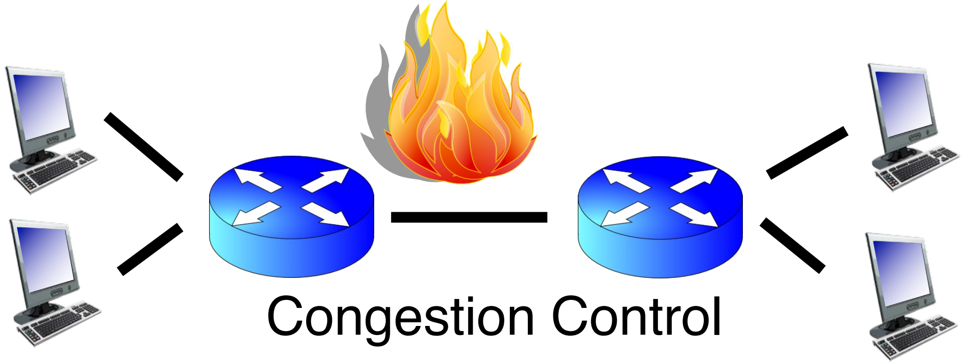
CS 352
Transport: Wrap-Up

Lecture 18

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

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

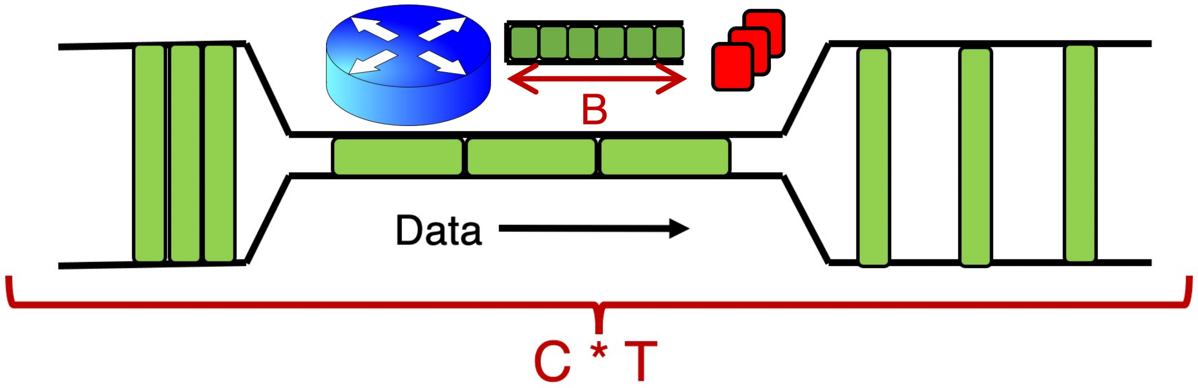


Sense and React

Signals

Knobs

Bandwidth-Delay Product

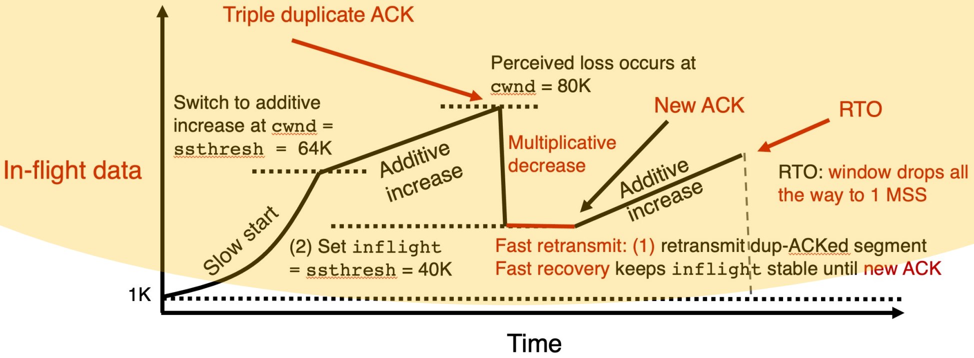


TCP New Reno

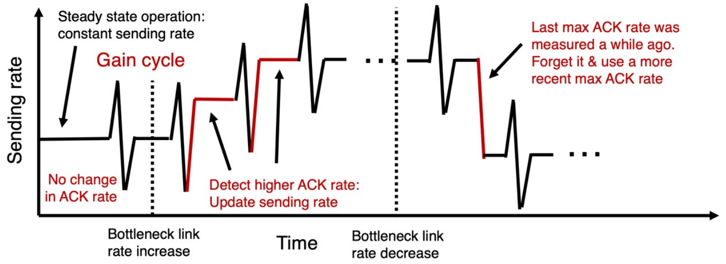
= slow start

+ congestion avoidance (AI)

+ fast retransmit & recovery (MD)



TCP BBR: Gain cycling



cwnd < BDP: sender under-uses the link

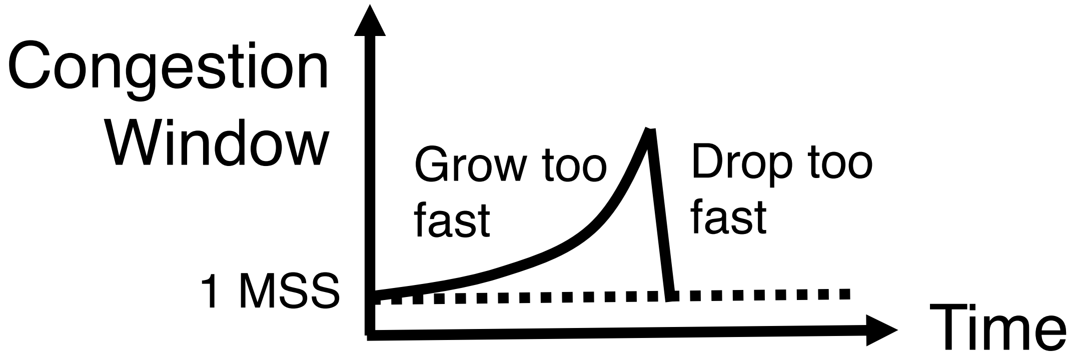
BDP = cwnd: 100% link use, zero queues (ideal)

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

BDP + B < cwnd: packet drops

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?

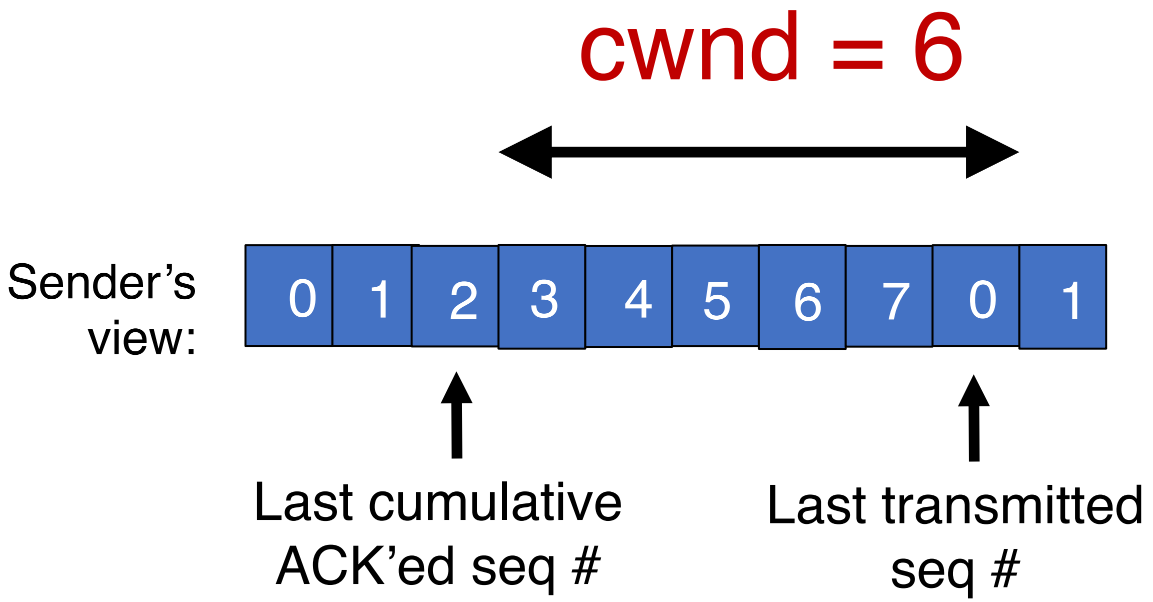
Can we detect loss earlier than RTO?

- 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

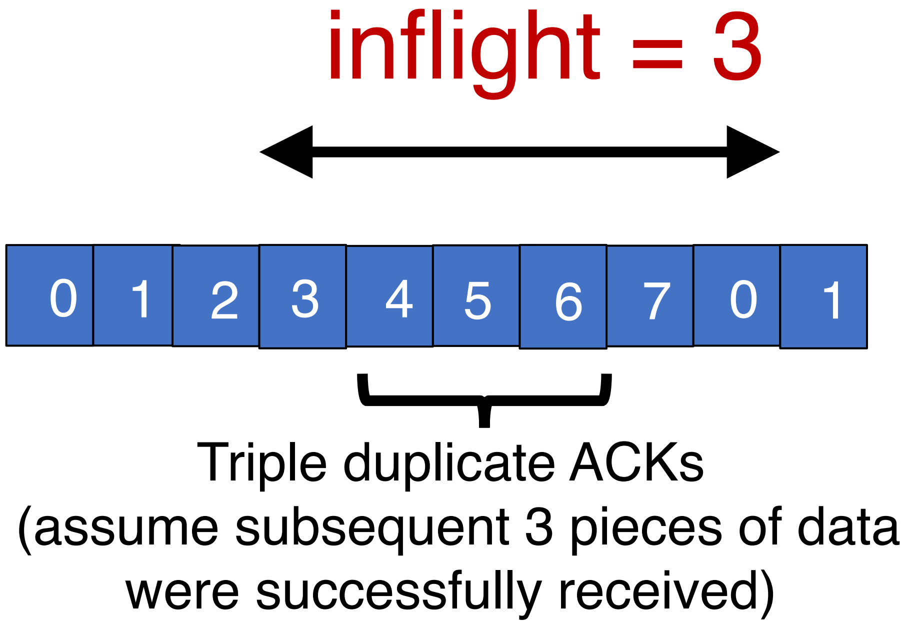
Fast Retransmit & Fast Recovery

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, albeit with some loss.
- Note: Before the dup ACKs arrive, we assume `inflight = cwnd`

- TCP sender does two actions with fast retransmit

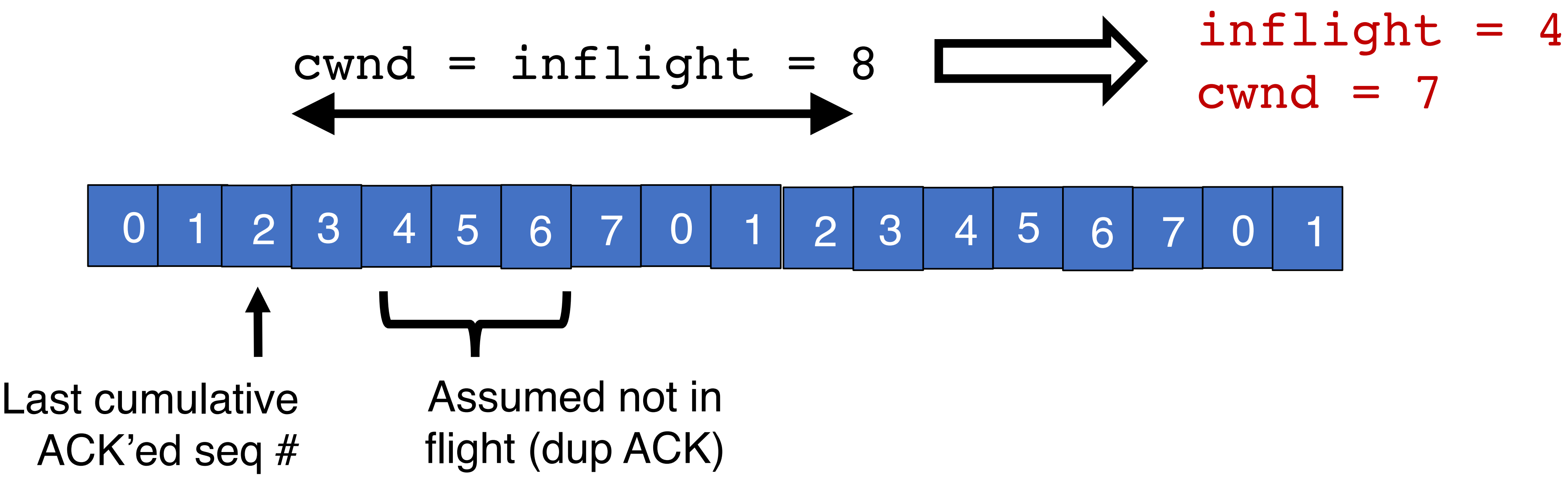
TCP fast retransmit (RFC 2581)

- (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

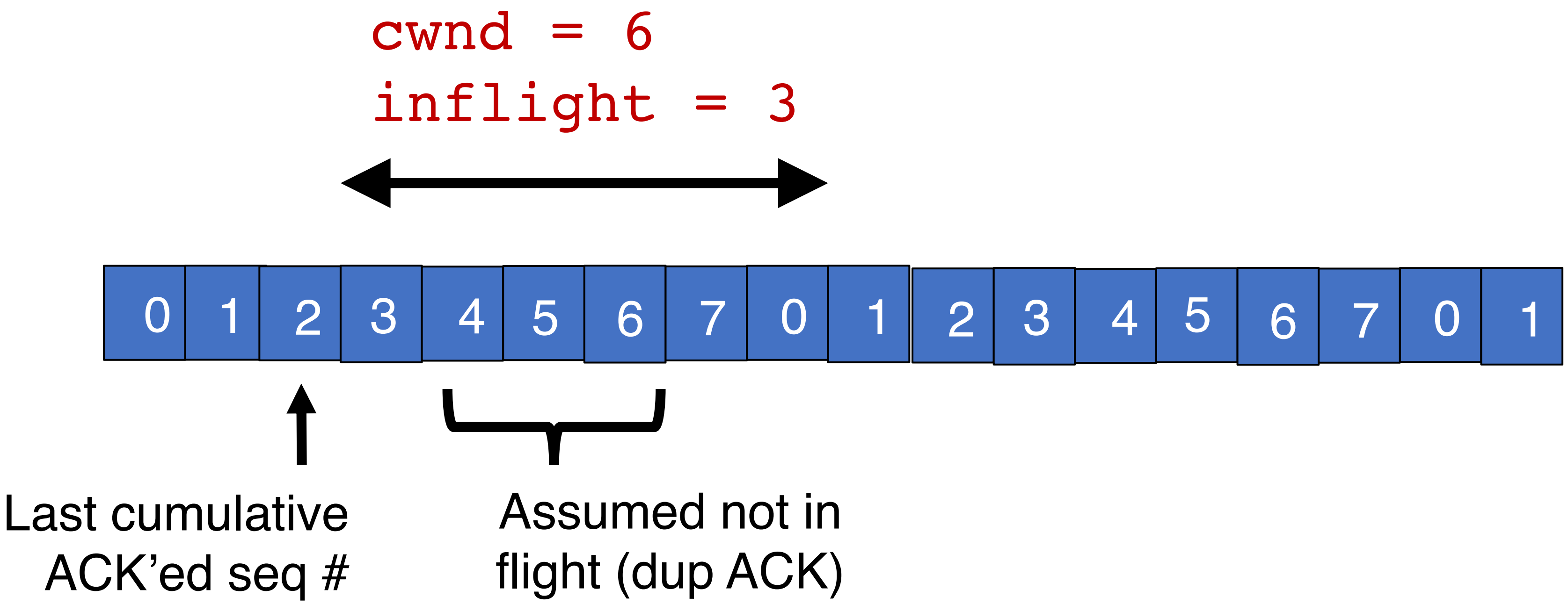
TCP fast recovery (RFC 2581)

- 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)

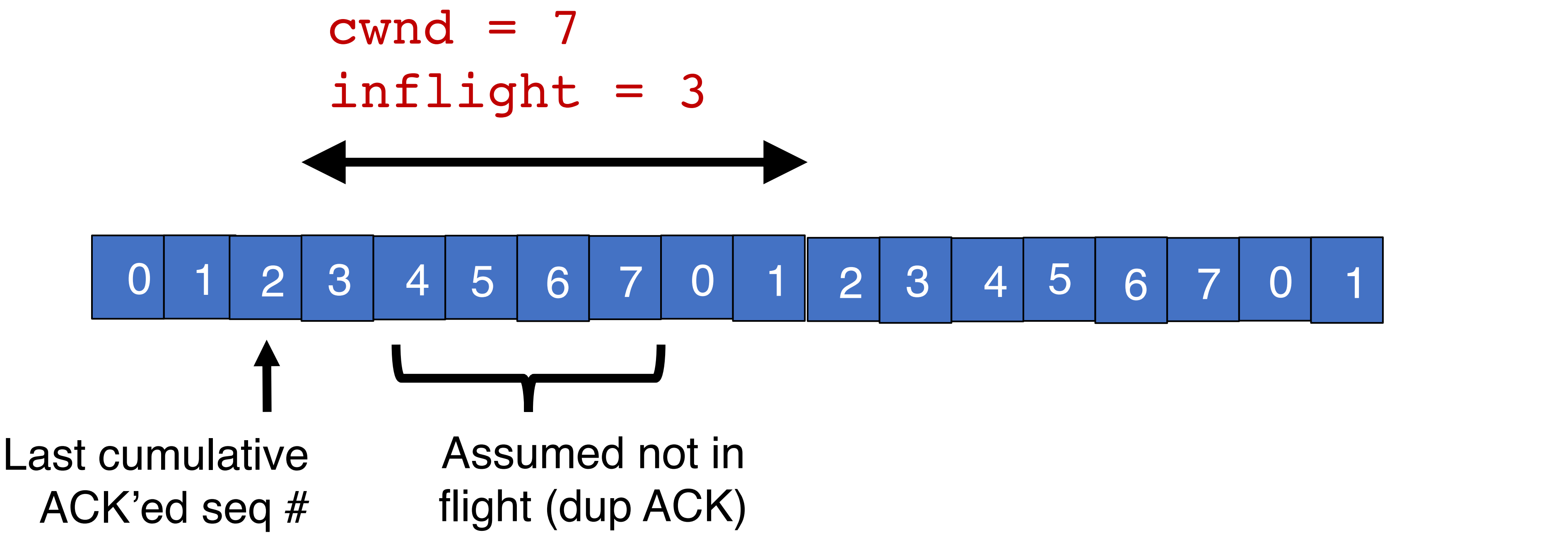
TCP fast recovery (RFC 2581)

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



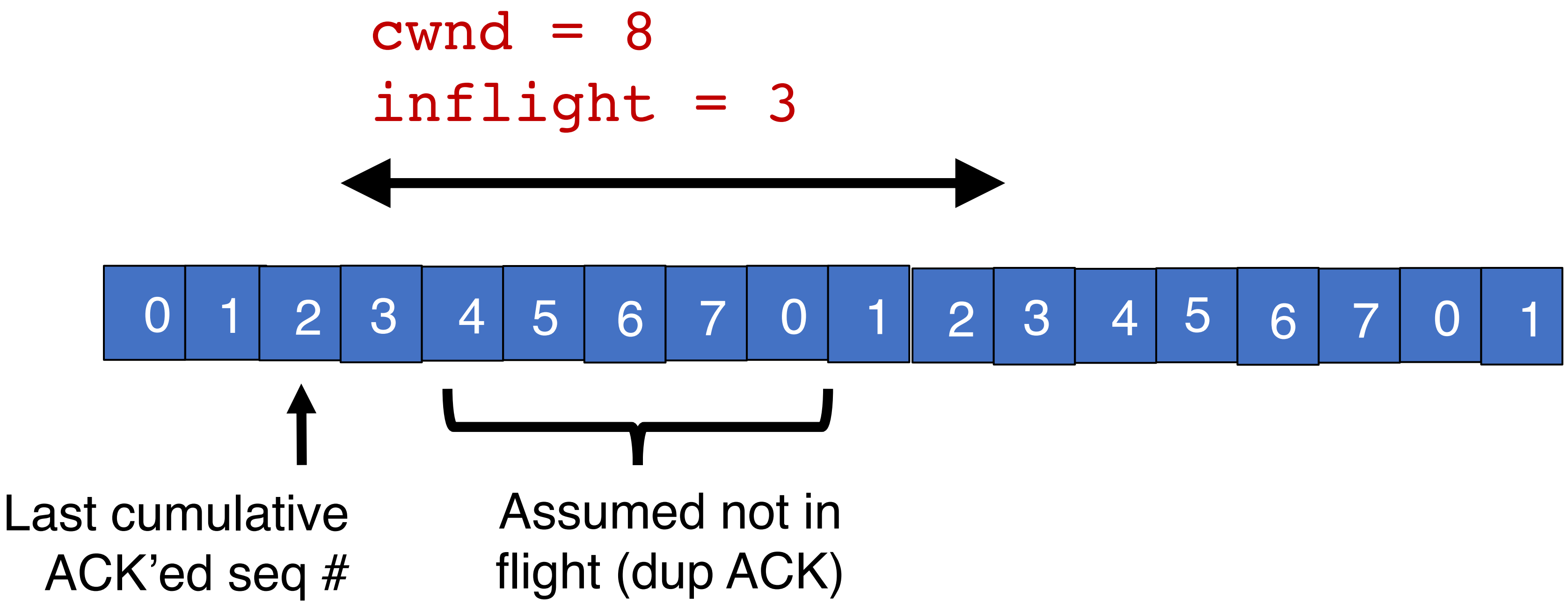
TCP fast recovery (RFC 2581)

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



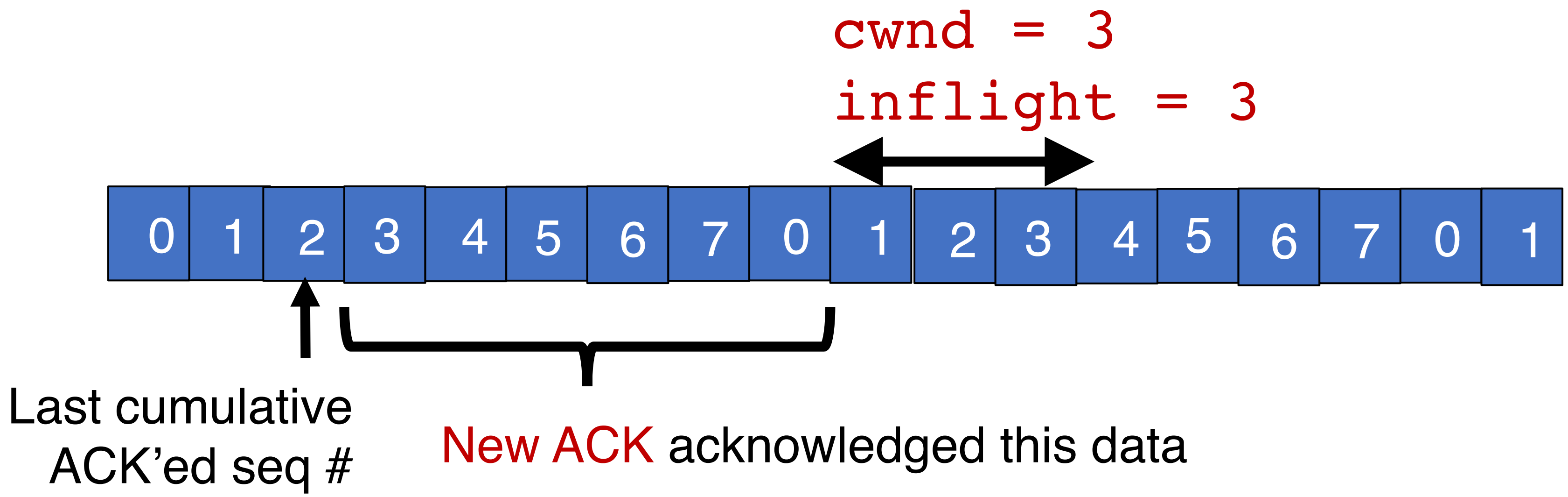
TCP fast recovery (RFC 2581)

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



TCP fast recovery (RFC 2581)

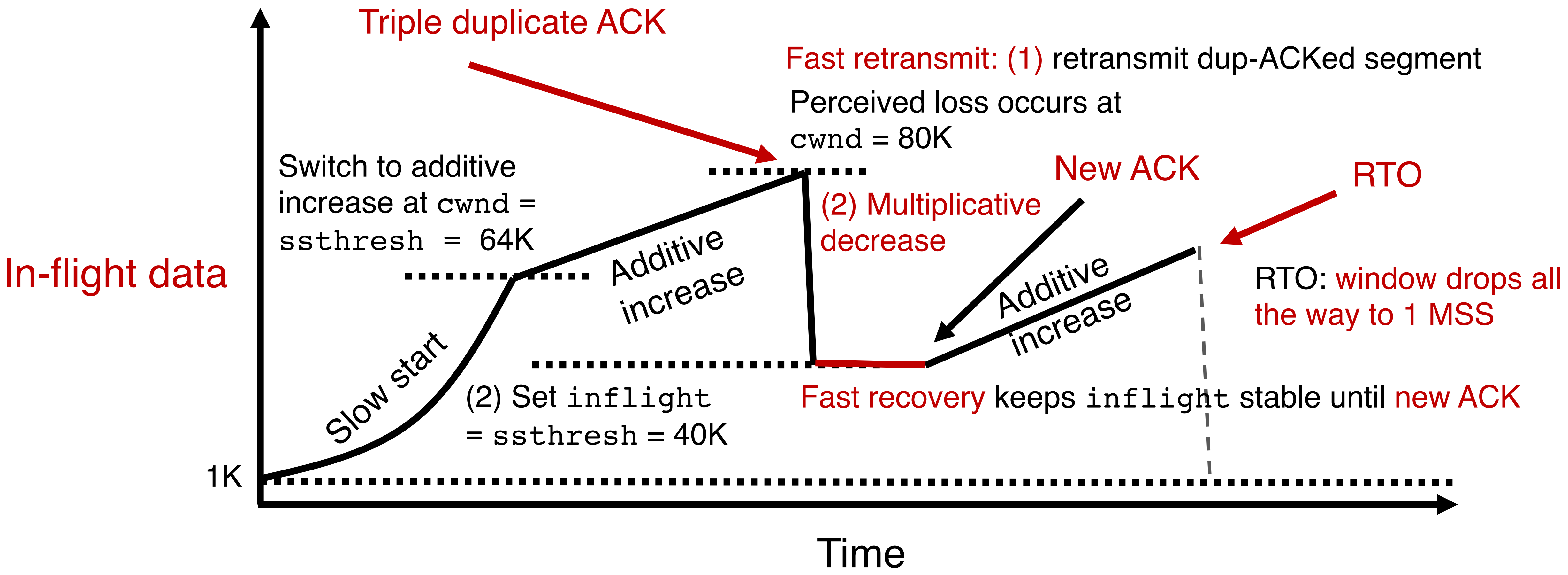
- Eventually a new ACK arrives, acknowledging the retransmitted data and all data in between
- Deflate `cwnd` to half of `cwnd` before fast retransmit.
  - `cwnd` and `inflight` are aligned and equal once again
- Perform additive increase from this point!



Additive Increase/Multiplicative Decrease

Say `MSS` = 1 KByte

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



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

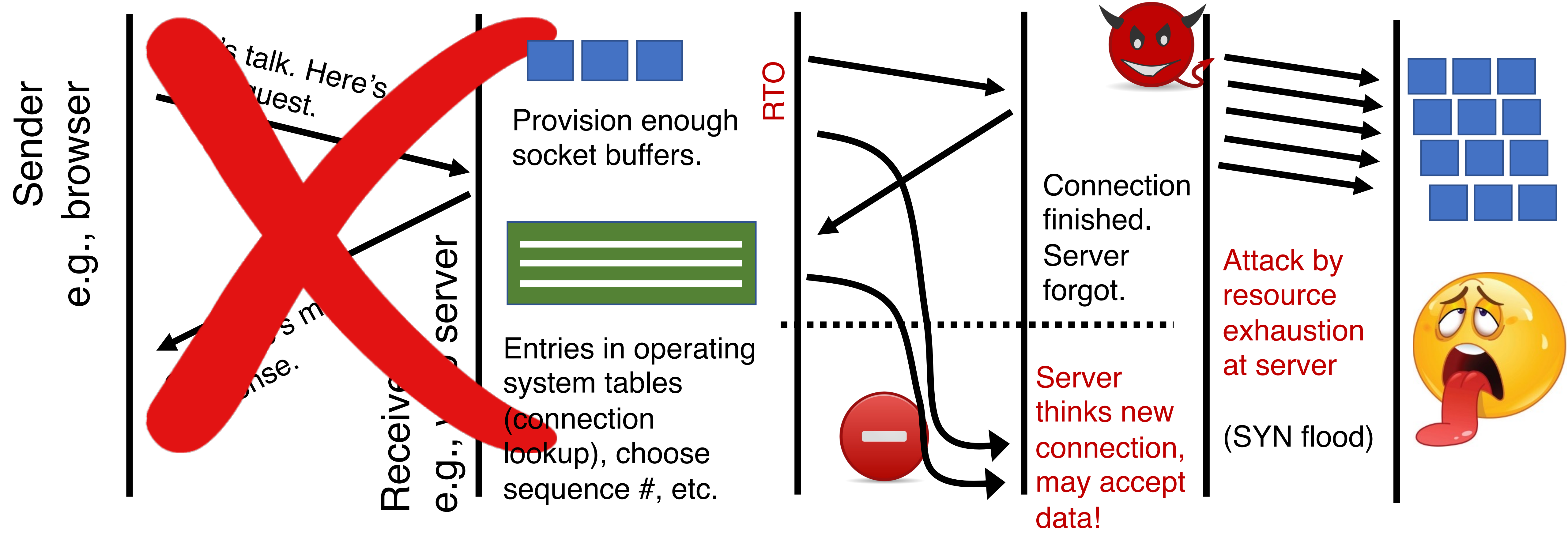
- Maintain this reduced amount of in-flight data as long as dup ACKs arrive
  - Data is successfully getting delivered
- When new ACK arrives, do additive increase from there on

Connection Management

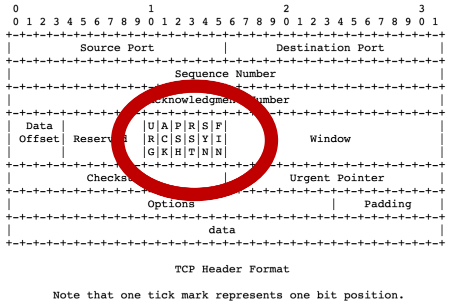
How does a TCP connection start?

Starting a TCP connection

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



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

No app data

choose init seq num, x
send TCP SYN msg

SYNbit=1, Seq=x

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

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

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

ACKbit=1, ACKnum=y+1

received ACK(y)
indicates client is live

Provision resources

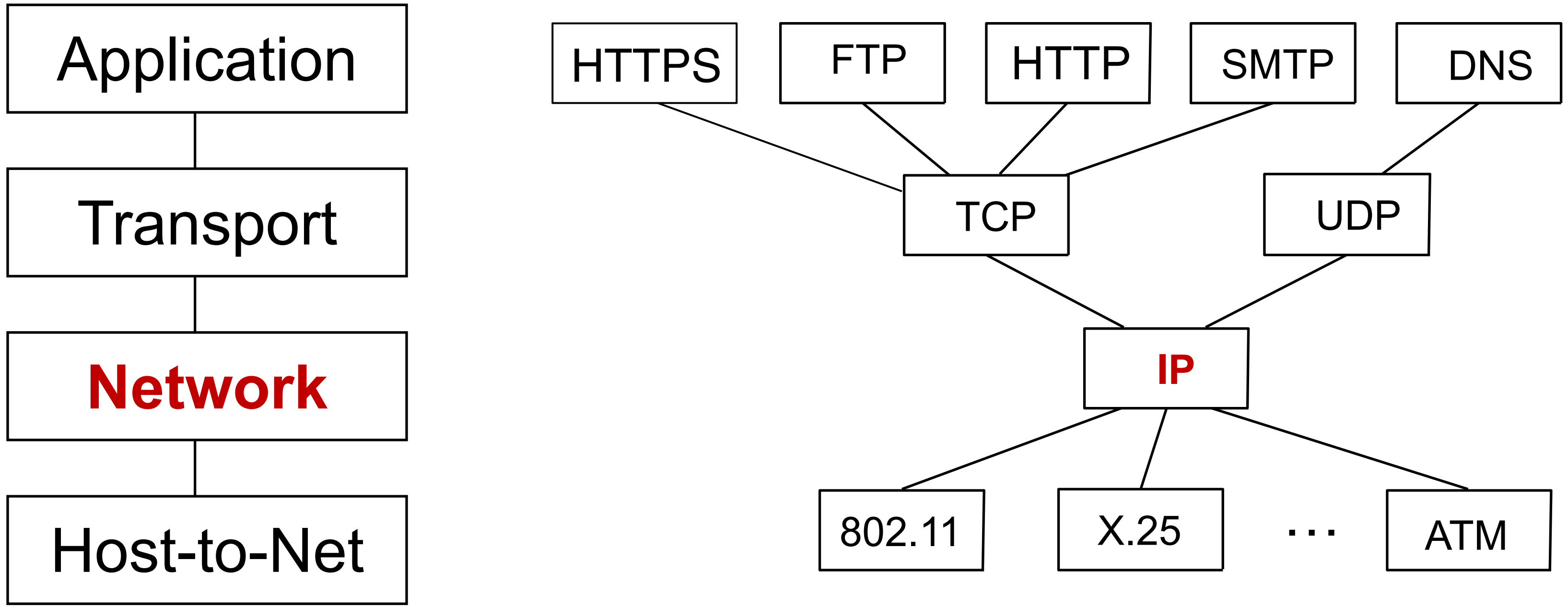
Implications of 3-way handshake

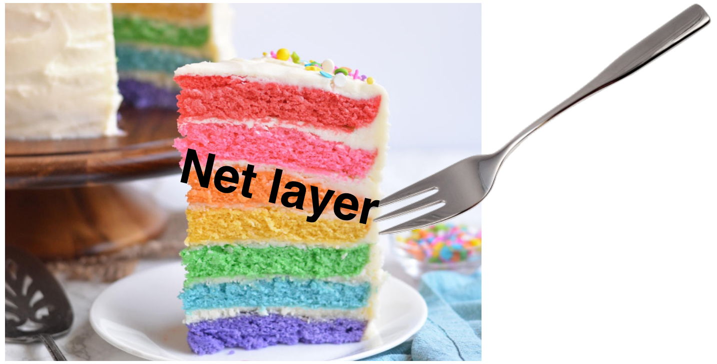
- Any application data can only be sent an RTT after
- Fresh connection: at least 2 RTTs to get a response
  - Often fruitful to use “persistent” connections (HTTP header)

The Transport Layer

- Communication abstraction between processes
- Intelligent endpoints implementing guarantees for applications

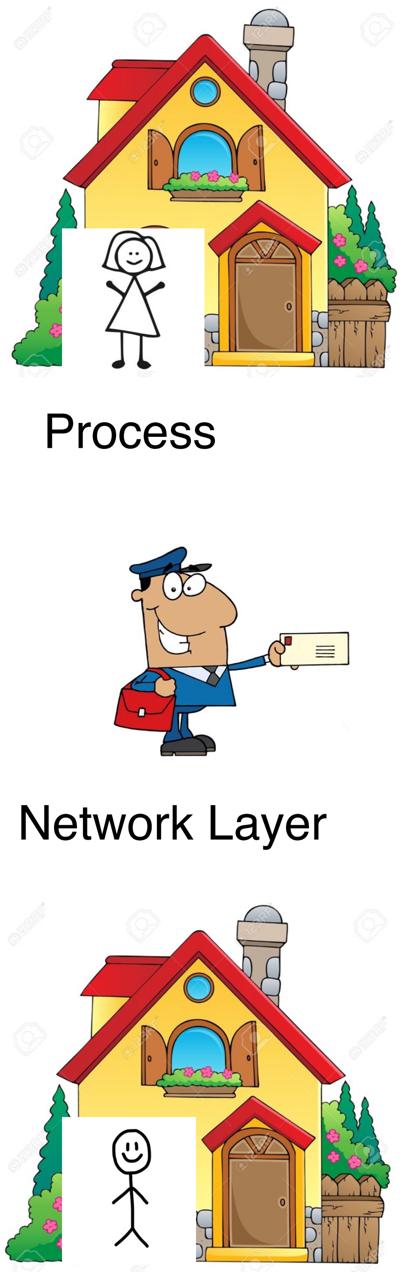
Network


Application
Transport
Network
Host-to-Net
HTTPS
FTP
HTTP
SMTP
DNS
TCP
UDP
IP
802.11
X.25
...
ATM


Net layer

The network layer

- Main function: Move data from sending to receiving endpoint
- on sending endpoint: encapsulate transport segments into datagrams
- on receiving endpoint: deliver datagrams to transport layer
- The network layer also runs in every router
  - Very challenging to evolve the network layer
- Routers examine headers on all passing through them



Two key network-layer functions

- Forwarding: move packets from router's input to appropriate router output

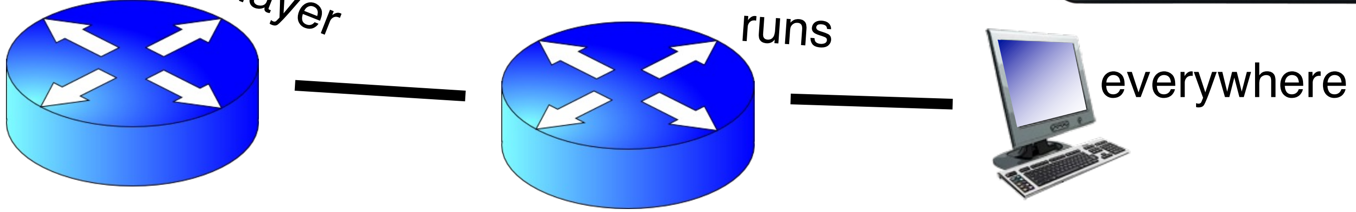
- Routing: determine route taken by packets from source to destination
  - routing algorithms

- The network layer solves the routing problem.

Analogy: taking a road trip

- Forwarding: process of getting through single interchange

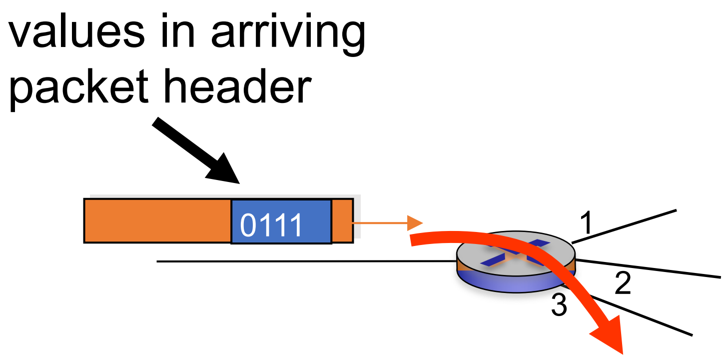
- Routing: process of planning trip from source to destination



Data plane and Control Plane

Data plane = Forwarding

- local, per-router function
- determines how datagram arriving on router input port is forwarded to router output port



Control plane = Routing

- network-wide logic
- determines how datagram is routed along end-to-end path from source to destination endpoint
- two control-plane approaches:
  - Distributed routing algorithm running on each router
  - Centralized routing algorithm running on a (logically) centralized server