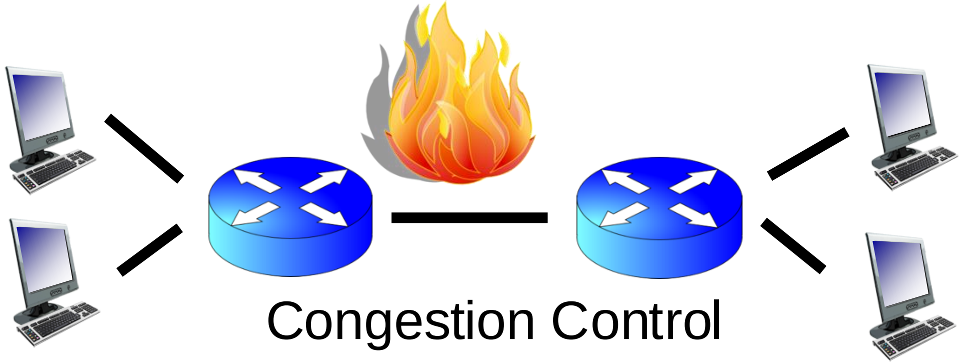
# Congestion Control III

Lecture 19

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

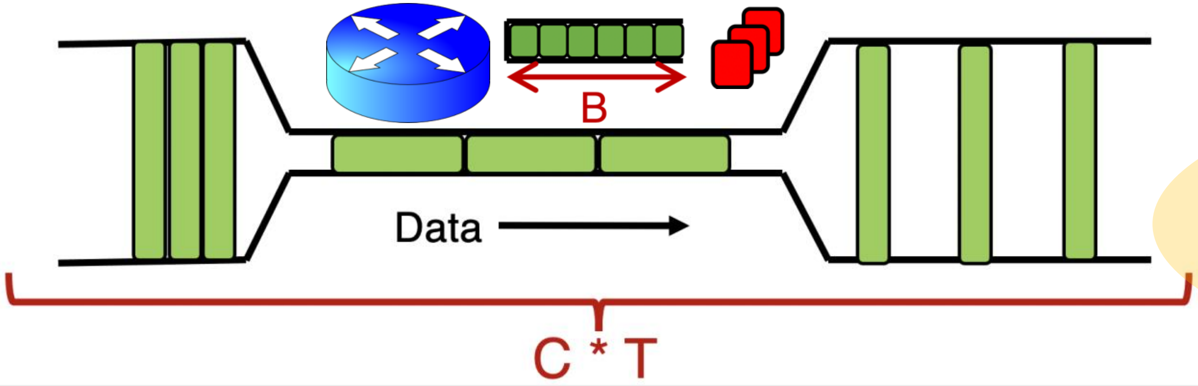
Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK




# Sense and React


## Bandwidth-Delay Product

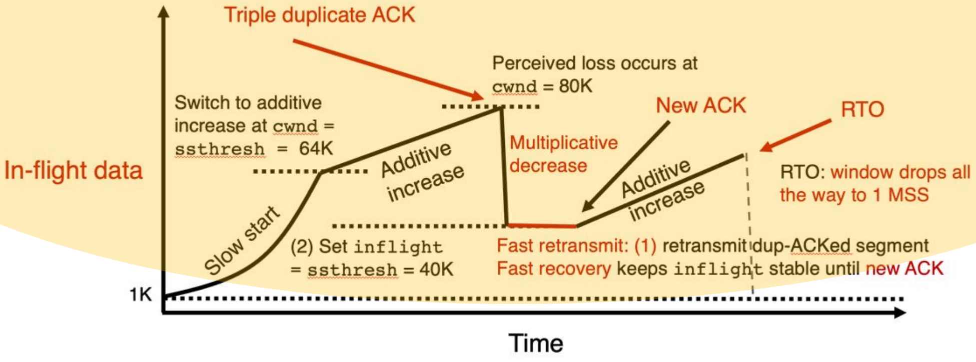



## TCP New Reno

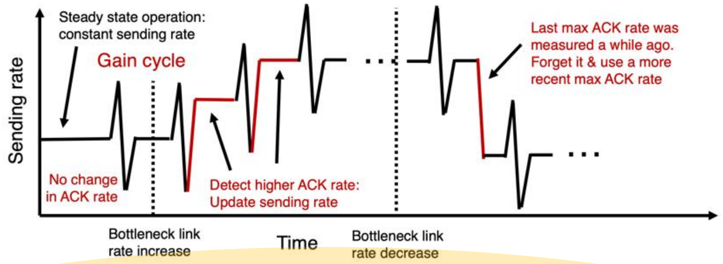
= slow start

+ congestion avoidance (AI)

+ fast retransmit & recovery (MD)




## TCP BBR: Gain cycling




`cwnd` < BDP: sender under-uses the link

BDP = `cwnd`: 100% link use, zero queues (ideal)

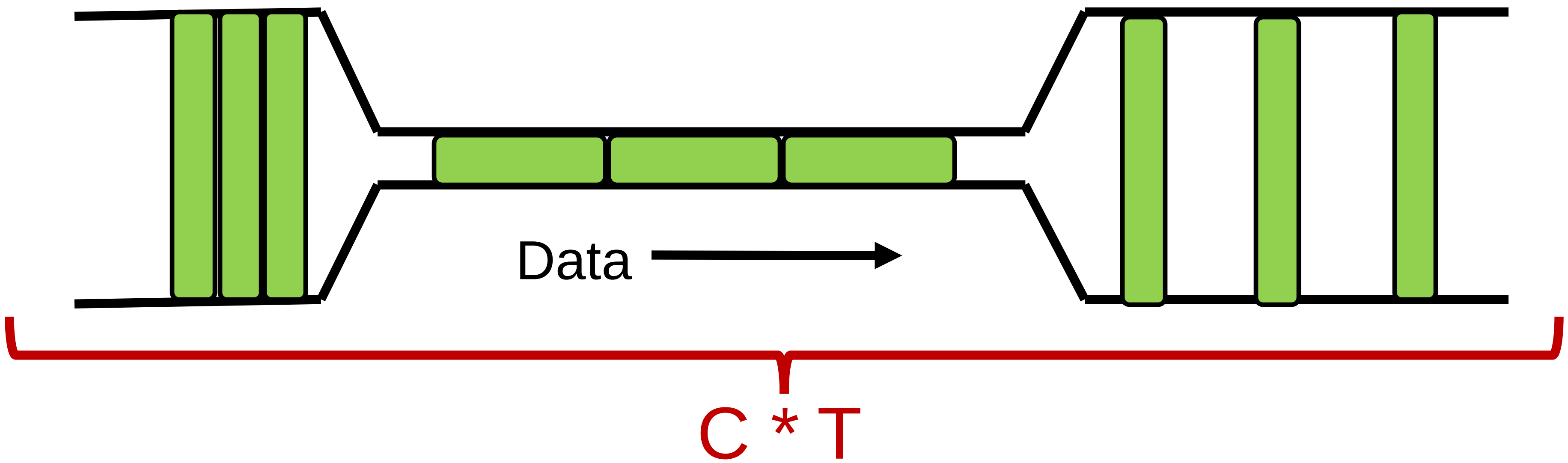
BDP < `cwnd` < BDP + B: persistent queue @ router

BDP + B < `cwnd`: packet drops

# The Bandwidth-Delay Product

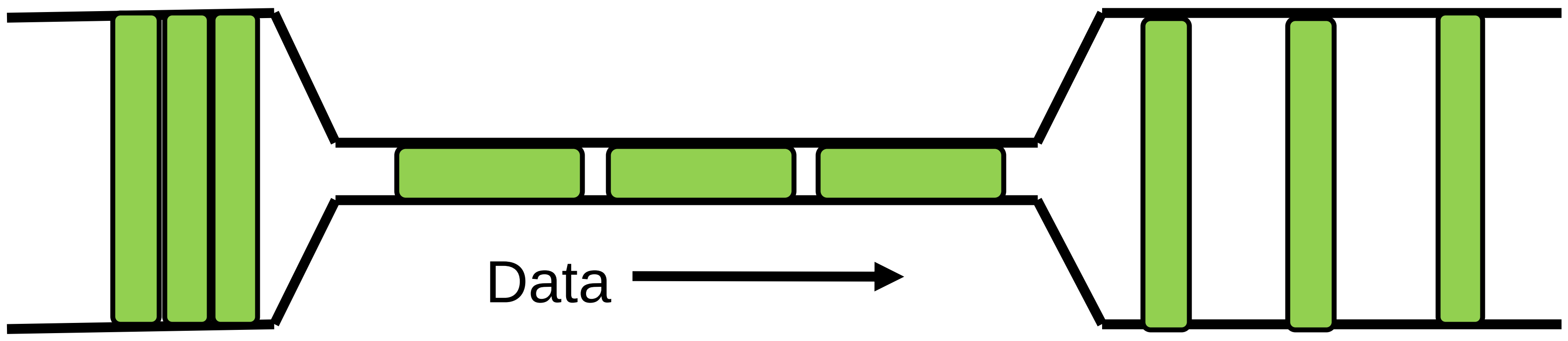
- C * T = bandwidth-delay product:
  - The amount of data in flight for a sender transmitting at the ideal rate during the ideal round-trip delay of a packet. (Assumed sole user of the link)

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



# The Bandwidth-Delay Product

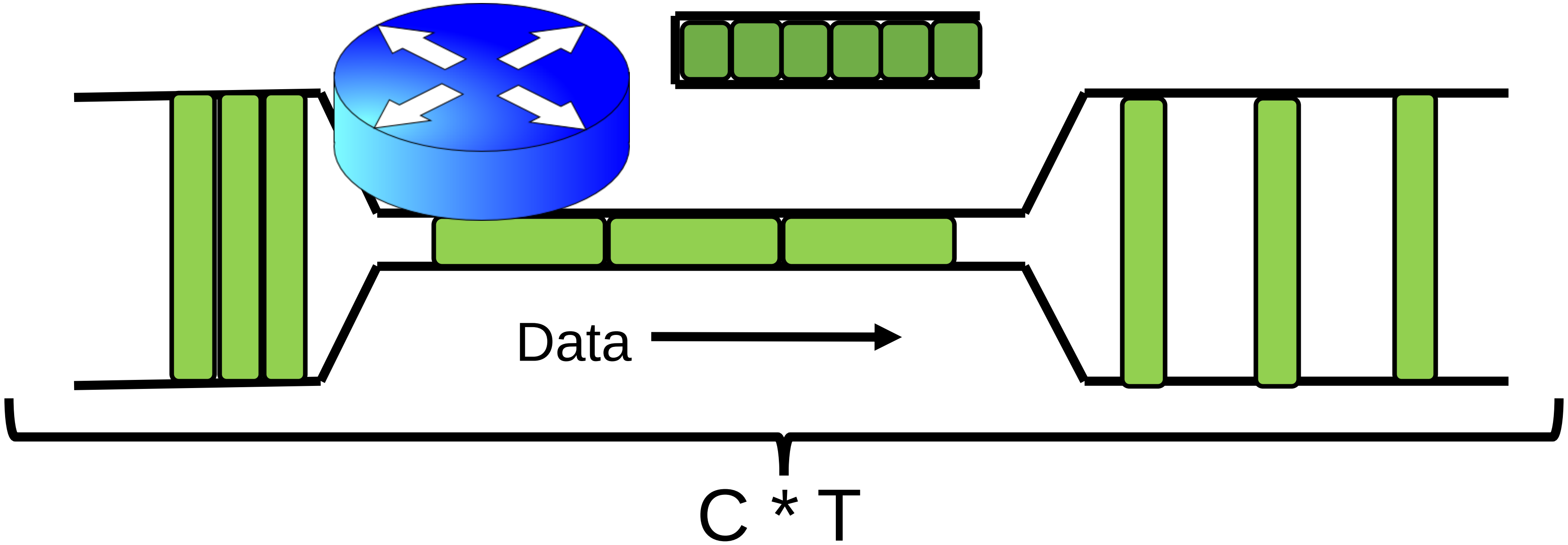
- Q: What happens if cwnd < C * T?
- A: Not sending back to back packets, link underused



# The Bandwidth-Delay Product

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

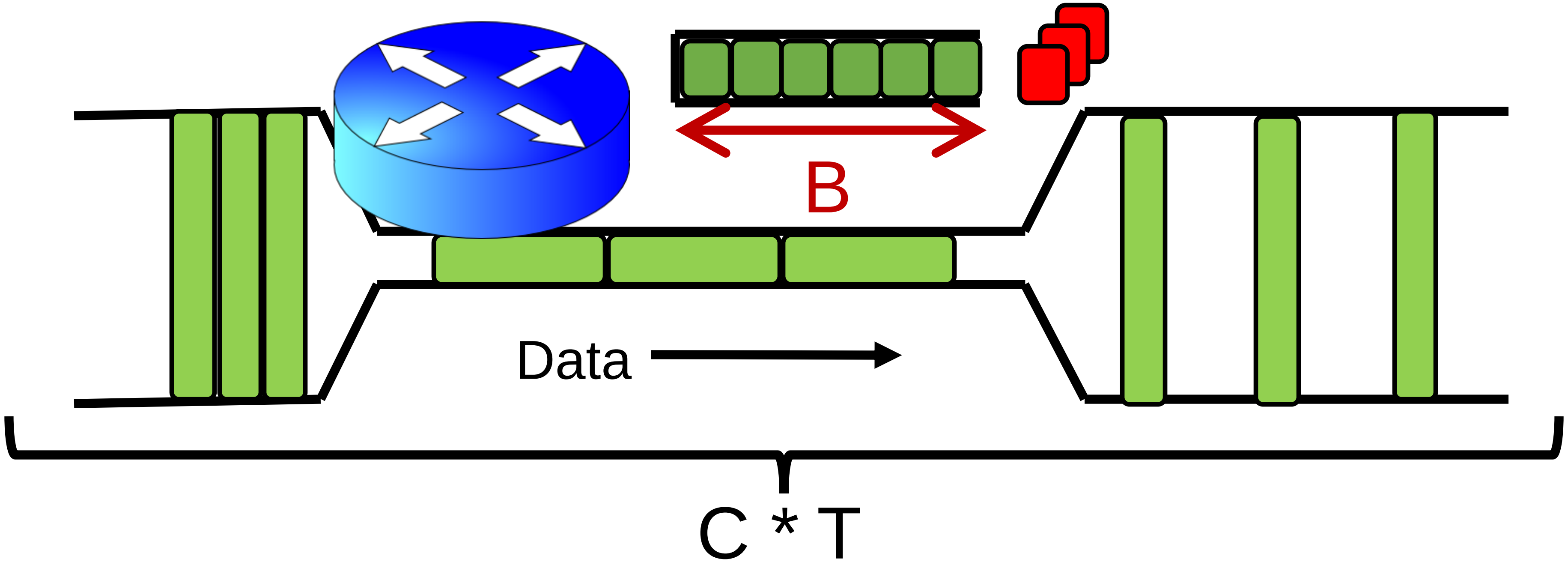
- A: Waiting at the bottleneck router queues



# 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!

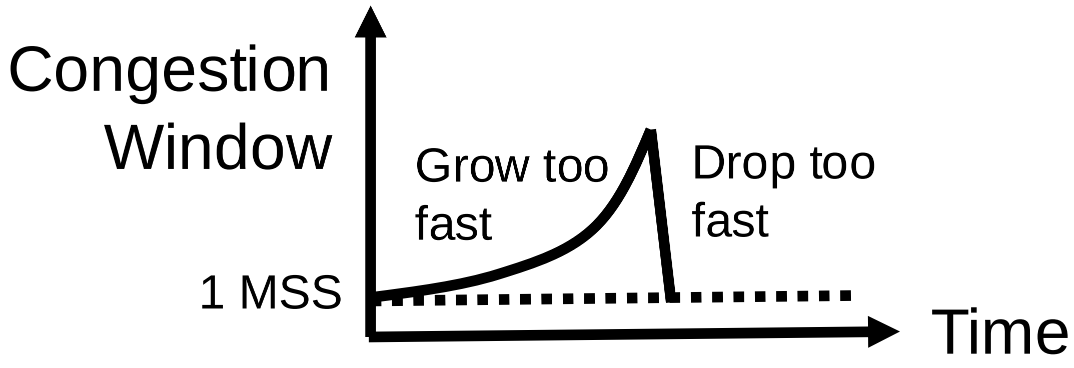


# Summary

- Bandwidth-Delay Product (BDP) governs the window size of a single connection 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, BDP is also the min socket buffer size to use the bottleneck link fully:
  - Important to set socket buffer sizes appropriately 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?

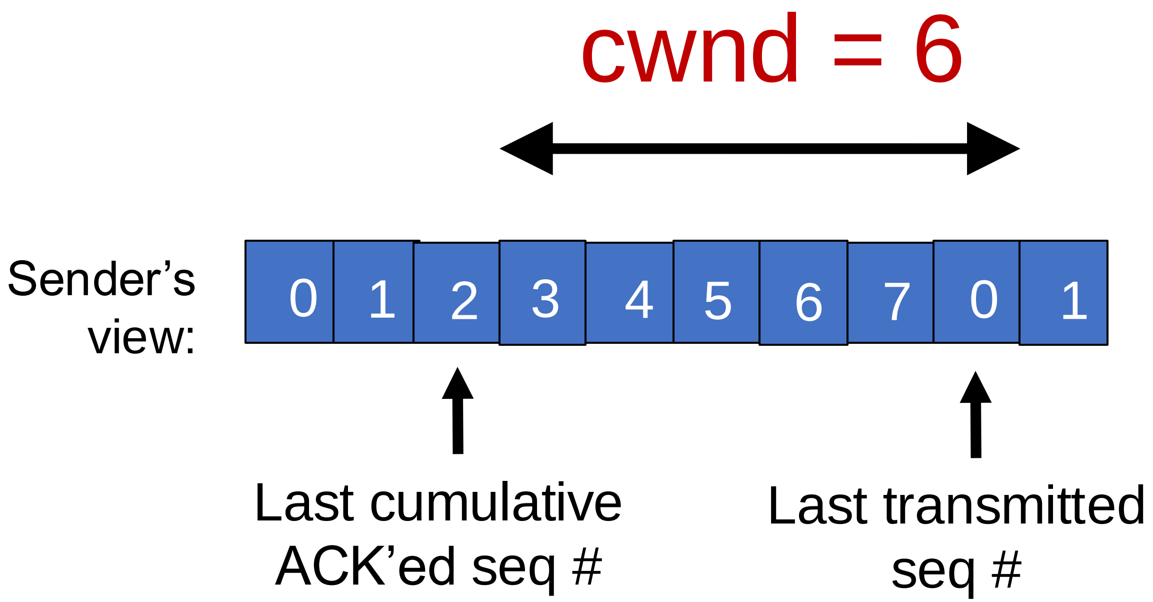
# Can we detect loss earlier than RTO?

- 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

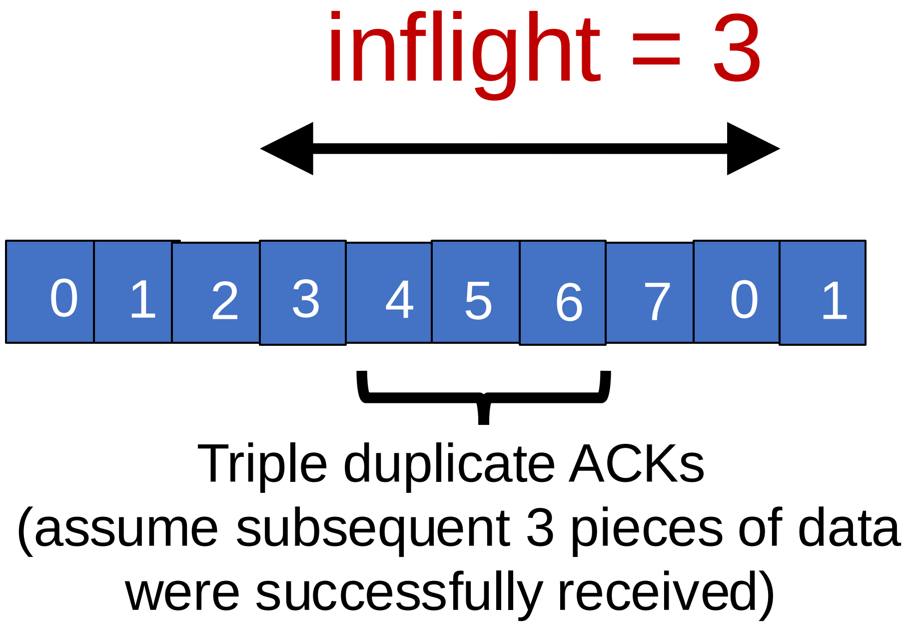
# Fast Retransmit & Fast Recovery

# Distinction: In-flight versus window

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




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




`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 performs two actions with fast retransmit

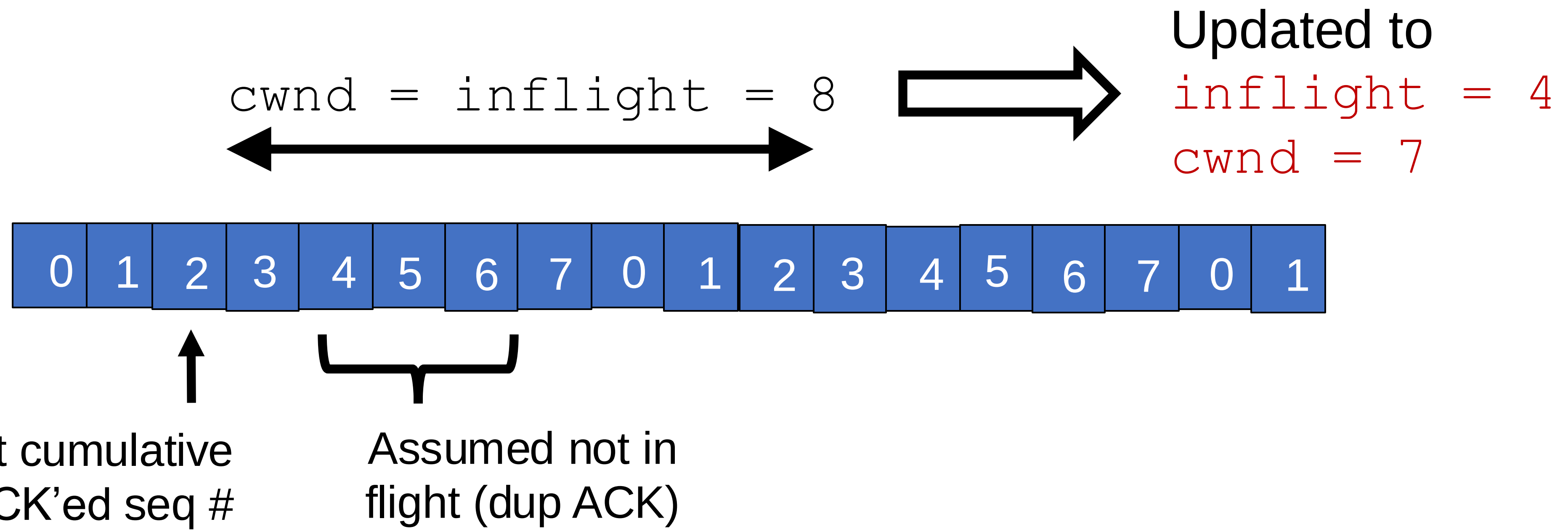
# TCP fast retransmit (RFC 2581)

- (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

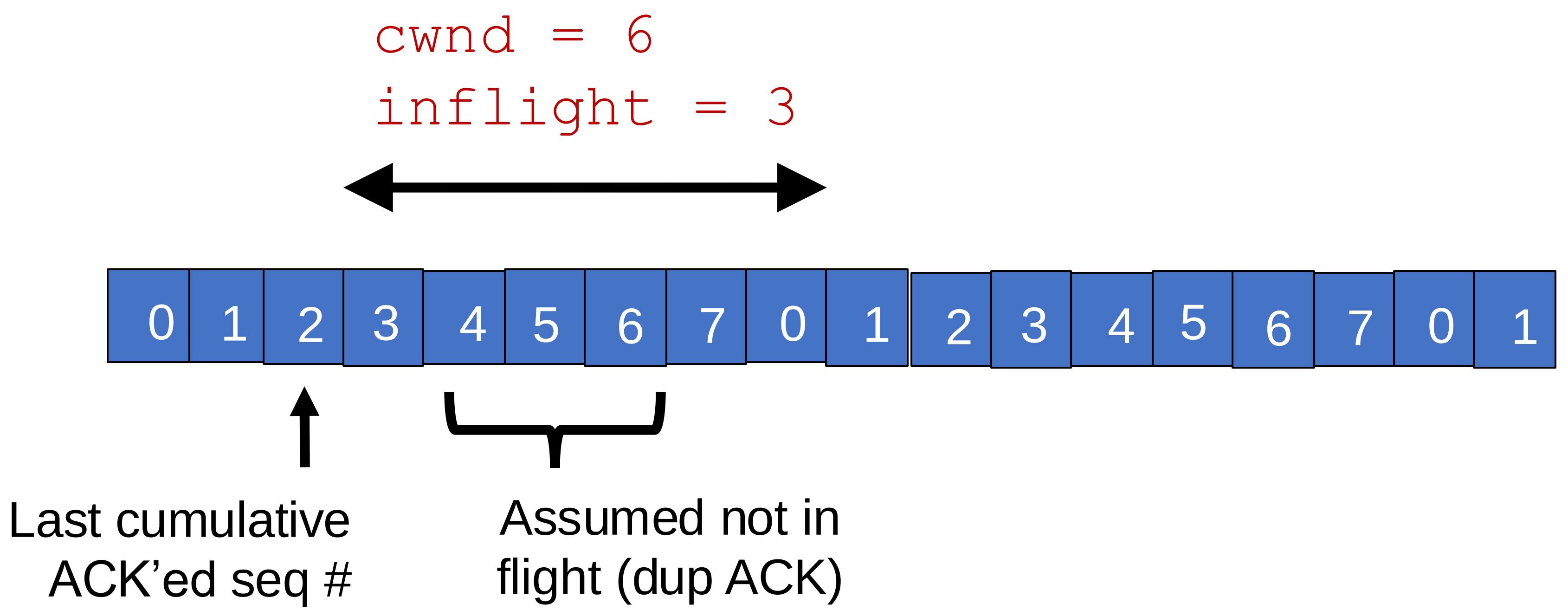
# TCP fast recovery (RFC 2581)

- 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 almost no data, if `cwnd` := 1)

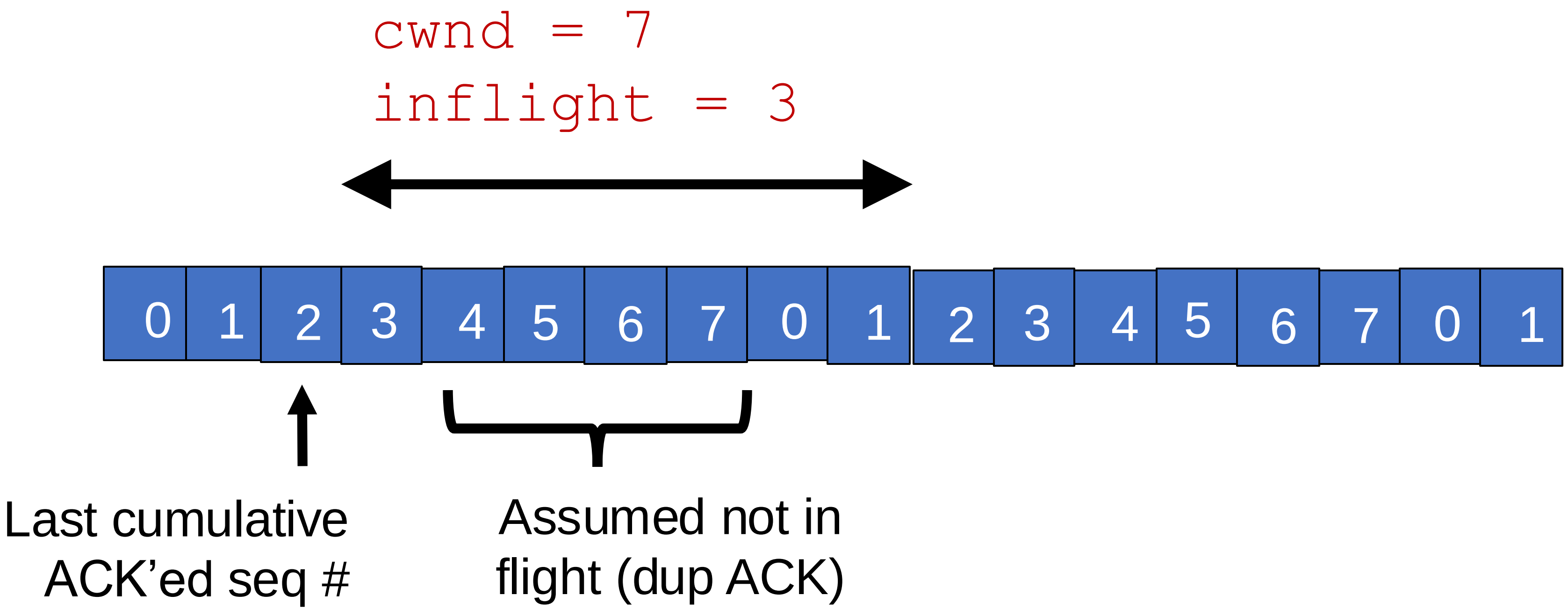
# TCP fast recovery (RFC 2581)

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



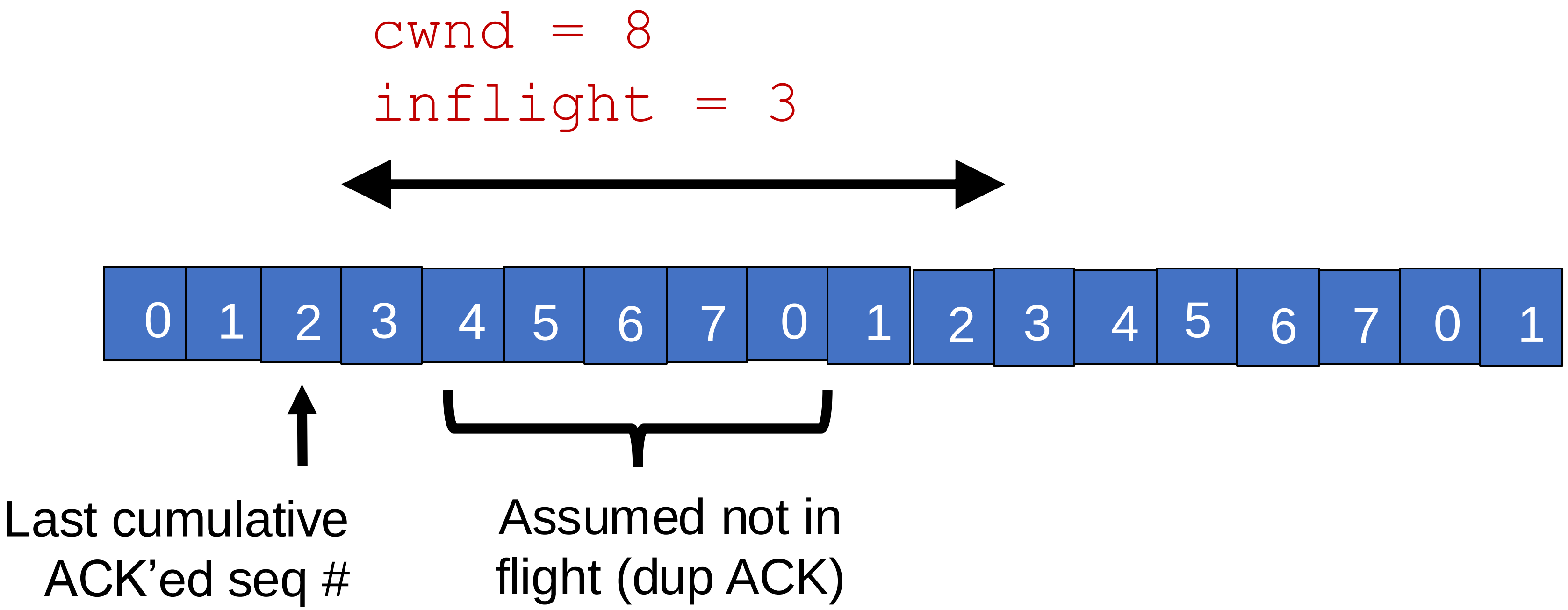
# TCP fast recovery (RFC 2581)

- 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

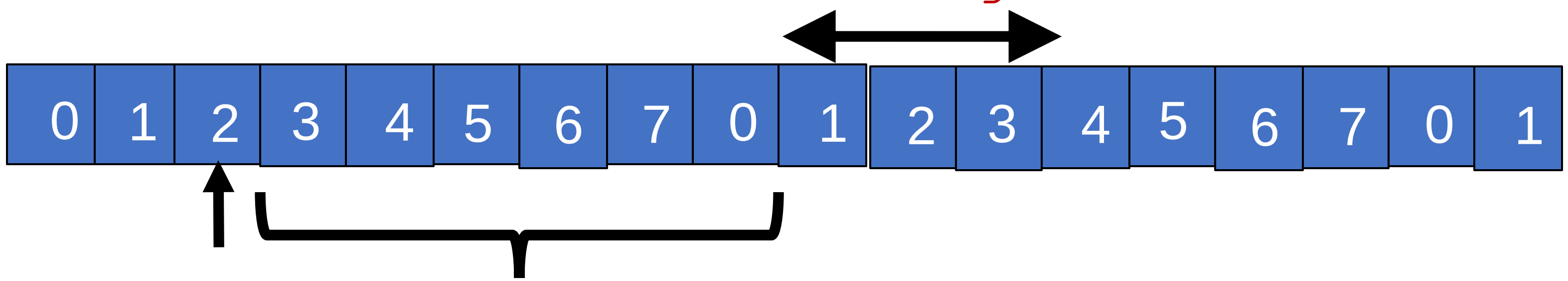


# TCP fast recovery (RFC 2581)

- 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!

`cwnd = 6/2 = 3`

`inflight = cwnd = 3`



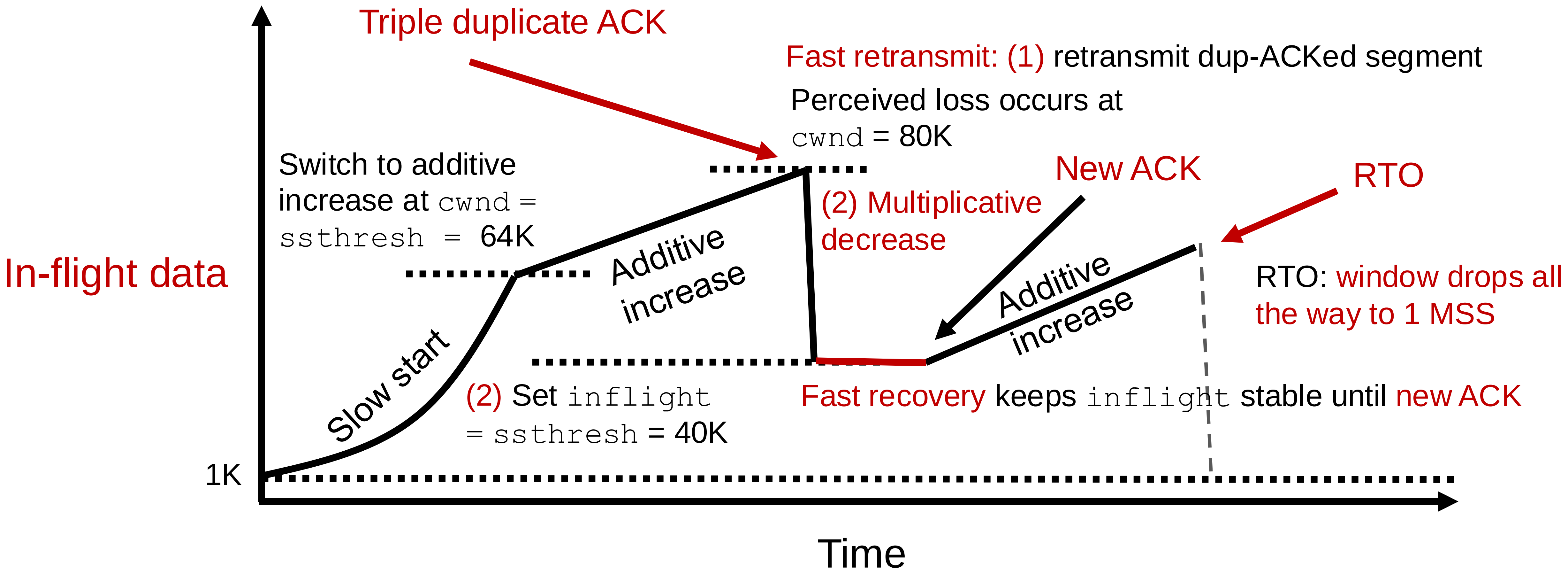

Last cumulative ACK’ed seq #

New ACK acknowledged this data

# 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.

[We didn’t cover this, but MD 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

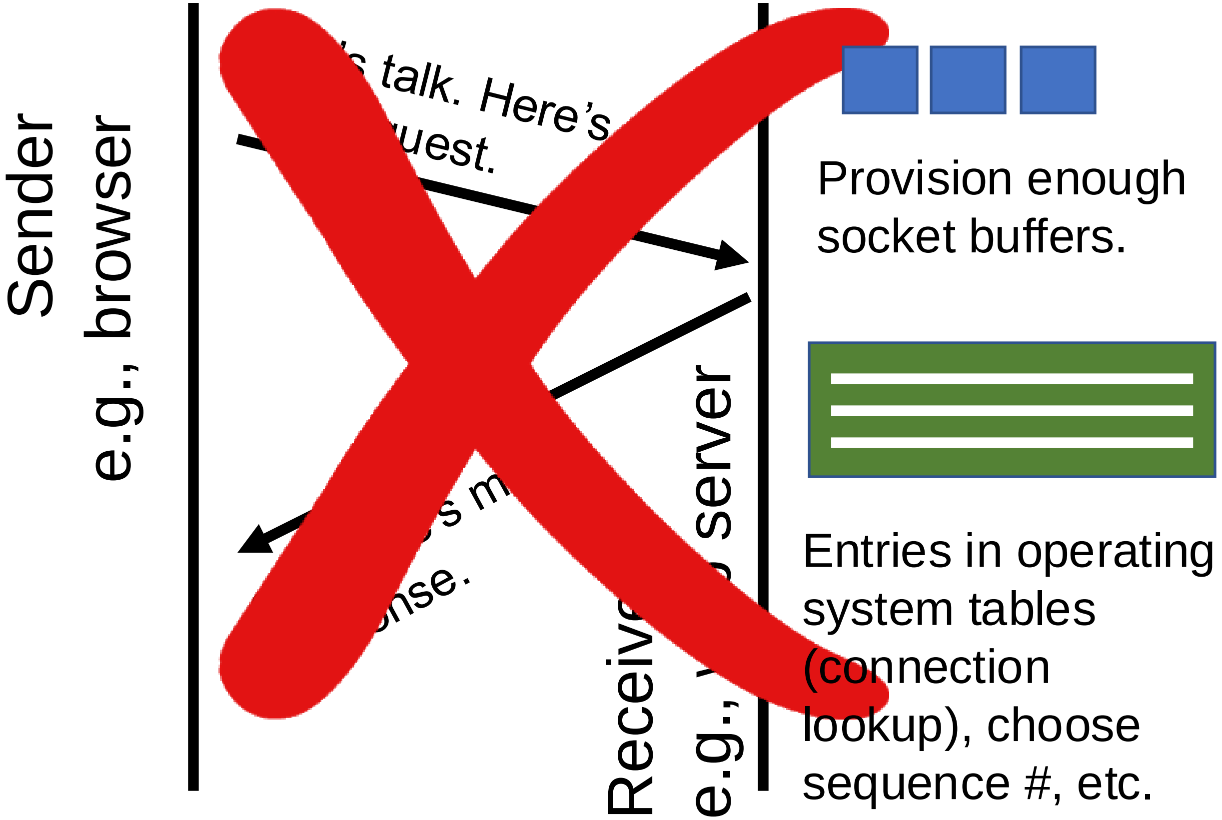
- Maintain this reduced amount of in-flight data as long as dup ACKs arrive
  - Data is successfully getting delivered
- When new ACK arrives, do additive increase from there on

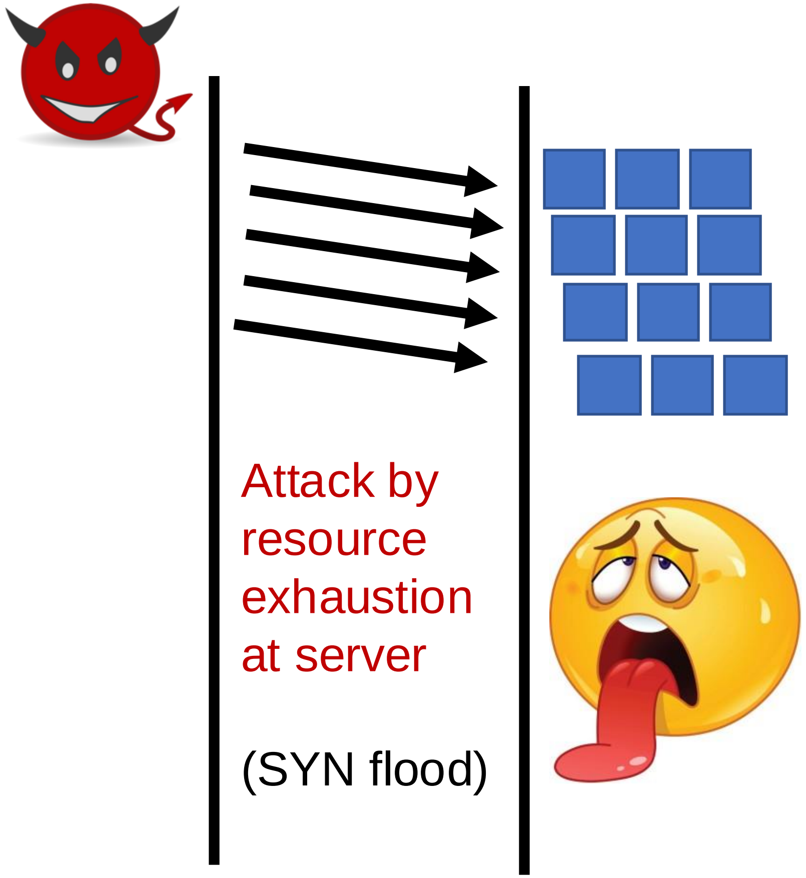
# Connection Management

How does a TCP connection start?

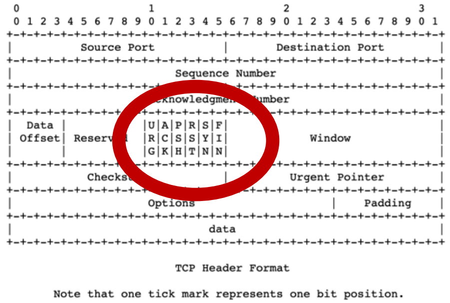
# Starting a TCP connection

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



# TCP 3-way handshake




## Client state

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

LISTEN

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

SYNSENT

ESTAB

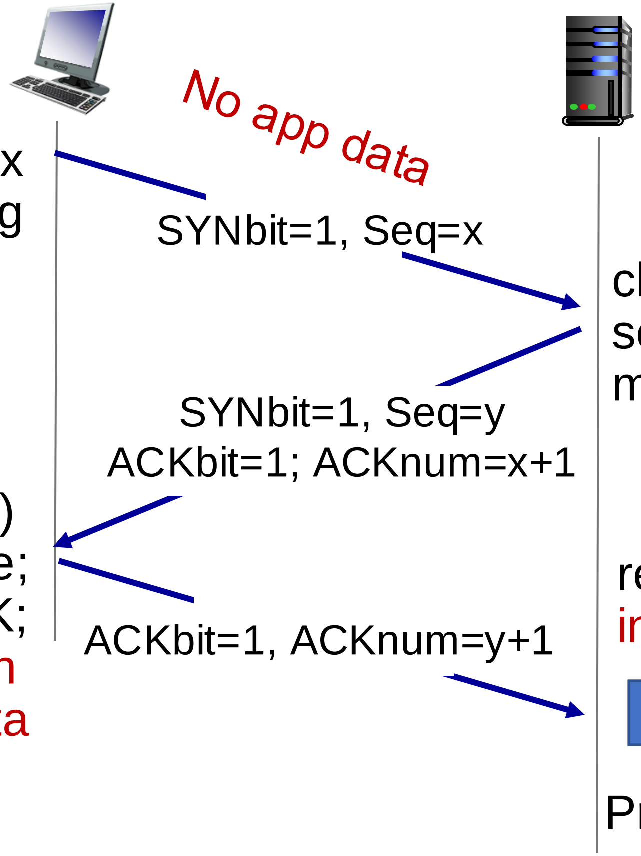
## Server state

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

LISTEN

SYN RCVD

ESTAB




choose init seq num, x
send TCP SYN msg

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

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

received ACK(y)
indicates client is live

Provision resources

# 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

- “Recent” measures to address the startup delay
  - TCP fast open
  - QUIC