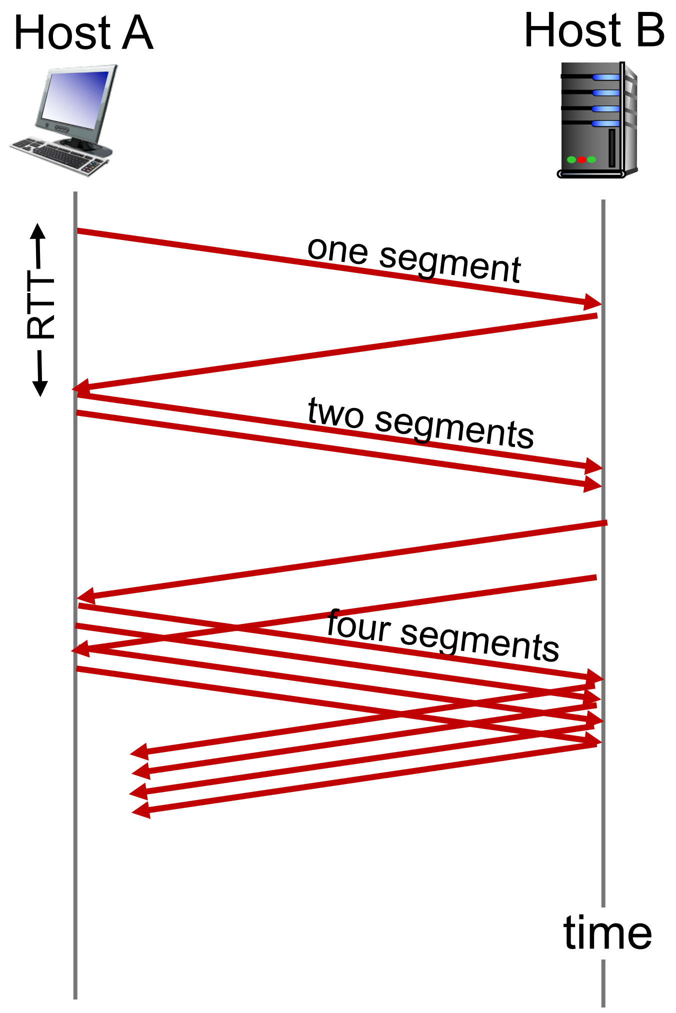
- There is an unknown bottleneck link rate that the sender must match

- If sender sends more than the bottleneck link rate:
  - packet loss, delays, etc.

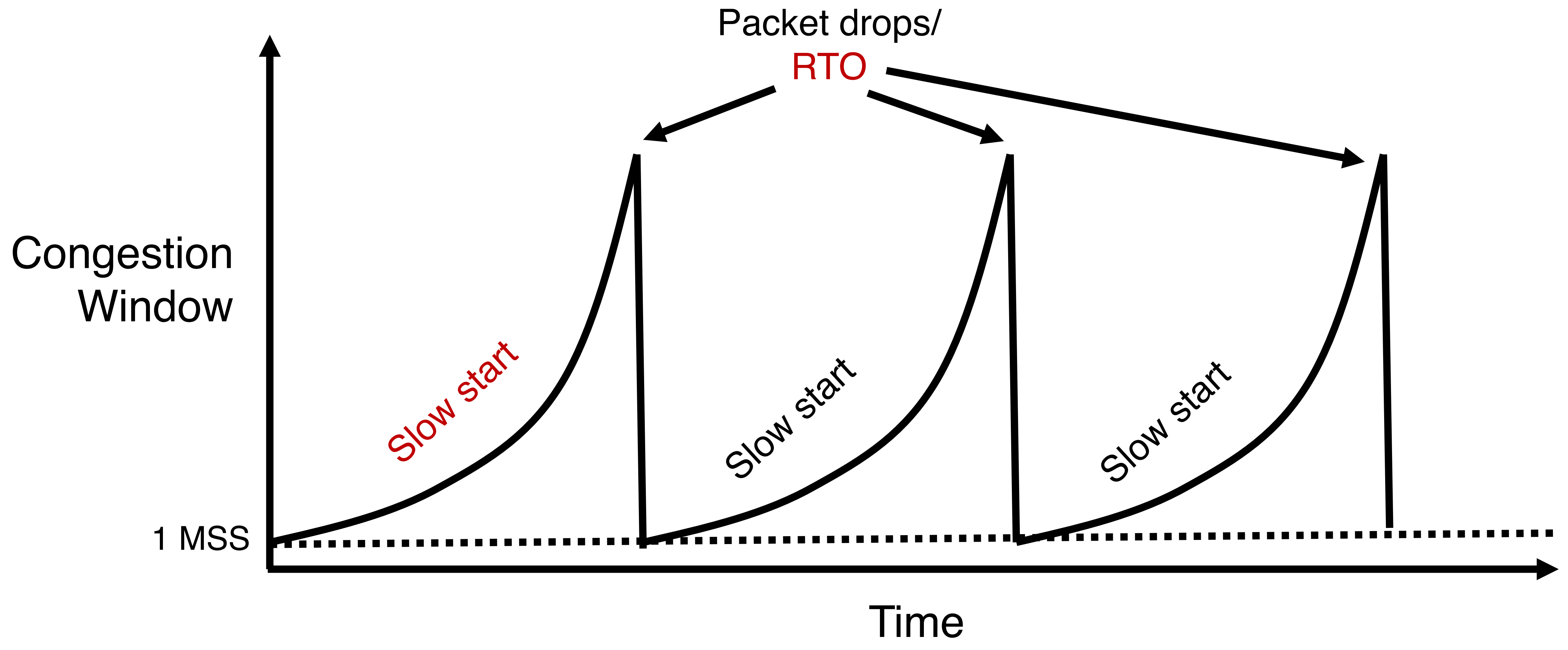
- If sender sends less than the bottleneck link rate:
  - all packets get through; successful ACKs

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

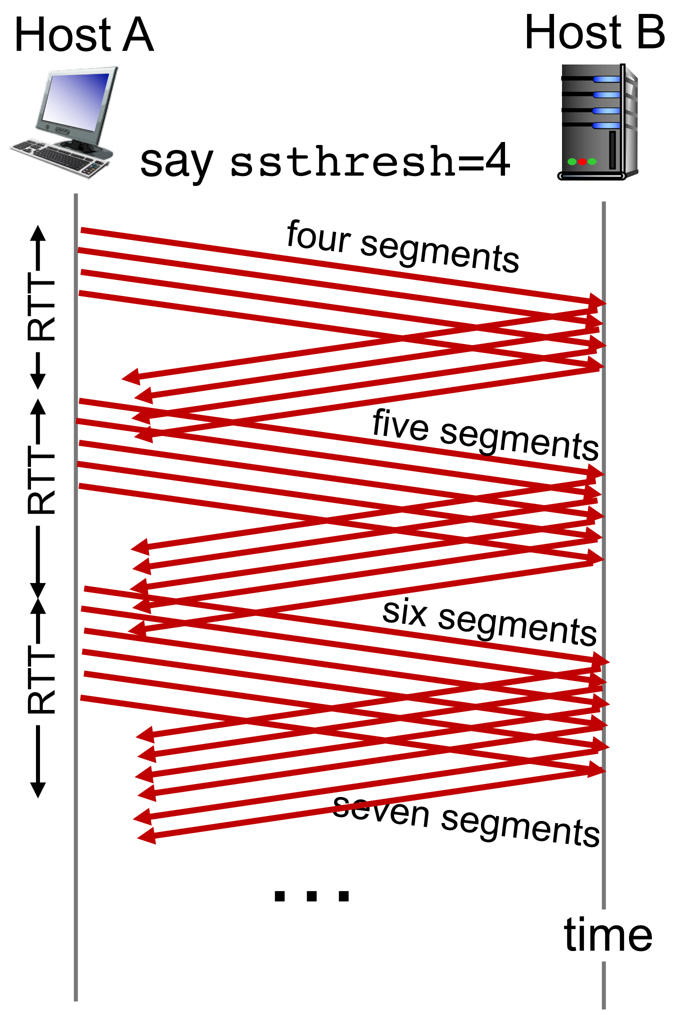
- The most studied, classic “textbook” TCP algorithm
- The primary knob is congestion window
- The primary signal is packet loss (RTO)
- Adjustment using additive increase

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



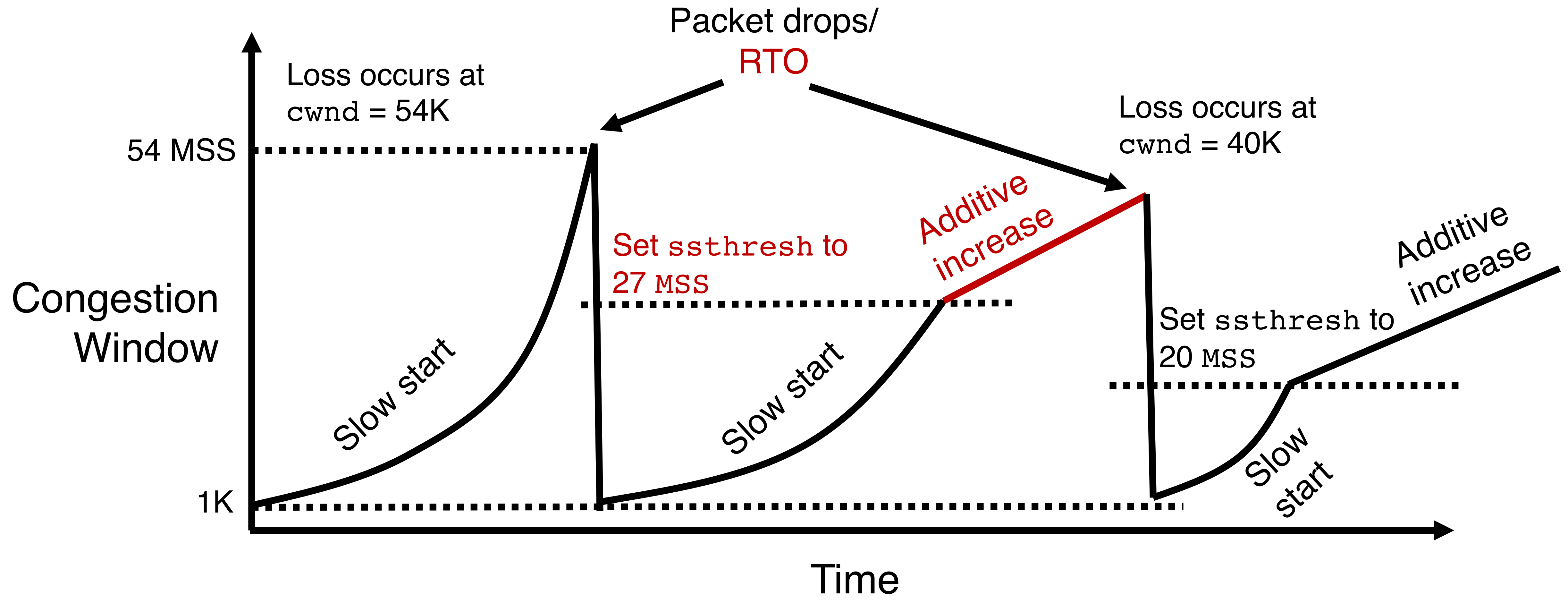
TCP New Reno: Additive increase

- 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`

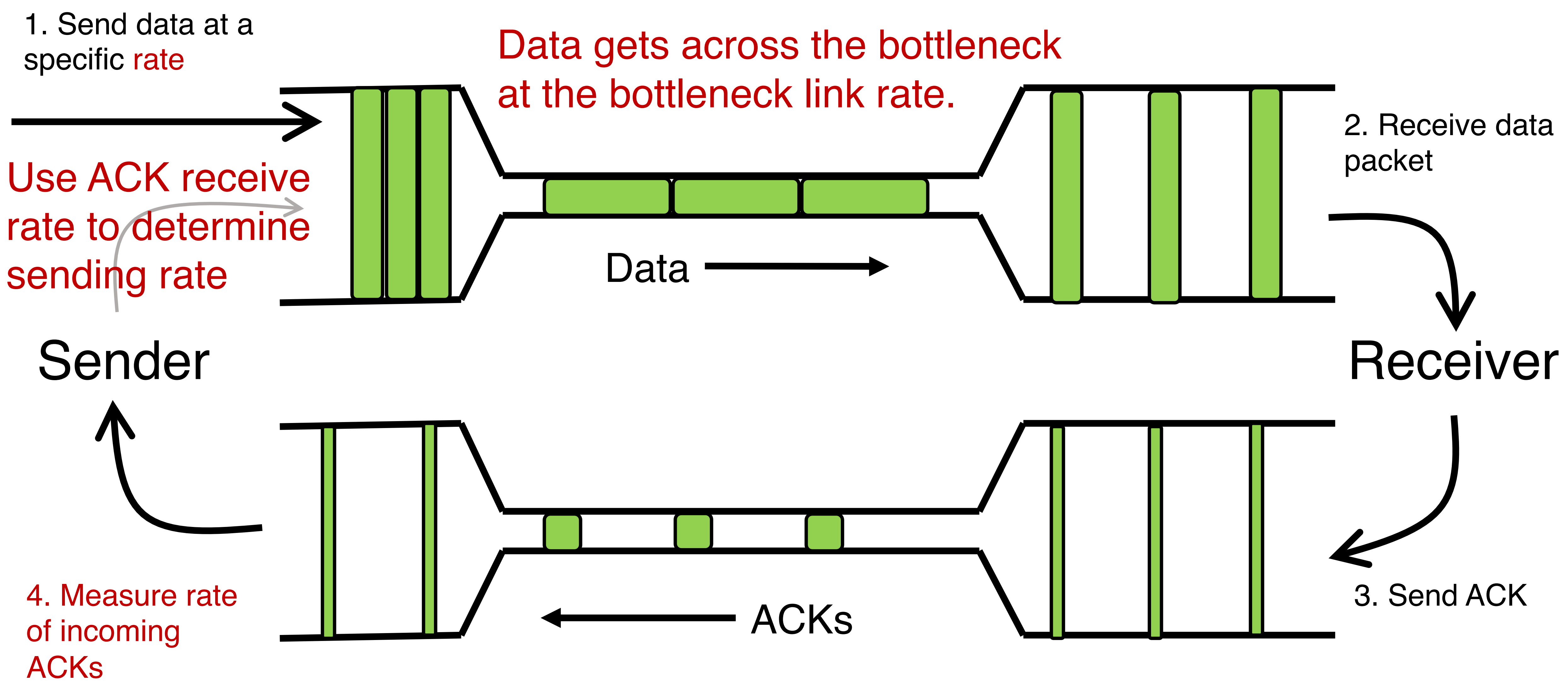
Behavior of Additive Increase

Say `MSS` = 1 KByte

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



TCP BBR: finding the bottleneck link rate


1. Send data at a specific rate
Data gets across the bottleneck at the bottleneck link rate.
2. Receive data packet
Use ACK receive rate to determine sending rate
Data
Sender
Receiver
3. Send ACK
ACKs
4. Measure rate of incoming ACKs

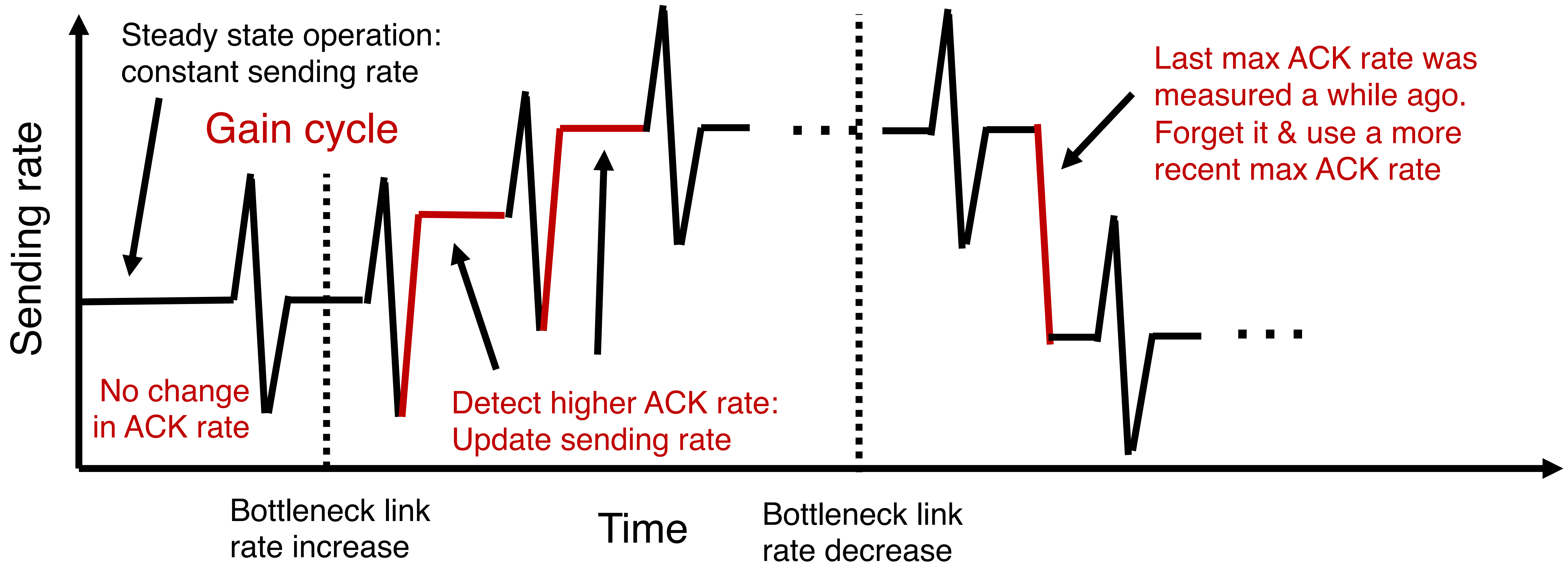
TCP BBR: finding the bottleneck link rate

- 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

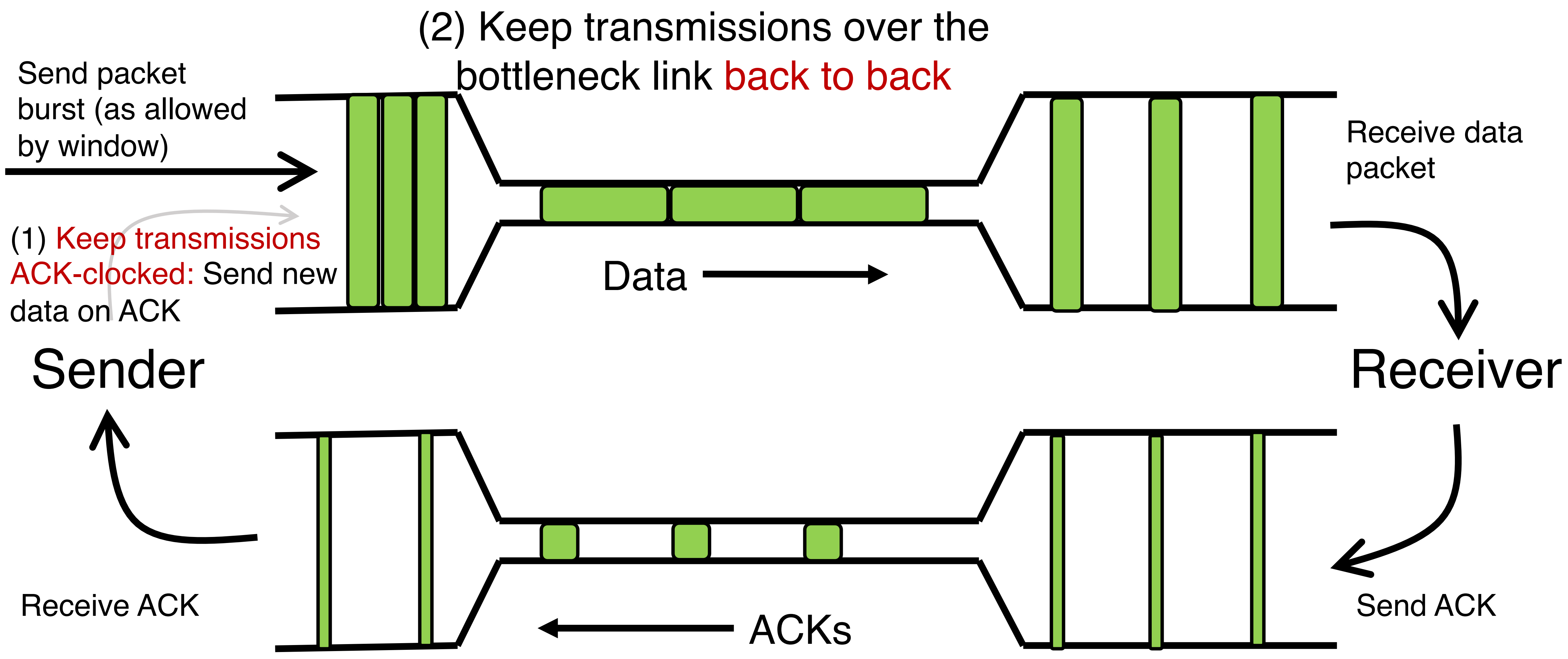
- Want to get to highest sending rate that doesn't congest the bottleneck link

- 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

Bandwidth-Delay Product

Goal of steady state operation


(2) Keep transmissions over the bottleneck link back to back
Send packet burst (as allowed by window)
(1) Keep transmissions ACK-clocked: Send new data on ACK
Sender
Data
Receive data packet
Receiver
Receive ACK
ACKs
Send ACK

Steady state `cwnd` for a single flow

- Suppose the bottleneck link has rate C
- Suppose the propagation round-trip delay (propRTT) between sender and receiver is T
- Ignore transmission delays for this example;
- Assume steady state: highest sending rate with no bottleneck congestion

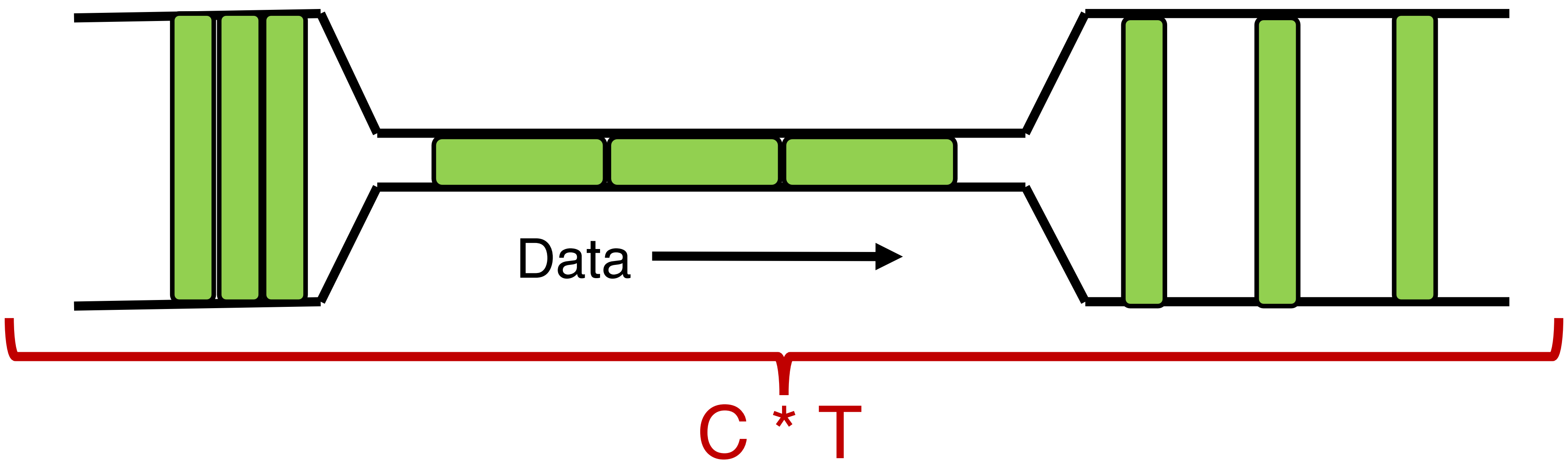
- Q: how much data is in flight over a single RTT?

- C * T data i.e., amount of data unACKed at any point in time
- ACKs take time T to arrive (without any queueing). In the meantime, sender is transmitting at rate C

The Bandwidth-Delay Product

- C * T = bandwidth-delay product:
  - The amount of data in flight for a sender transmitting at the ideal rate during the ideal round-trip delay of a packet

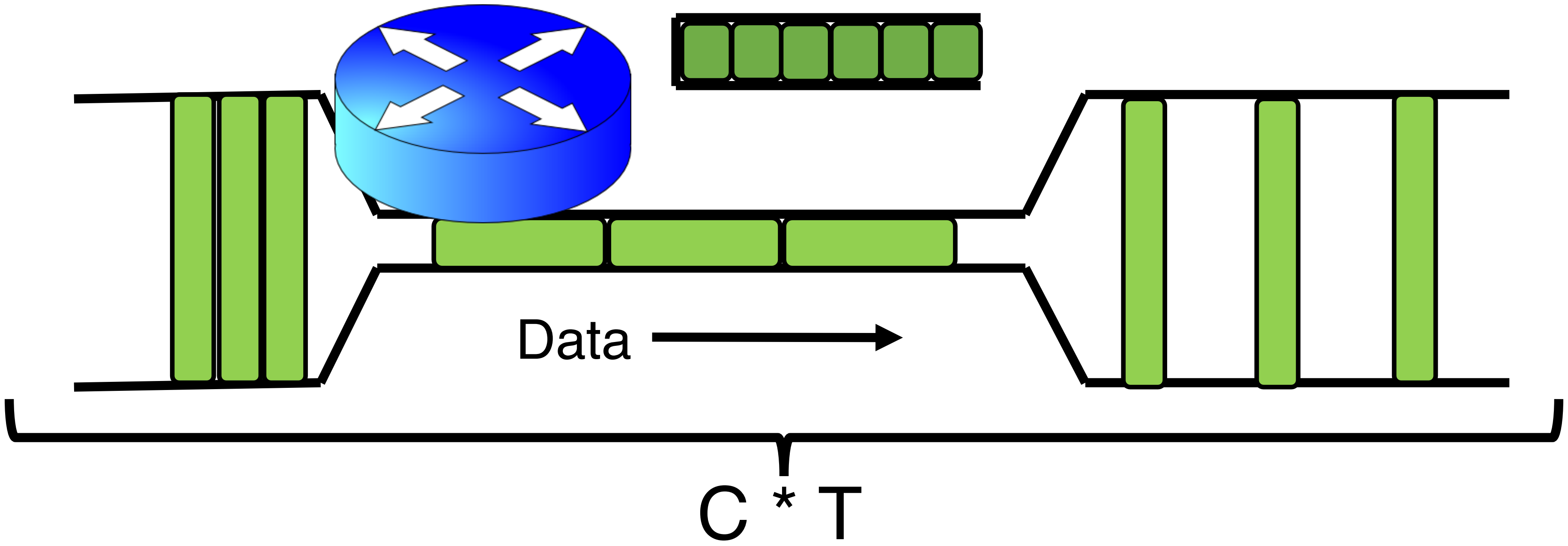
- Note: this is just the amount of data "on the pipe"



The Bandwidth-Delay Product

- Q: What happens if cwnd > C * T?
  - i.e., where are the rest of the in-flight packets?

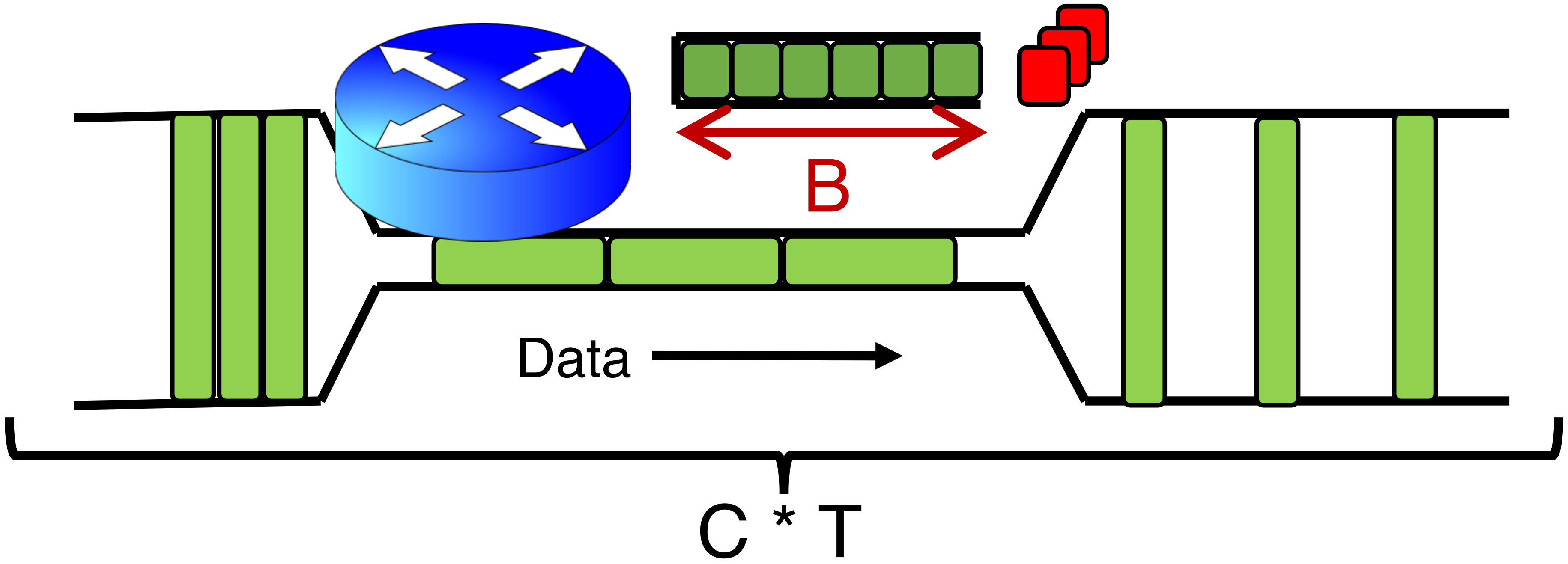
- A: Waiting at the bottleneck router queues



Router buffers and the max `cwnd`

- Router buffer memory is finite: queues can only be so long
  - If the router buffer size is B, there is at most B data waiting in the queue

- If cwnd increases beyond C * T + B, data is dropped!



Summary

- Bandwidth-Delay Product (BDP) governs the window size of a single flow at steady state
- The bottleneck router buffer size governs how much the `cwnd` can exceed the BDP before packet drops occur

Detecting and Reacting to Packet Loss

Detecting packet loss

- So far, all the algorithms we've studied have a coarse loss detection mechanism: RTO timer expiration
  - Let the RTO expire, drop `cwnd` all the way to 1 MSS
- Analogy: you're driving a car
  - You're waiting until the next car in front is super close to you (RTO) and then hitting the brakes really hard (set cwnd := 1)
  - Q: Can you see obstacles from afar and slow down proportionately?
- That is, can the sender see packet loss coming in advance?
  - And reduce `cwnd` more gently?

Can we detect loss earlier than RTO?

- Key idea: use the information in the ACKs. How?

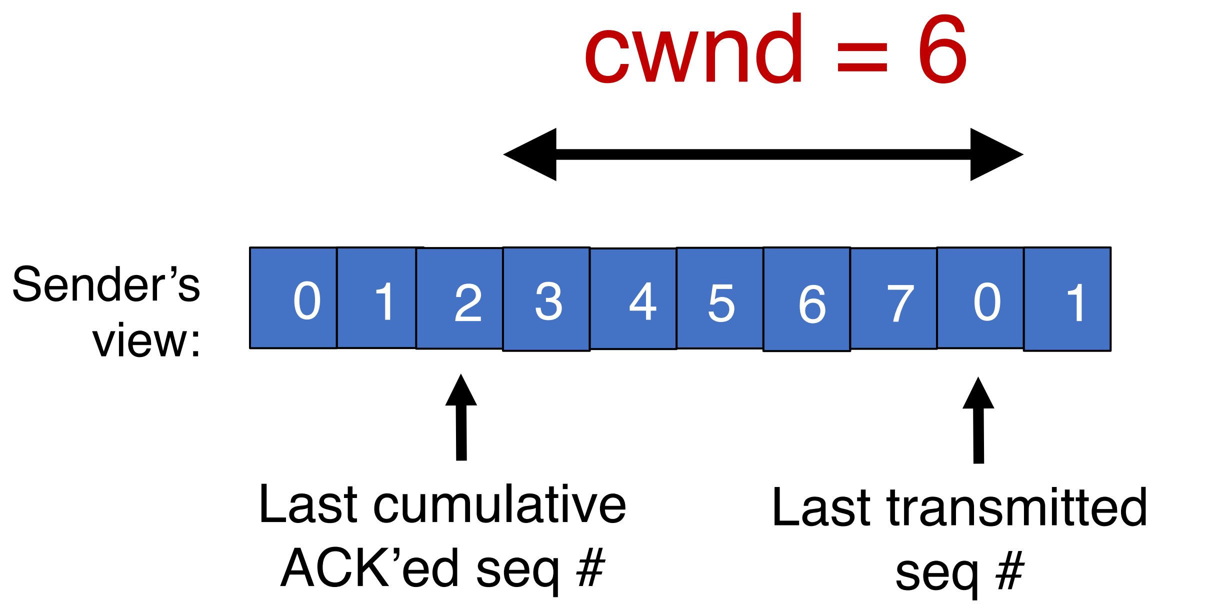
- Suppose successive (cumulative) ACKs contain the same ACK#
  - Also called duplicate ACKs
  - Occur when network is reordering packets, or one (but not most) packets in the window were lost

- Reduce `cwnd` when you see many duplicate ACKs
  - Consider many dup ACKs a strong indication that packet was lost
  - Default threshold: 3 dup ACKs, i.e., triple duplicate ACK
  - Make cwnd reduction gentler than setting cwnd = 1; recover faster

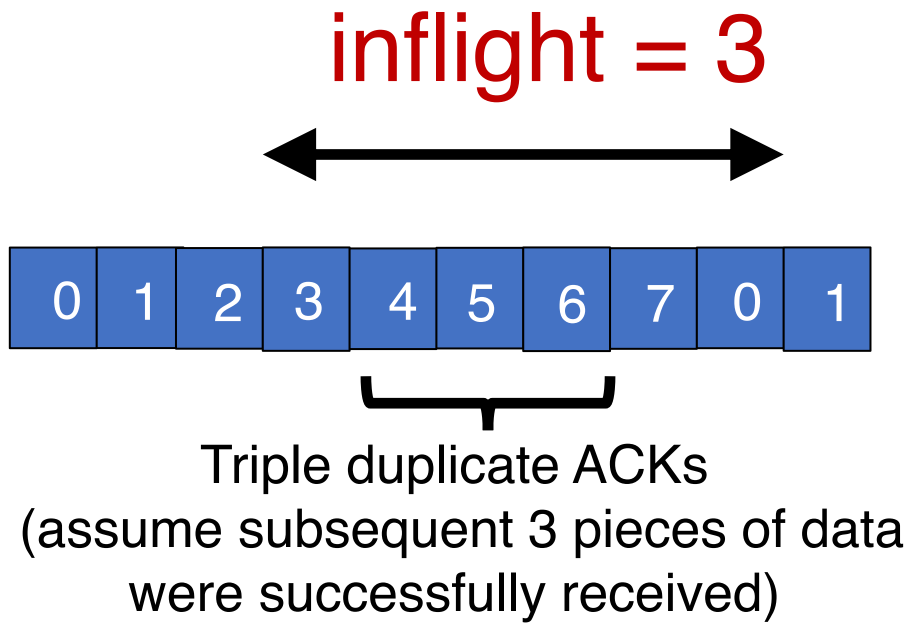
Fast Retransmit & Fast Recovery

Distinction: In-flight versus window

- So far, window and in-flight referred to the same data
- Fast retransmit & fast recovery differentiate the two notions



`cwnd` is the interval between the last cumulatively ACK'ed seq# and the last transmitted seq#


inflight = 3
0
1
2
3
4
5
6
7
0
1
Triple duplicate ACKs (assume subsequent 3 pieces of data were successfully received)

`inflight` is the data currently believed to be in flight.

TCP fast retransmit (RFC 2581)

- The fact that ACKs are coming means that data is getting delivered to the receiver, albeit with some loss.
- Note: Before the dup ACKs arrive, we assume `inflight = cwnd`
- TCP sender does two actions with fast retransmit

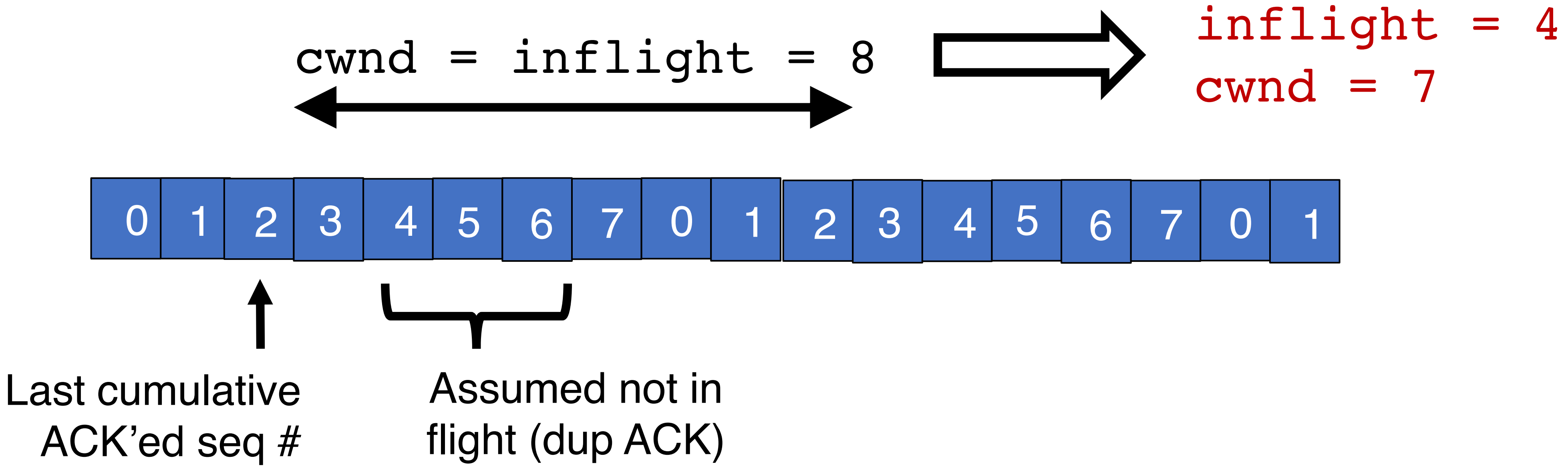
TCP fast retransmit (RFC 2581)

- (1) Reduce the `cwnd` and `in-flight` gently
    - Don't drop `cwnd` all the way down to 1 MSS

- Reduce the amount of in-flight data multiplicatively
    - Set `inflight → inflight / 2`
    - That is, set `cwnd = (inflight / 2) + 3MSS`
    - This step is called multiplicative decrease
    - Algorithm also sets `ssthresh` to `inflight / 2`

TCP fast retransmit (RFC 2581)

- Example: Suppose `cwnd` and `inflight` (before triple dup ACK) were both 8 MSS.
- After triple dup ACK, reduce `inflight` to 4 MSS
- *Assume* 3 of those 8 MSS no longer in flight; set `cwnd` = 7 MSS



TCP fast retransmit (RFC 2581)

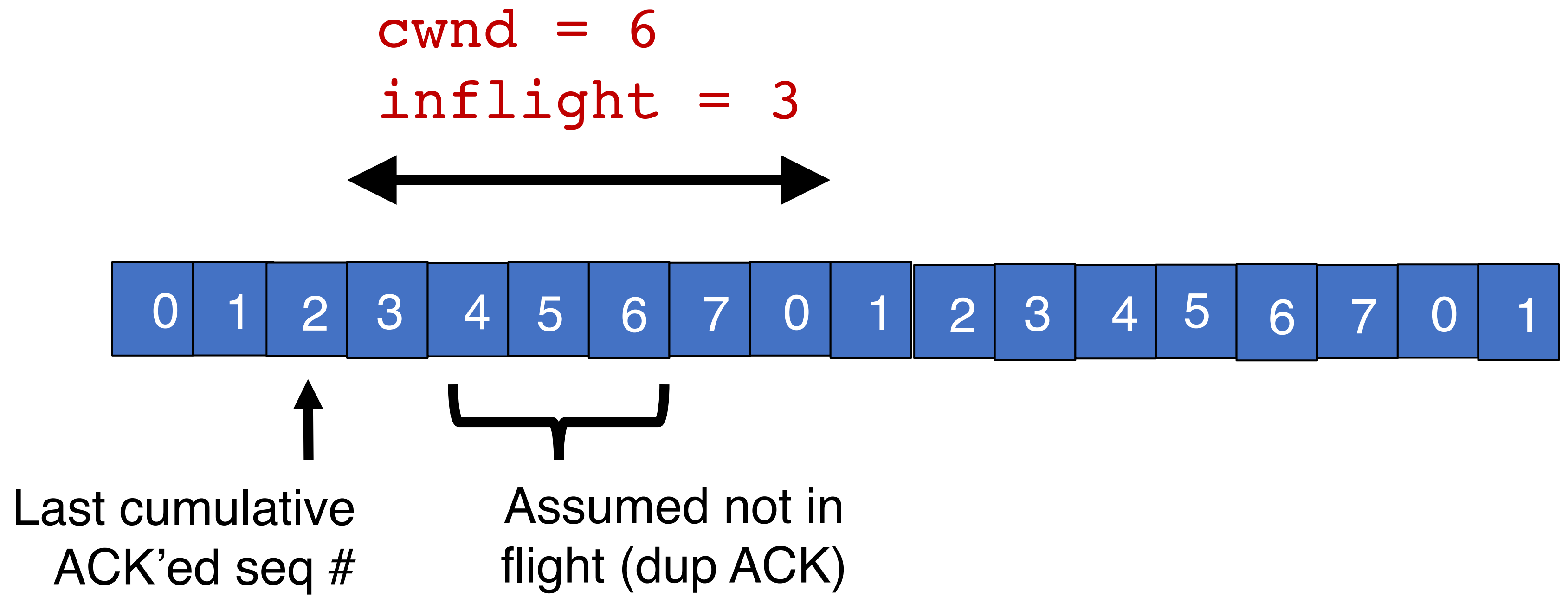
- (2) The seq# from dup ACKs is immediately retransmitted
- That is, don’t wait for an RTO if there is sufficiently strong evidence that a packet was lost

TCP fast recovery (RFC 2581)

- Sender keeps the reduced `inflight` until a new ACK arrives
    - New ACK: an ACK for the seq# that was just retransmitted
    - May also include the (three or more) pieces of data that were subsequently delivered to generate the duplicate ACKs
- Conserve packets in flight: transmit *some* data over lossy periods (rather than no data, which would happen if `cwnd` := 1)

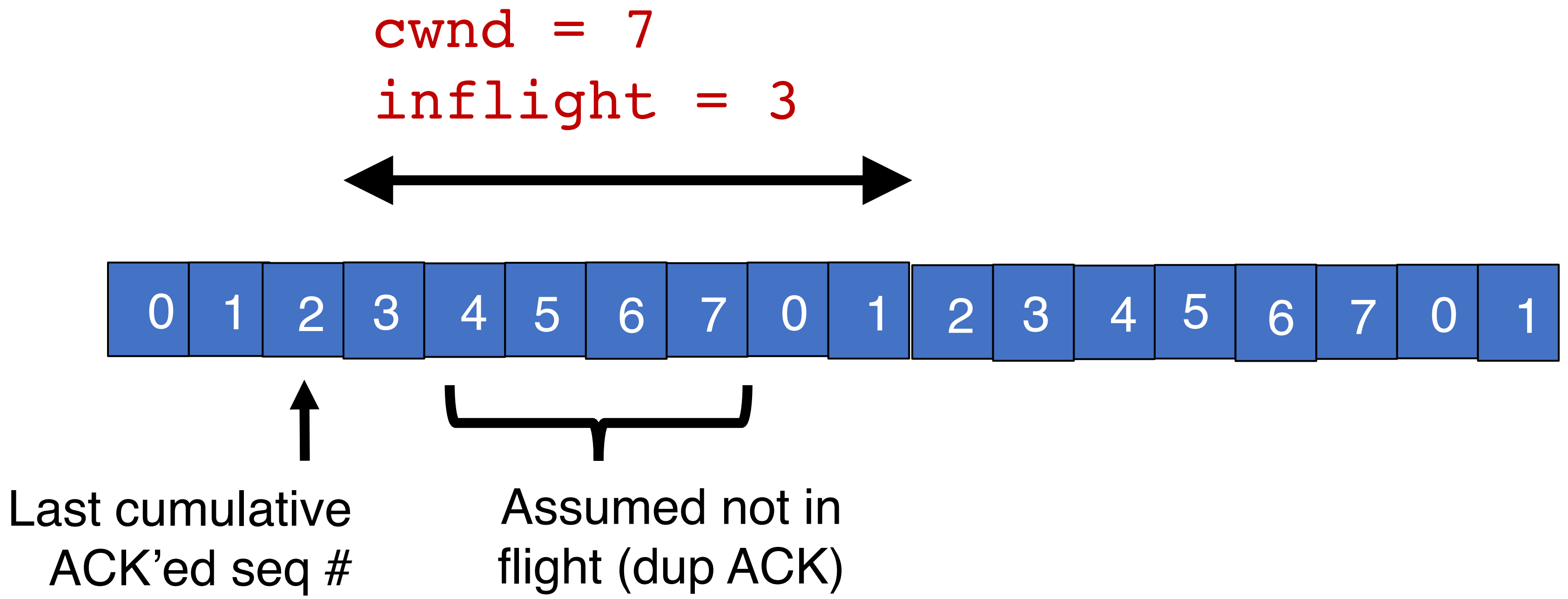
TCP fast recovery (RFC 2581)

- Keep incrementing `cwnd` by 1 MSS for each dup ACK



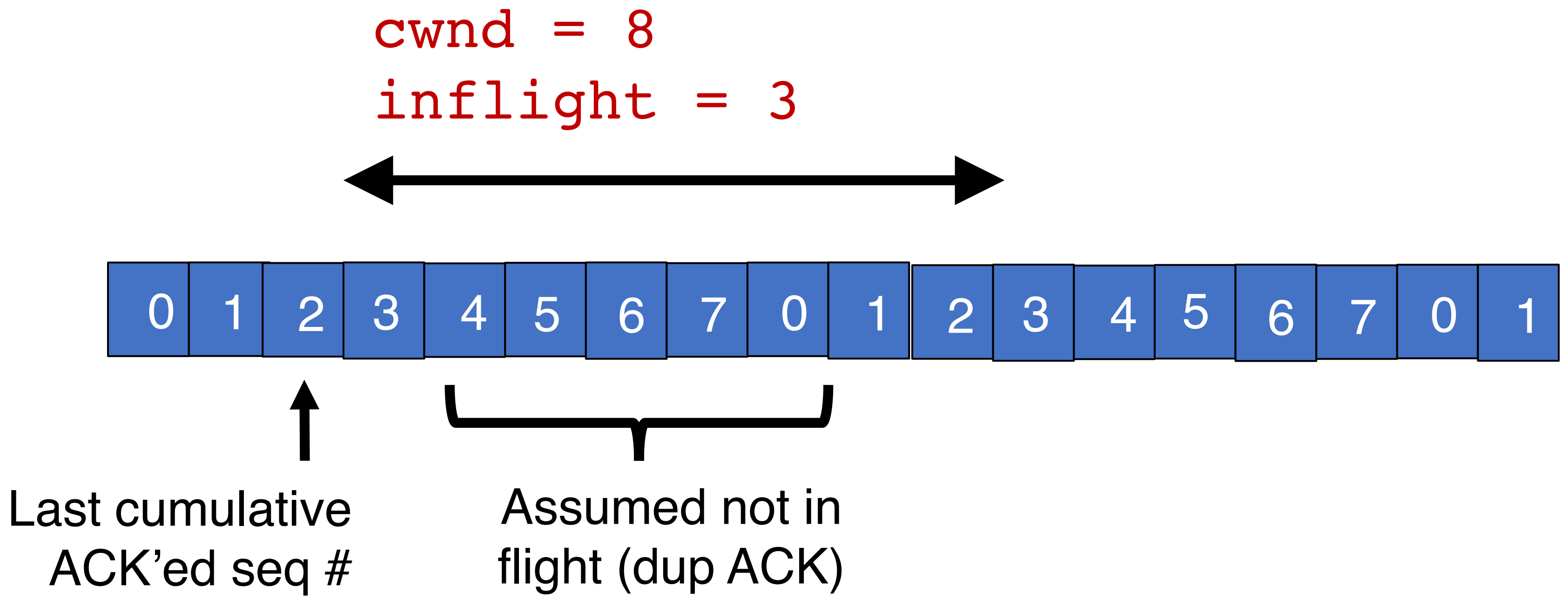
TCP fast recovery (RFC 2581)

- Keep incrementing `cwnd` by 1 MSS for each dup ACK



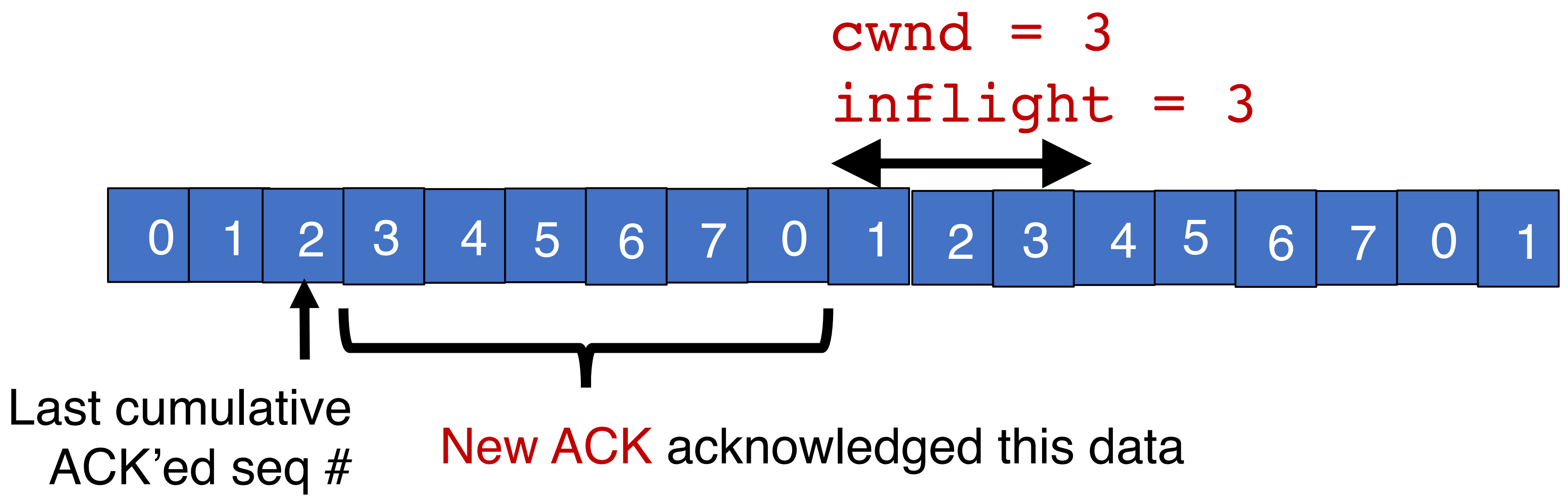
TCP fast recovery (RFC 2581)

- Keep incrementing `cwnd` by 1 MSS for each dup ACK



TCP fast recovery (RFC 2581)

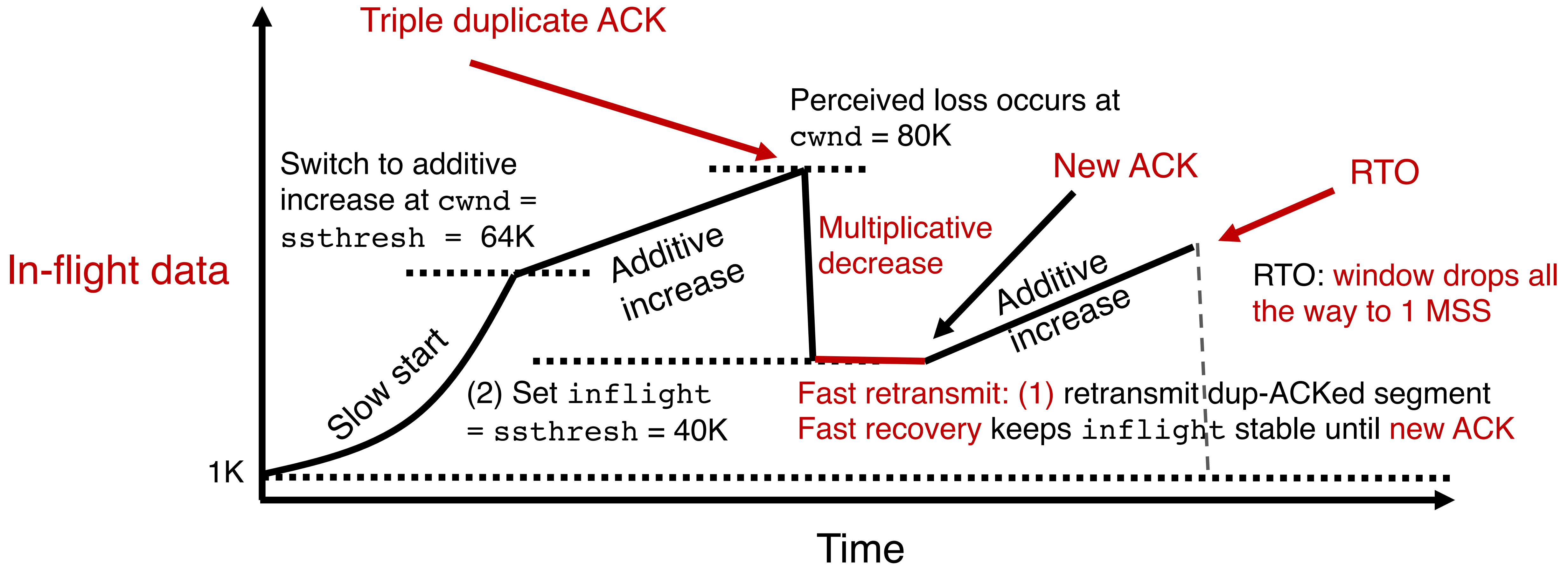
- Eventually a new ACK arrives, acknowledging the retransmitted data and all data in between
- Deflate `cwnd` to half of `cwnd` before fast retransmit.
  - `cwnd` and `inflight` are aligned and equal once again
- Perform additive increase from this point!



Additive Increase/Multiplicative Decrease

Say `MSS` = 1 KByte

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


Triple duplicate ACK
Perceived loss occurs at cwnd = 80K
New ACK
RTO
Switch to additive increase at cwnd = ssthresh = 64K
Additive increase
Multiplicative decrease
Additive increase
RTO: window drops all the way to 1 MSS
In-flight data
Slow start
(2) Set inflight = ssthresh = 40K
Fast retransmit: (1) retransmit dup-ACKed segment
Fast recovery keeps inflight stable until new ACK
1K
Time

TCP New Reno performs additive increase and multiplicative decrease of its congestion window.

In short, we often refer to this as AIMD.

Multiplicative decrease is a part of all TCP algorithms, including BBR.
[It is necessary for fairness across TCP flows.]

Summary: TCP loss detection & reaction

- Don't wait for an RTO and then set the `cwnd` to 1 MSS
  - Tantamount to waiting to get super close to the car in front and then jamming the brakes really hard
- Instead, react proportionately by sensing pkt loss in advance

Fast Retransmit

- Triple dup ACK: sufficiently strong signal that network has dropped data, before RTO
- Immediately retransmit data
- Multiplicatively decrease in-flight data to half of its value

Fast Recovery

- Maintain this reduced amount of in-flight data as long as dup ACKs arrive
  - Data is successfully getting delivered
- When new ACK arrives, do additive increase from there on