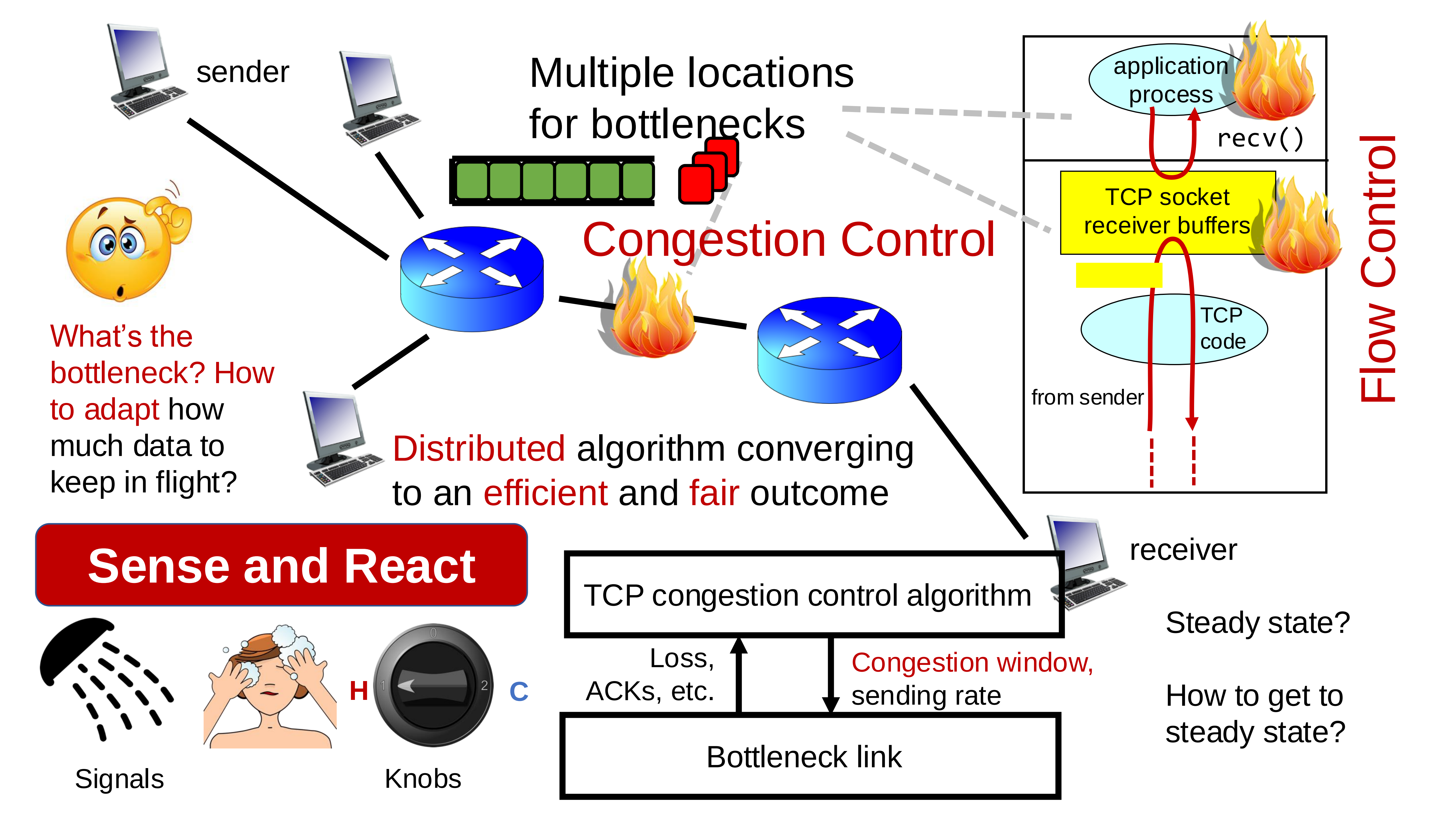
Congestion Control II

Lecture 18

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

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK



sender
Multiple locations for bottlenecks
application process
recv()
TCP socket receiver buffers
Congestion Control
TCP code
from sender
Flow Control
What's the bottleneck? How to adapt how much data to keep in flight?
Distributed algorithm converging to an efficient and fair outcome
receiver
Sense and React
TCP congestion control algorithm
Steady state?
Loss, ACKs, etc.
Congestion window, sending rate
How to get to steady state?
H
C
0
1
2
Bottleneck link
Signals
Knobs

Congestion window

- The sender maintains an estimate of the amount of in-flight data needed to keep the link fully busy 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
  - Rate = window / RTT

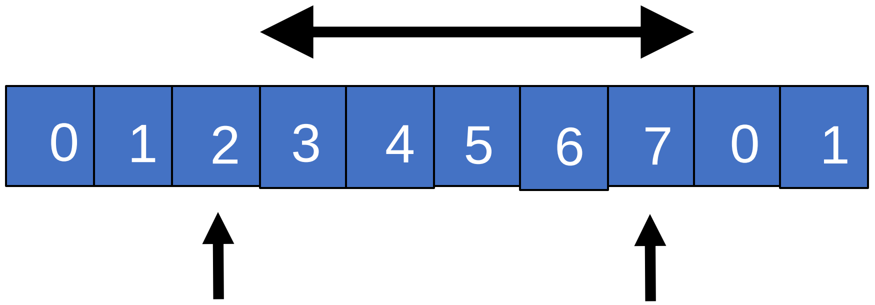
Interaction b/w flow & congestion control

- 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)

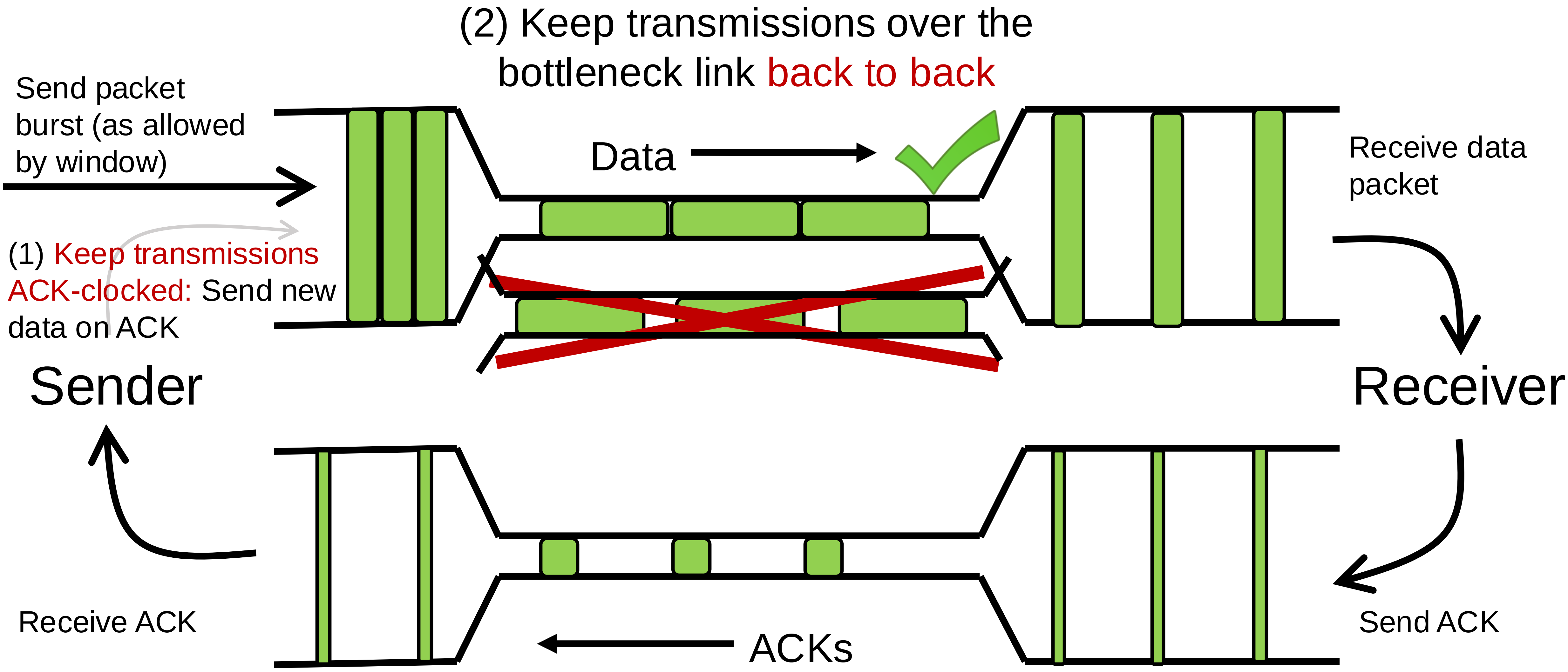
Sender's view:



Last cumulative ACK'ed seq #

Last transmitted seq #

Review: Steady state operation



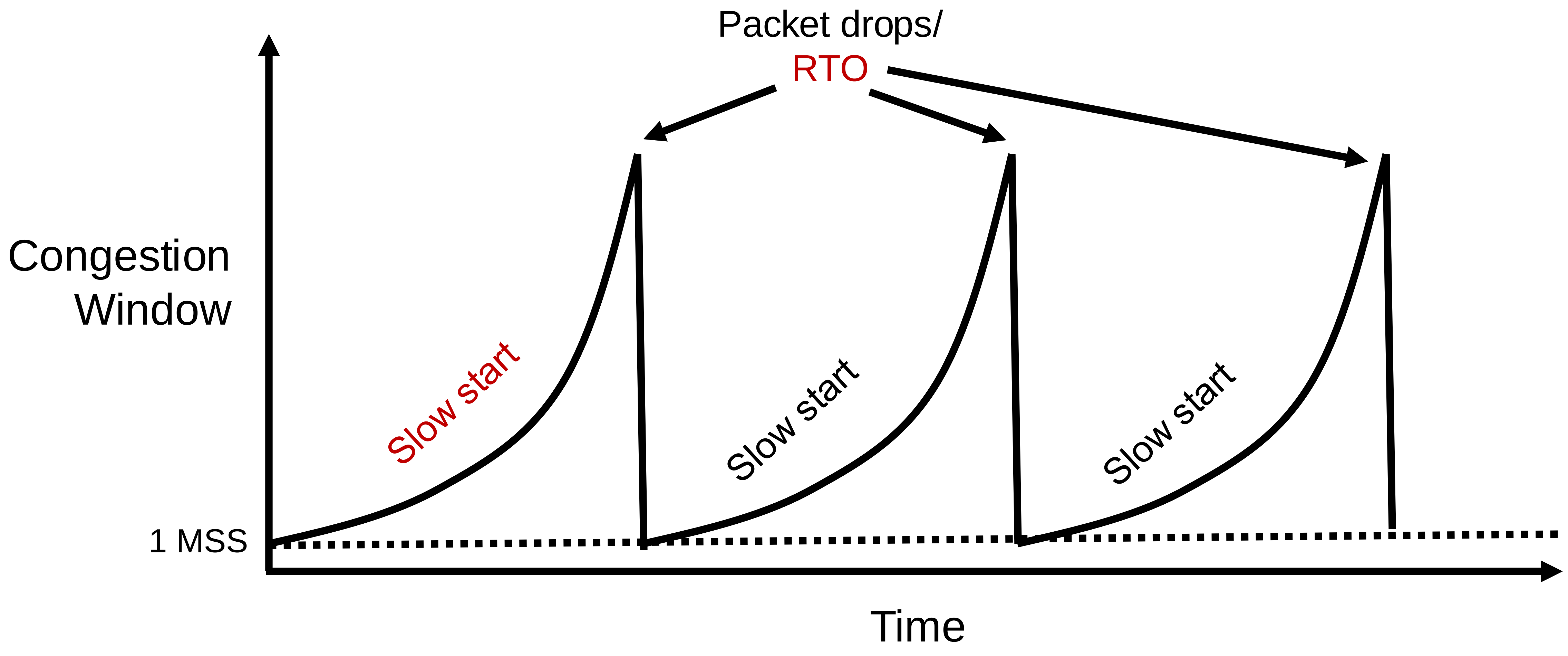
Review: Slow start

Q: How to get to steady state?

Problems:

- Congestion window grows too fast!
- Congestion window drops too fast!

Need gentler adaptation of cwnd closer to steady state



TCP Congestion Avoidance

Two congestion control algorithms

TCP New Reno

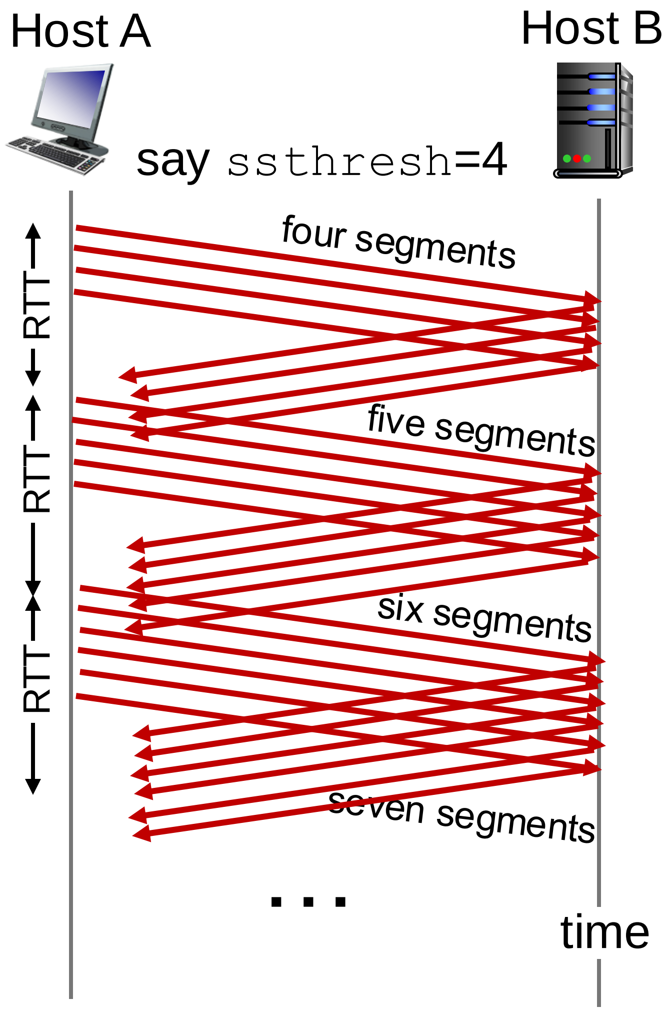
- The most studied, classic "textbook" TCP algorithm
- The primary knob is congestion window
- The primary signal is packet loss (RTO)
- Adjustment using additive increase

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

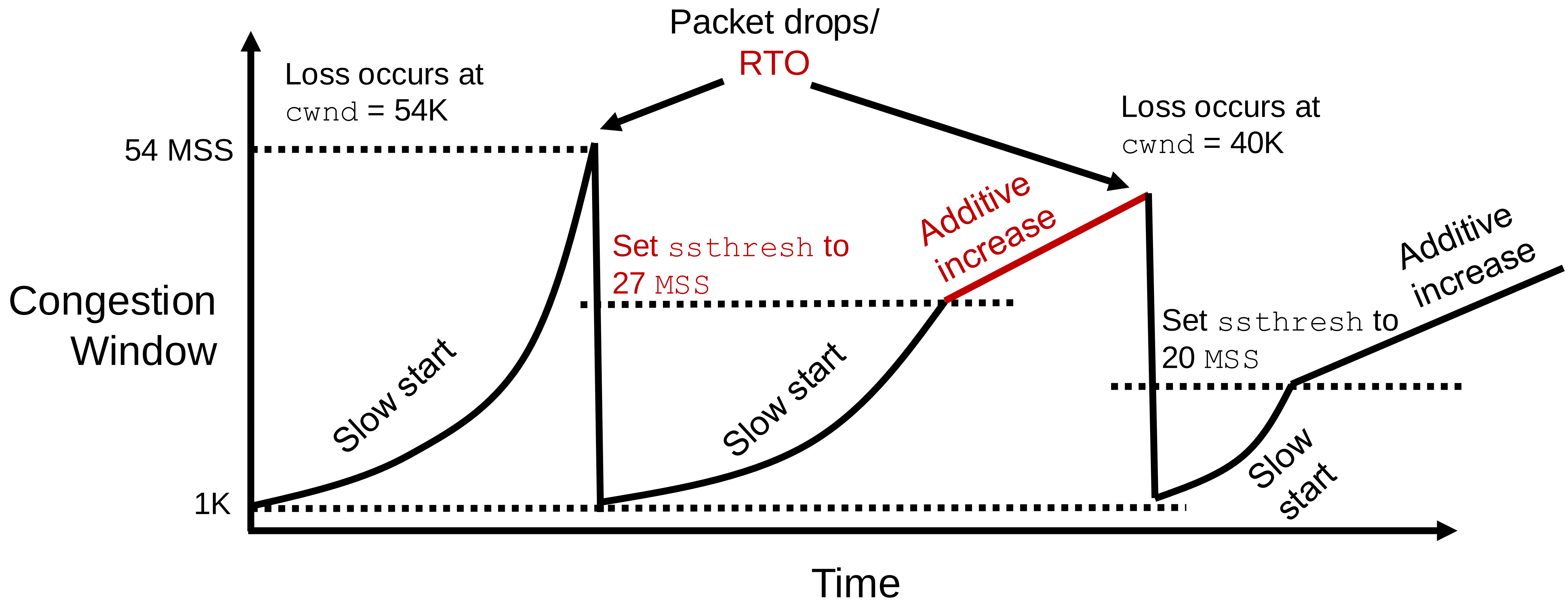


TCP New Reno: Additive increase

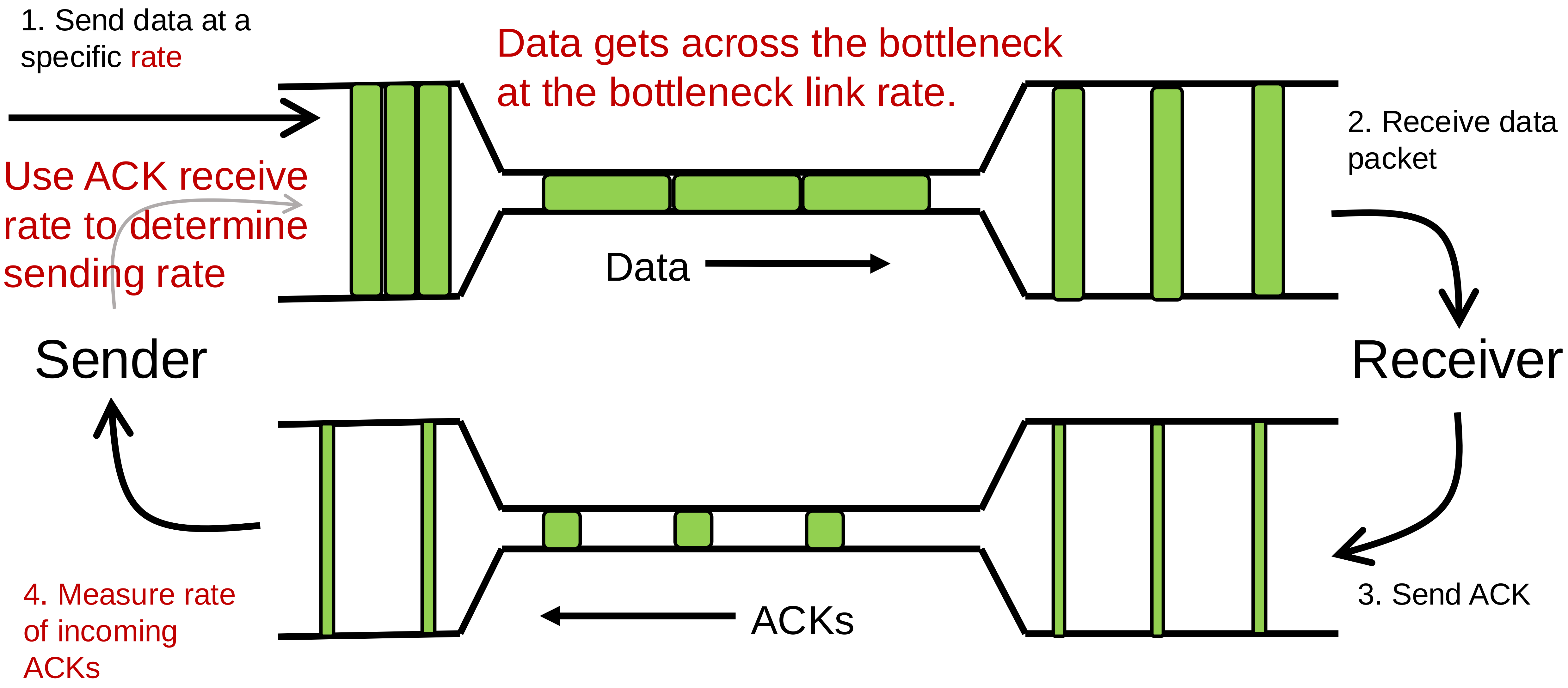
- Start with `ssthresh = 64K bytes` (TCP default)
- Do slow start until `ssthresh`
- Once the threshold is passed, do additive increase
  - Add one MSS to `cwnd` for each `cwnd` worth data ACK'ed
  - For each MSS ACK'ed, `cwnd = cwnd + (MSS/cwnd) * MSS`
- Upon a TCP timeout (RTO),
  - Set `cwnd = 1 MSS`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



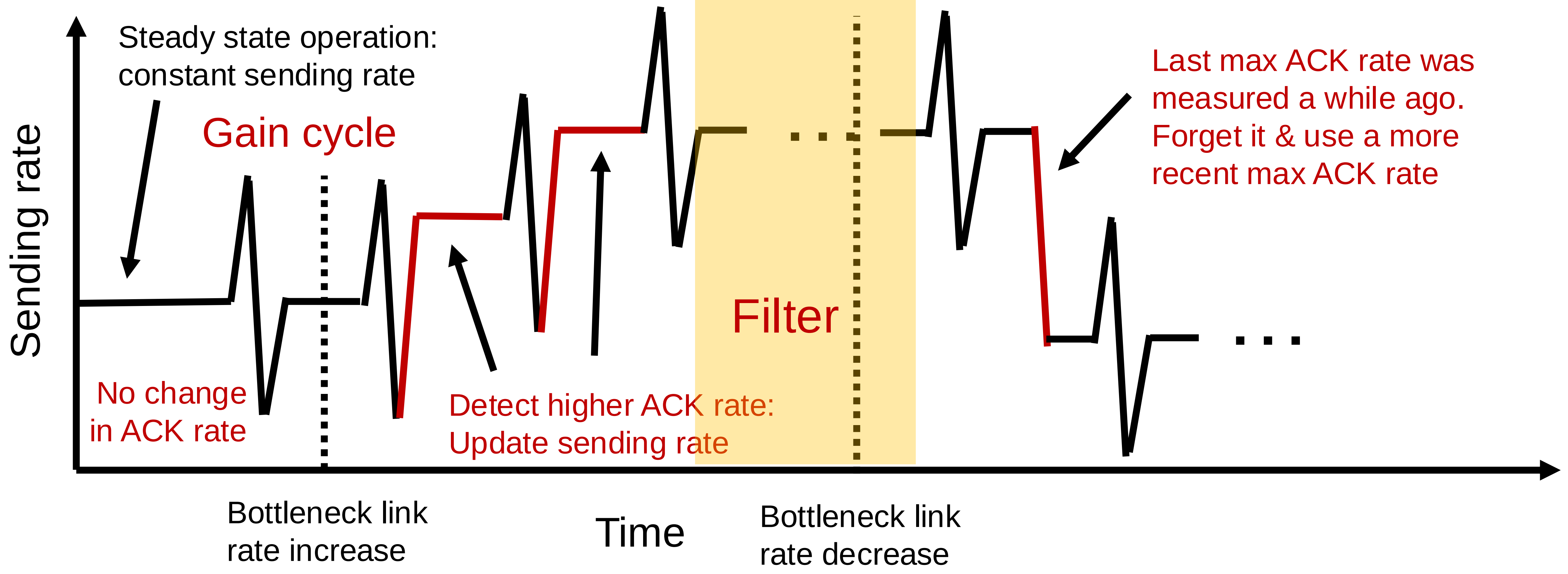
TCP BBR: finding the bottleneck link rate

- 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)



Bandwidth-Delay Product

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; back-to-back packets over bottleneck link

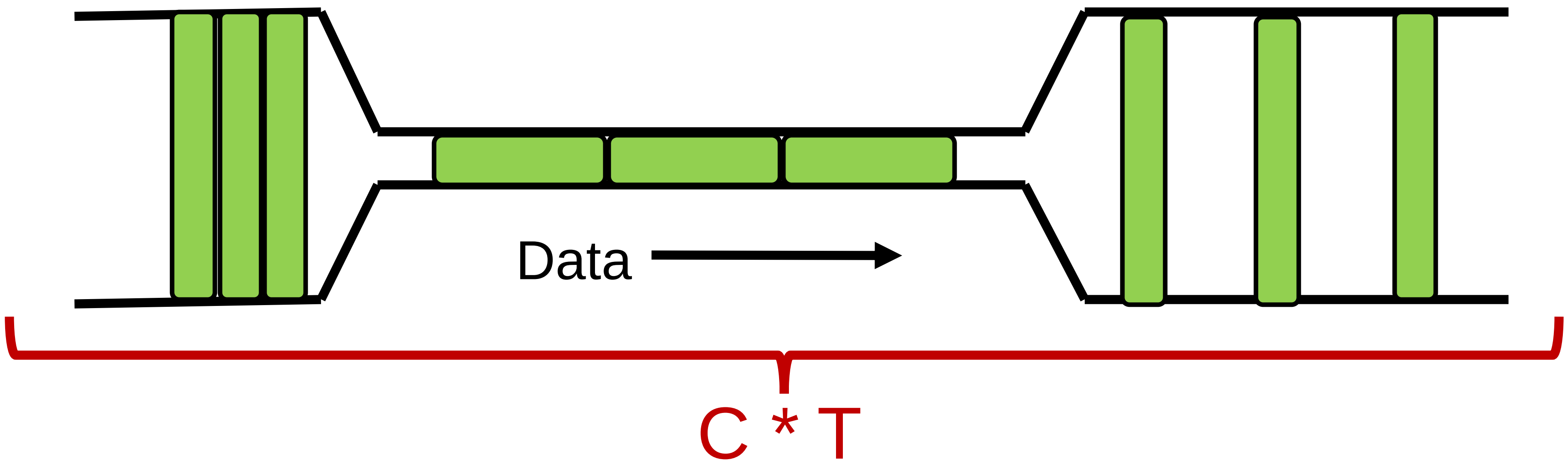
- Q: how much data is in flight over a single RTT?

- 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

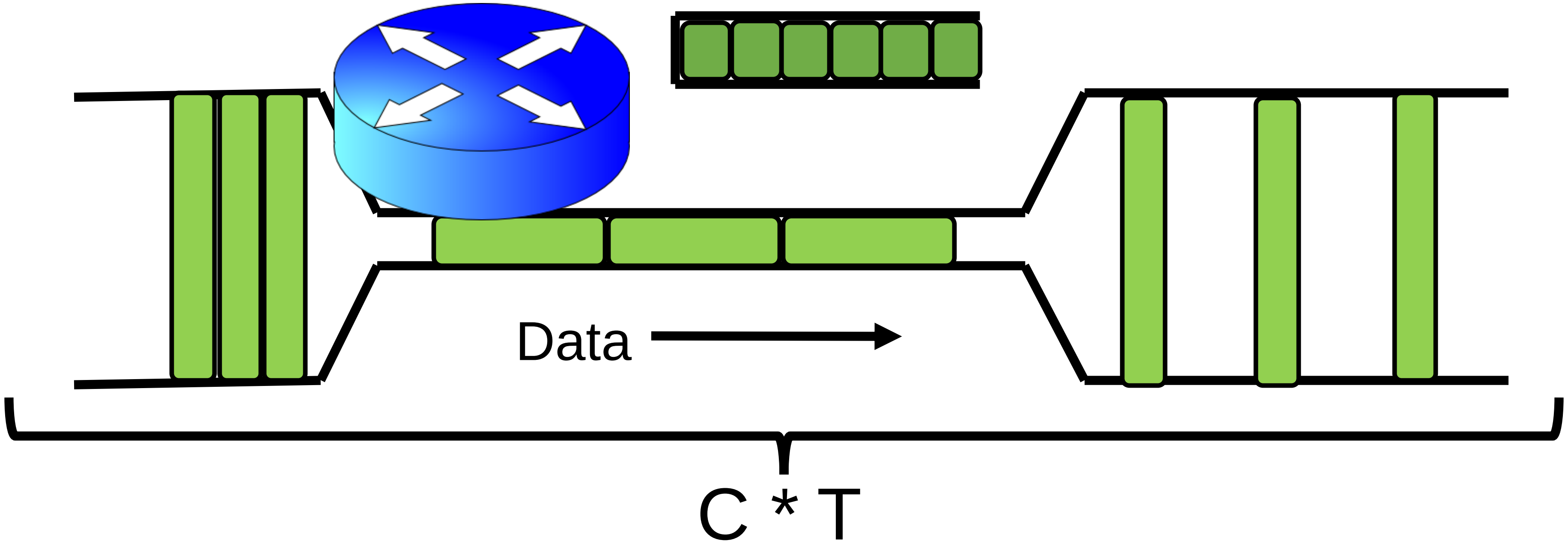
- Note: this is just the amount of data "on the pipes"



The Bandwidth-Delay Product

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

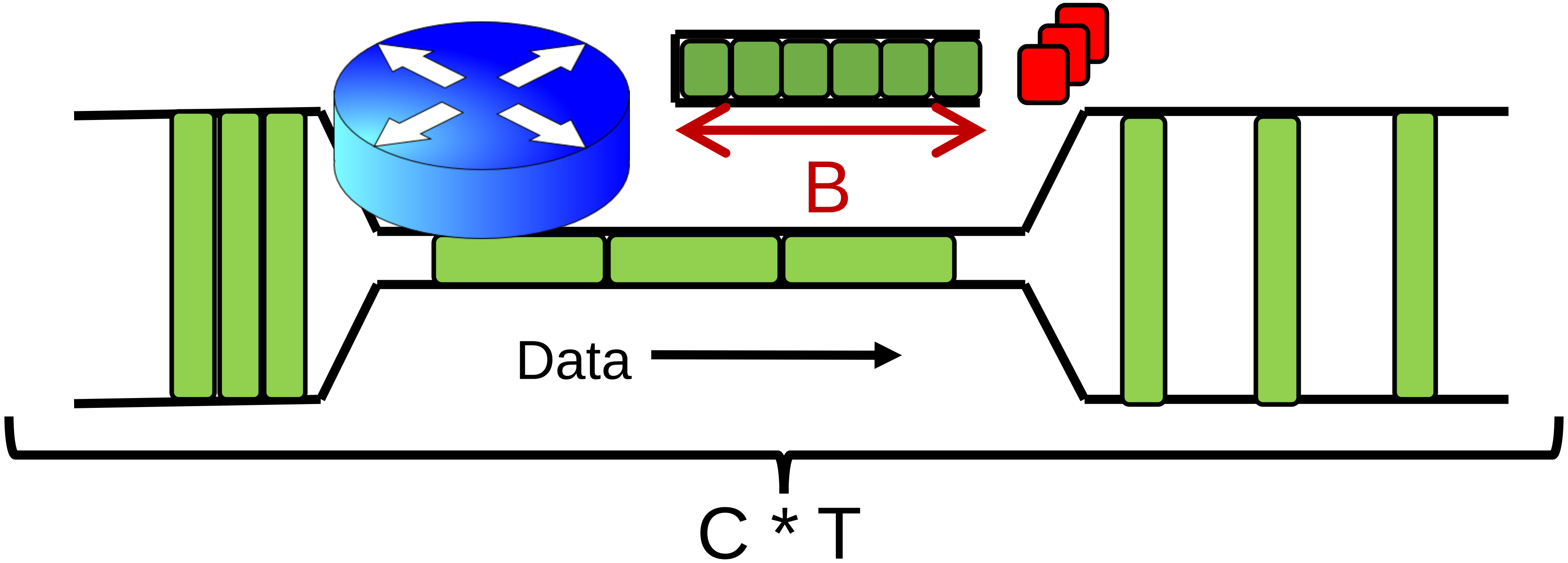
- A: Waiting at the bottleneck router queues



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!

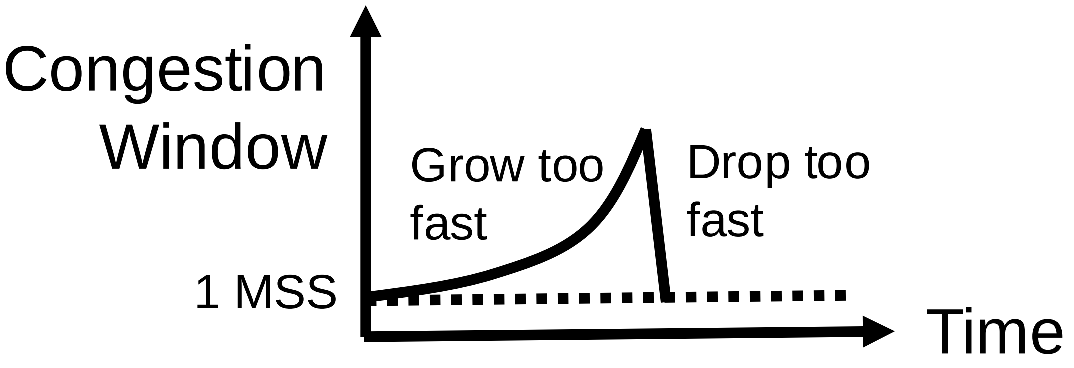


Summary

- 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
- BDP is the ideal desired window size to use the full bottleneck link, without any queueing.
  - Accommodating flow control, also the min socket buffer size to use the bottleneck link fully:
  - Important to set socket buffer sizes well for high BDP paths

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 accelerate until the next car in front is super close to you (RTO) and then hit the brakes hard (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

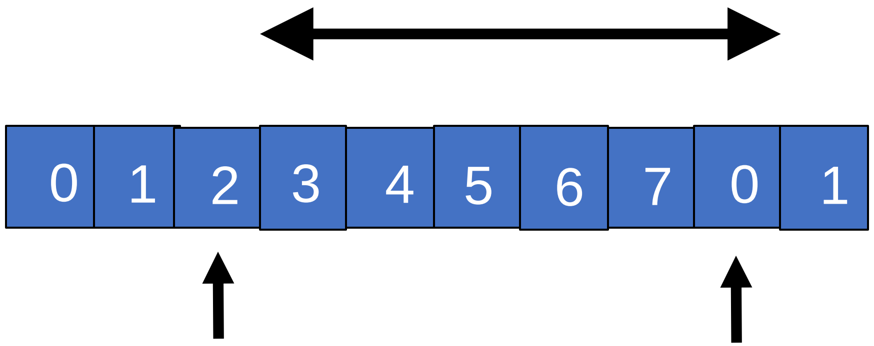
Fast Retransmit & Fast Recovery

Distinction: In-flight versus window

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

cwnd = 6

Sender's view:

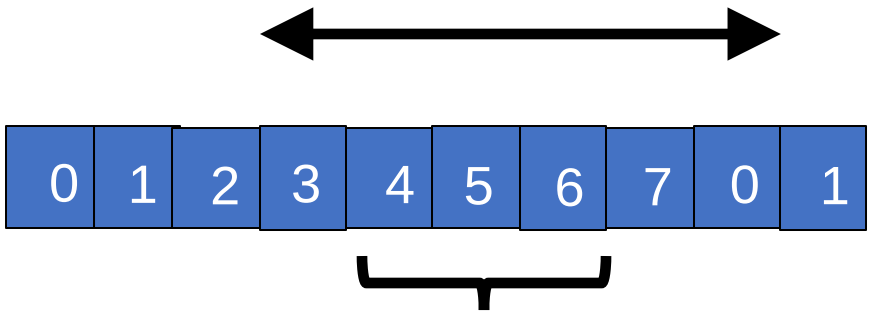


Last cumulative ACK'ed seq #

Last transmitted seq #

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

inflight = 3



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)

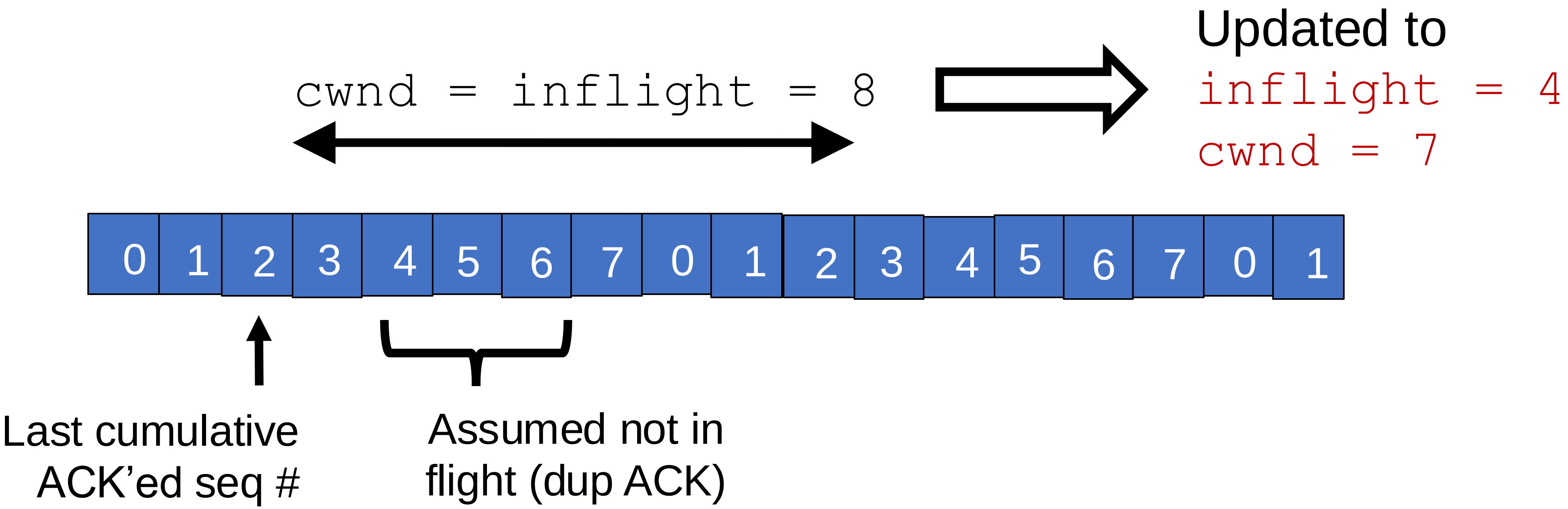
- The fact that ACKs are coming means that data is getting delivered to the receiver, although with some loss.
- Before the dup ACKs arrive, we assume `inflight = cwnd`
- TCP sender does two actions with fast retransmit

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

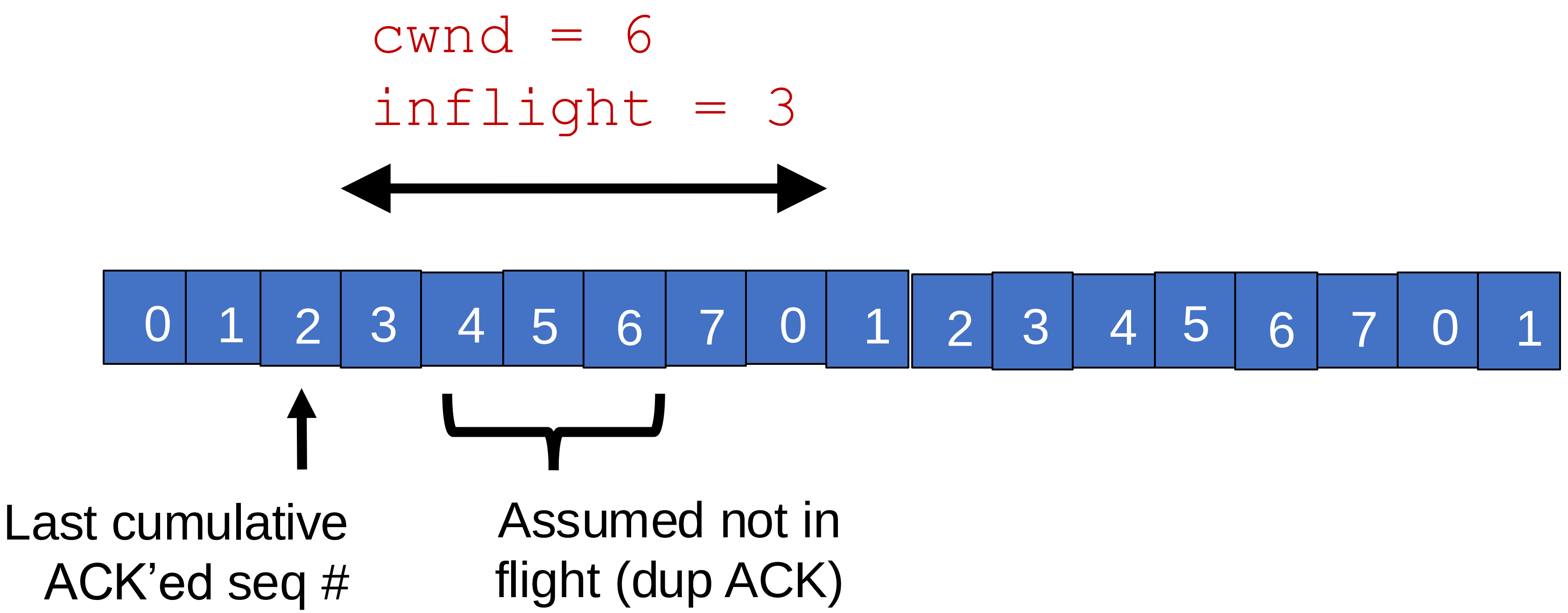
TCP fast recovery (RFC 2581)

- Sender keeps the reduced `inflight` until a new ACK arrives
  - New ACK: an ACK for the seq# that was just retransmitted
  - Cumulative ACK may also indicate the (three or more) pieces of data that were previously 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)

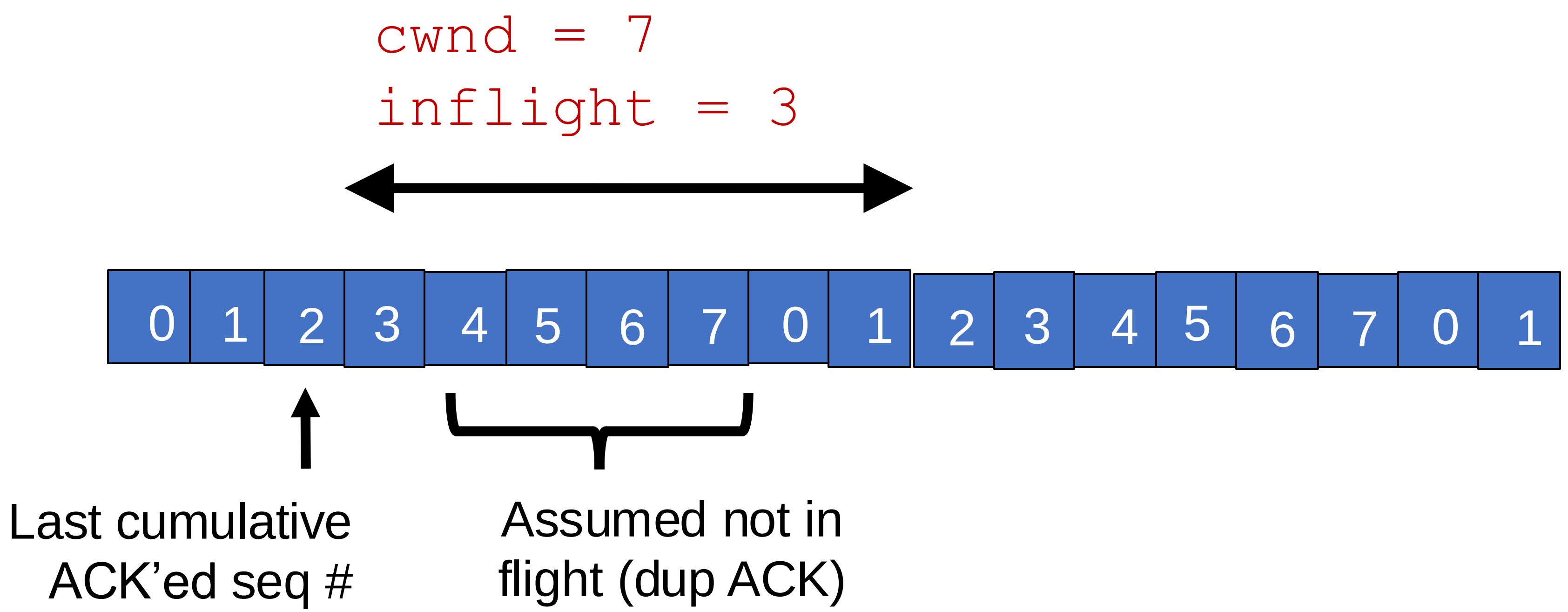
TCP fast recovery (RFC 2581)

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



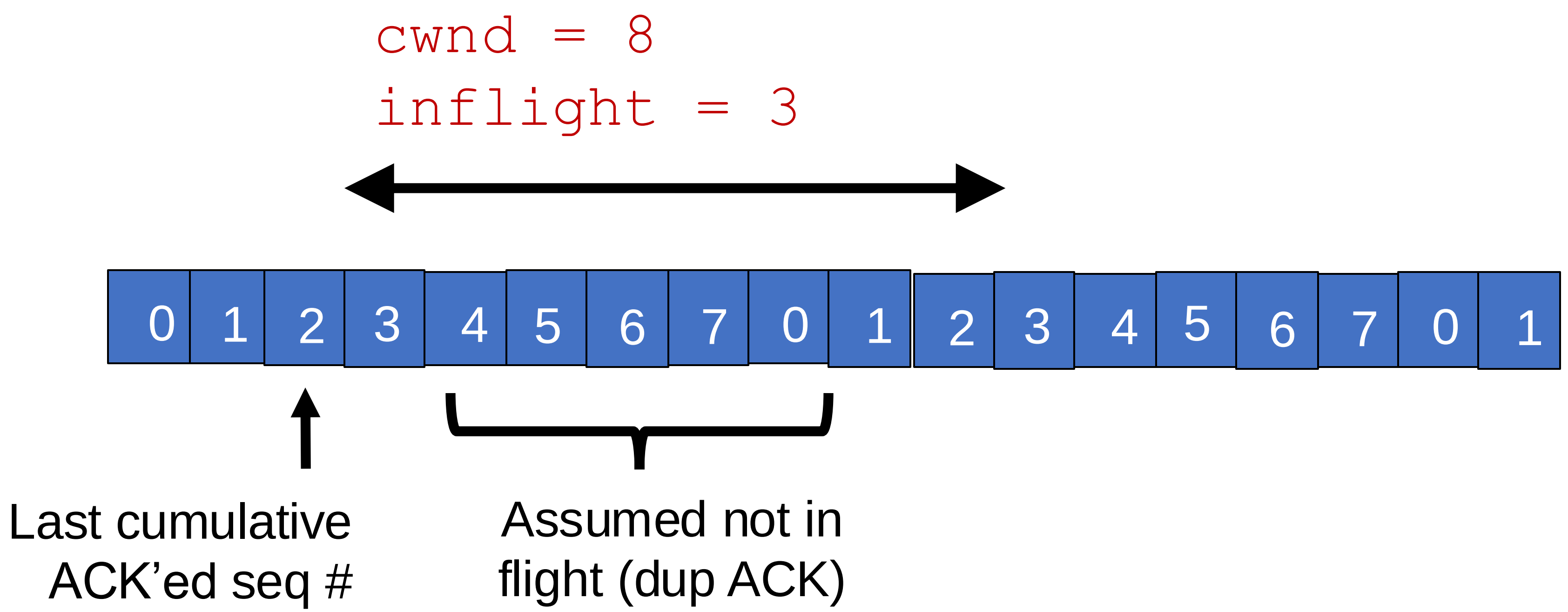
TCP fast recovery (RFC 2581)

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



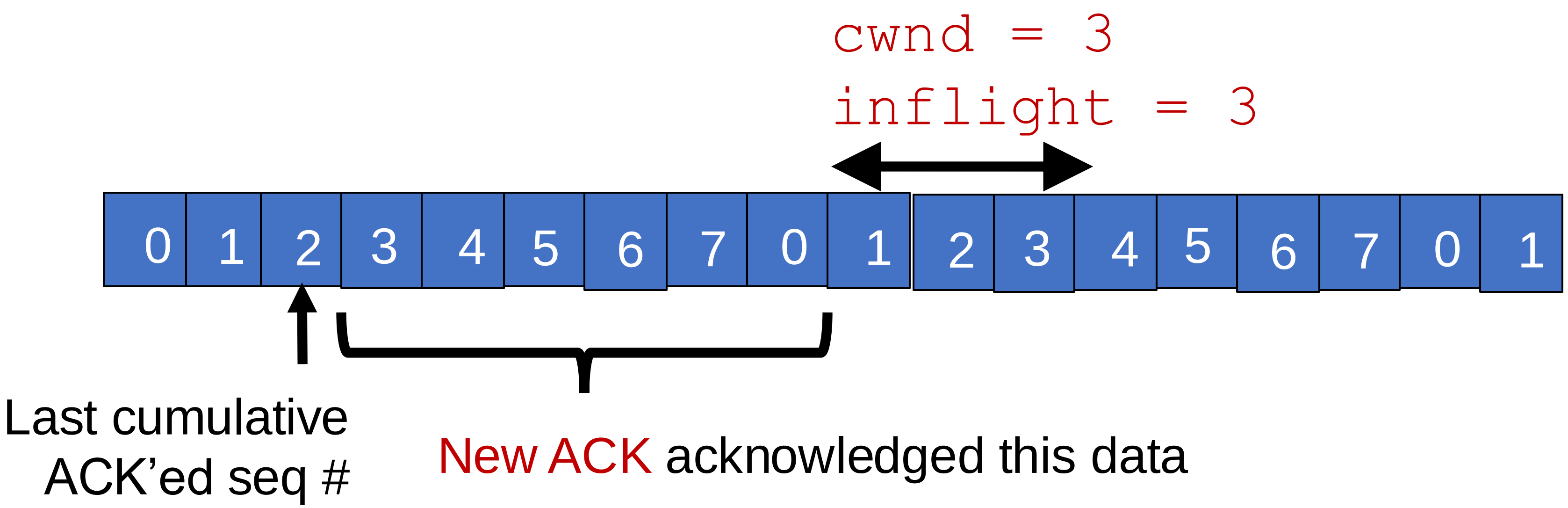
TCP fast recovery (RFC 2581)

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



TCP fast recovery (RFC 2581)

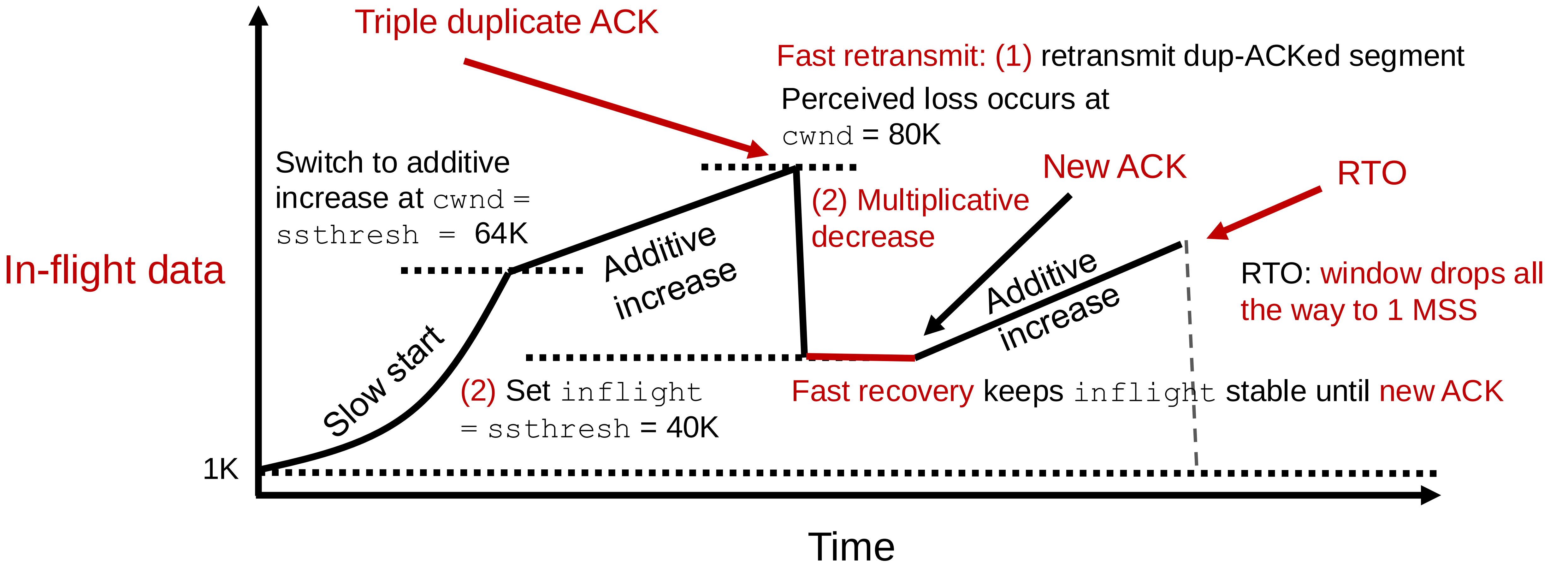
- Eventually a new ACK arrives, acknowledging the retransmitted data and all data in between
- Deflate `cwnd` to half of `cwnd` before fast retransmit.
  - `cwnd` and `inflight` are aligned and equal once again
- Perform additive increase from this point!



Additive Increase/Multiplicative Decrease

Say `MSS` = 1 KByte

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



TCP New Reno performs additive increase and multiplicative decrease of 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
- 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

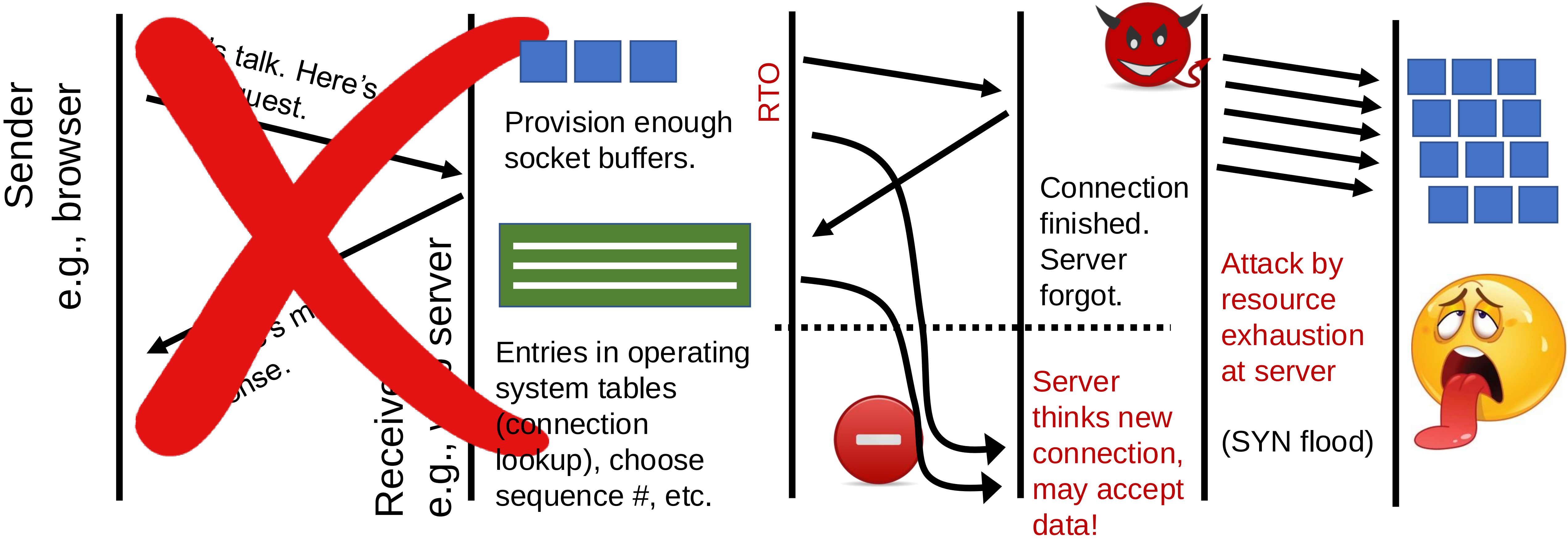
- Maintain this reduced amount of in-flight data as long as dup ACKs arrive
  - Data is successfully getting delivered
- When new ACK arrives, do additive increase from there on

Connection Management

How does a TCP connection start?

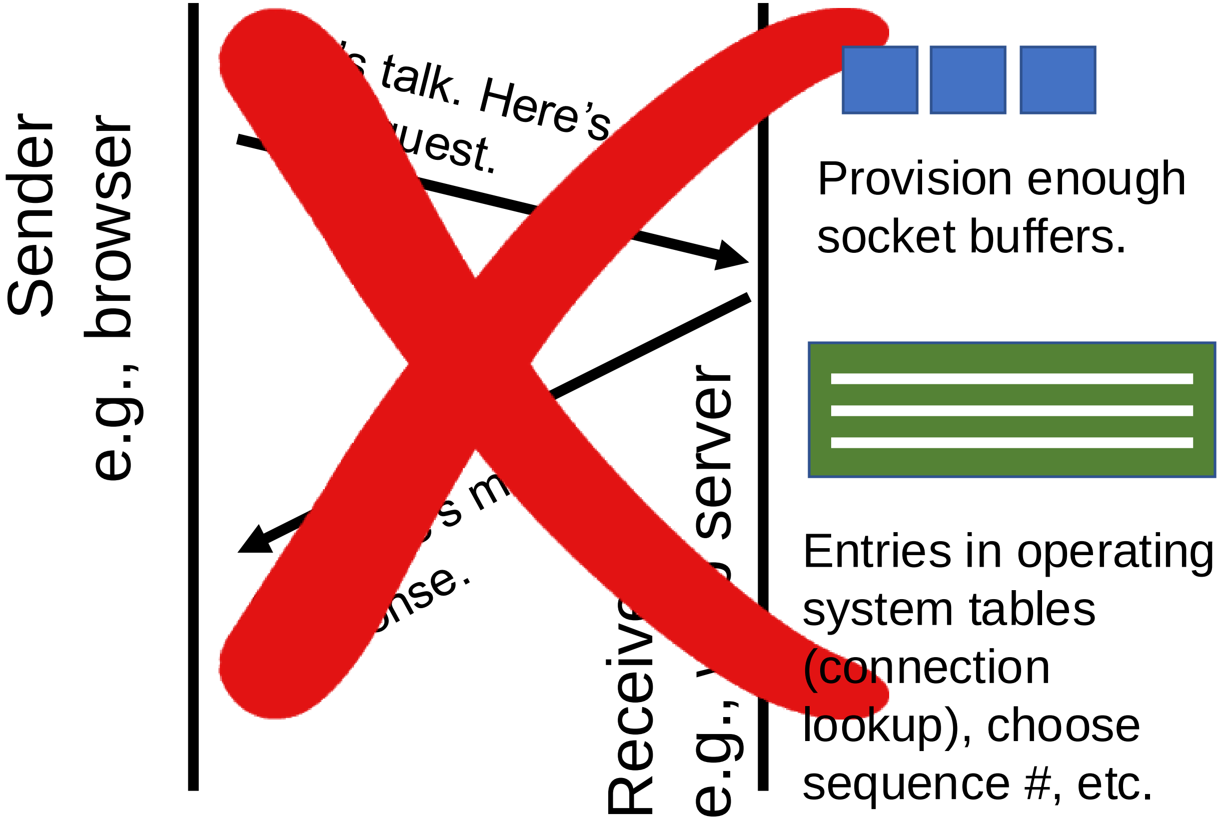
Starting a TCP connection

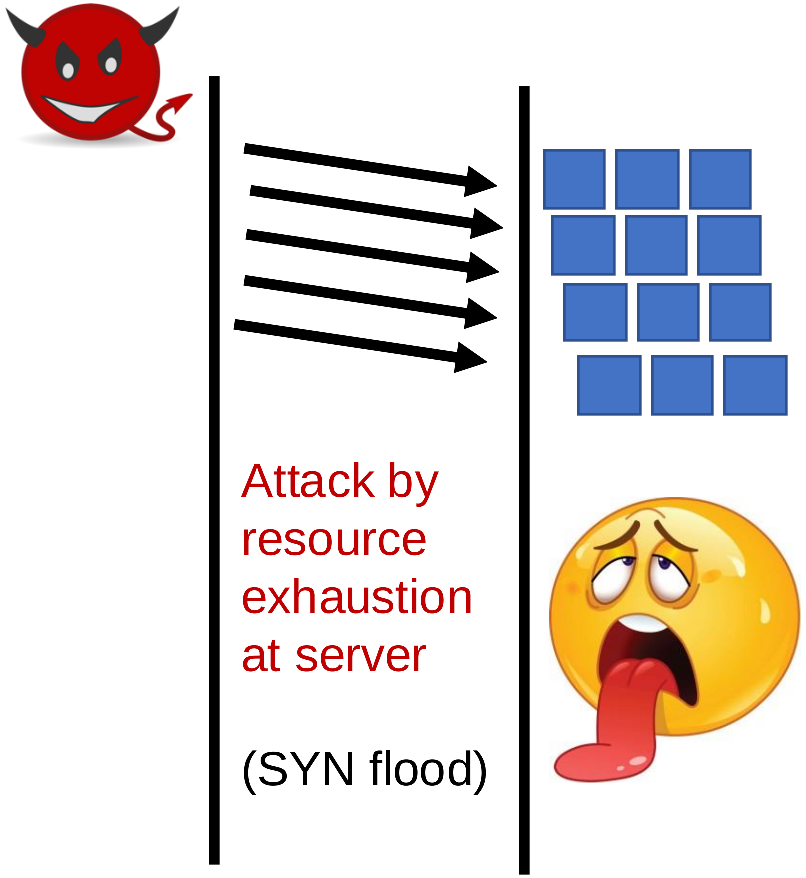
- TCP requires sender/receiver to set up some context
  - Sequence numbers, window size, buffers, OS table entries



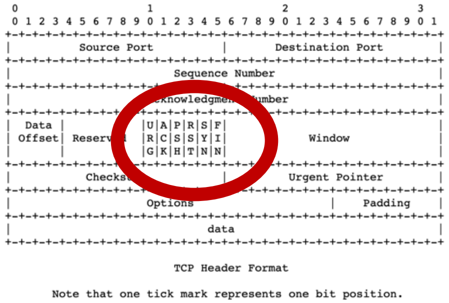
Starting a TCP connection

- TCP requires sender/receiver to set up some context
  - Sequence numbers, window size, buffers, OS table entries





TCP 3-way handshake



Client state

```
cs = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
cs.connect((host,server_port))
```

choose init seq num, x
send TCP SYN msg

SYNSENT

No app data

SYNbit=1, Seq=x

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

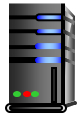
ESTAB

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

ACKbit=1, ACKnum=y+1

Server state

```
ss = socket(AF_INET,SOCK_STREAM)
ss.bind(('',server_port))
ss.listen(1)
csockid, addr = ss.accept()
```



LISTEN

choose init seq num, y
send TCP SYNACK
msg, acking SYN

SYN RCVD

received ACK(y)
indicates client is live

Provision resources

ESTAB

Implications of 3-way handshake

- Any application data can only be sent an RTT after

- Fresh connection: at least 2 RTTs to get a response
  - Often fruitful to use persistent connections

- Measures to address this startup delay:
  - TCP fast open
  - QUIC