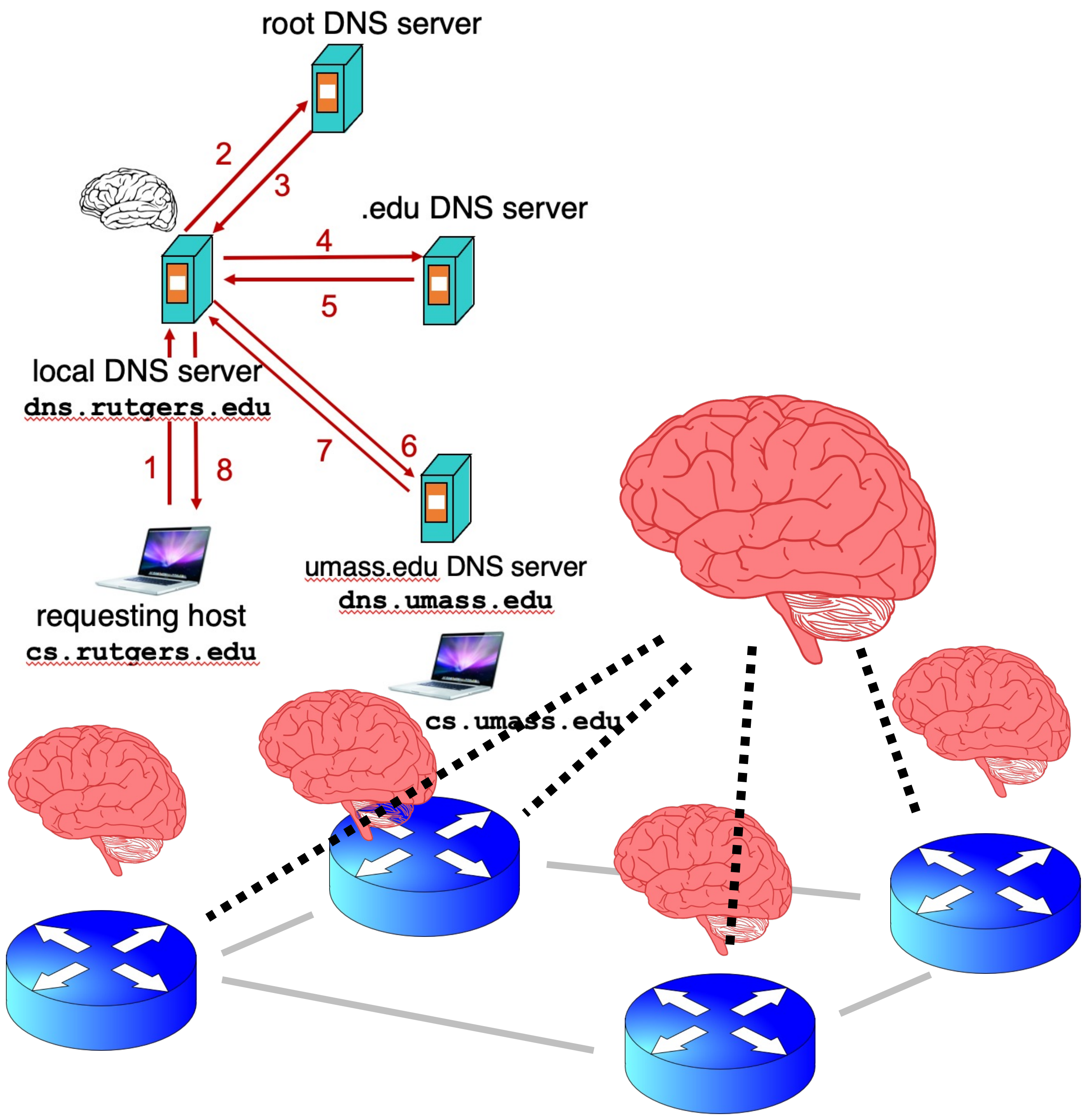
Transport

RUTGERS
UNIVERSITY | NEW BRUNSWICK

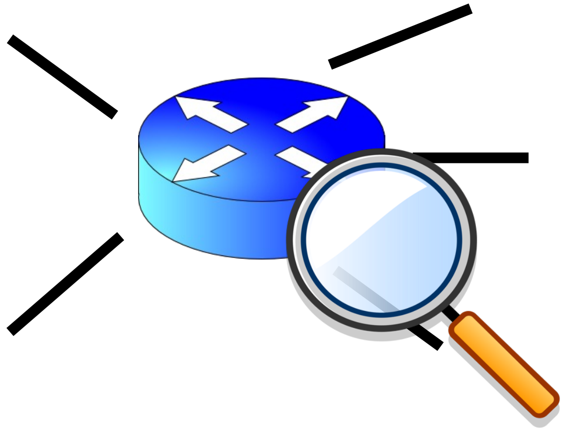
Some fundamental problems

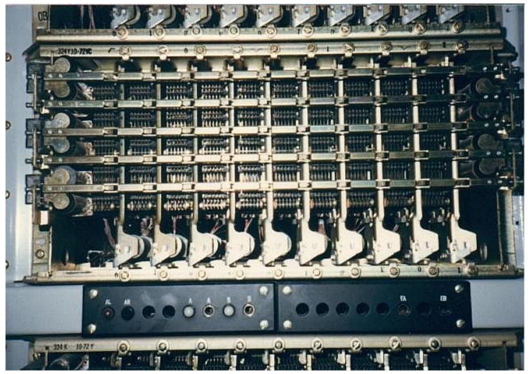
Problems so far

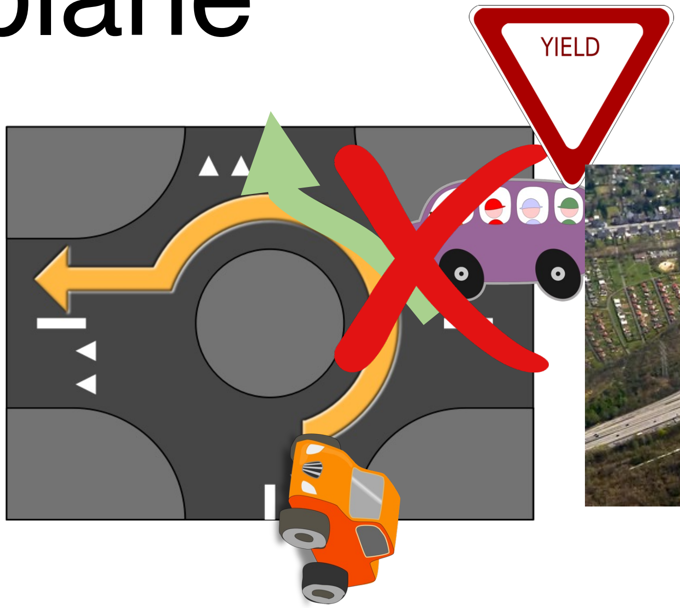
- (0) Name resolution
- (1) Routing
    - Control plane, data plane
    - routing, forwarding

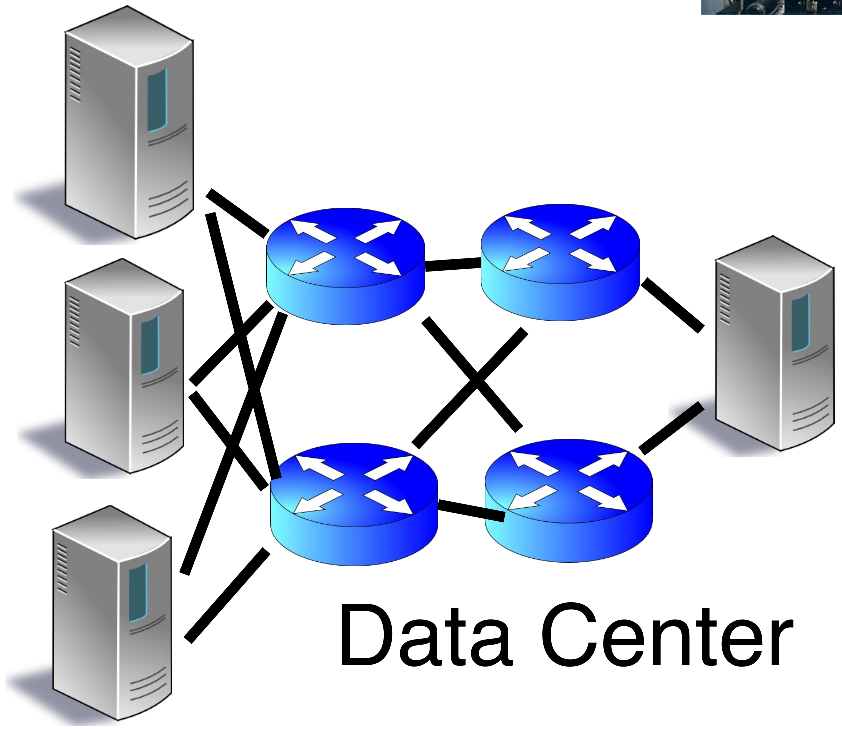


(2) High-speed data plane









- Transport won't help if the network has choke points: e.g., routers
- How to design high-speed hardware routers?
- How to design high-speed software routers?
- Data centers, middleboxes

In general, networks give no guarantees

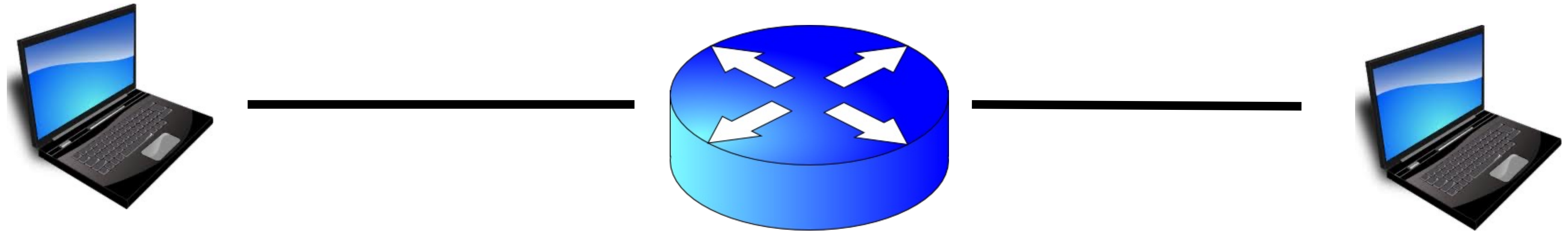
- Packets may be lost, corrupted, reordered, on the way to the destination
  - Best effort delivery

- Advantage: The network becomes very simple to build
  - Don’t have to make it reliable
  - Don’t need to implement any performance guarantees
  - Don’t need to maintain packet ordering
  - Almost any medium can deliver individual packets
    - Example: RFC 1149: “IP Datagrams over Avian Carriers”

- Early Internet thrived: easy to engineer, no guarantees to worry about

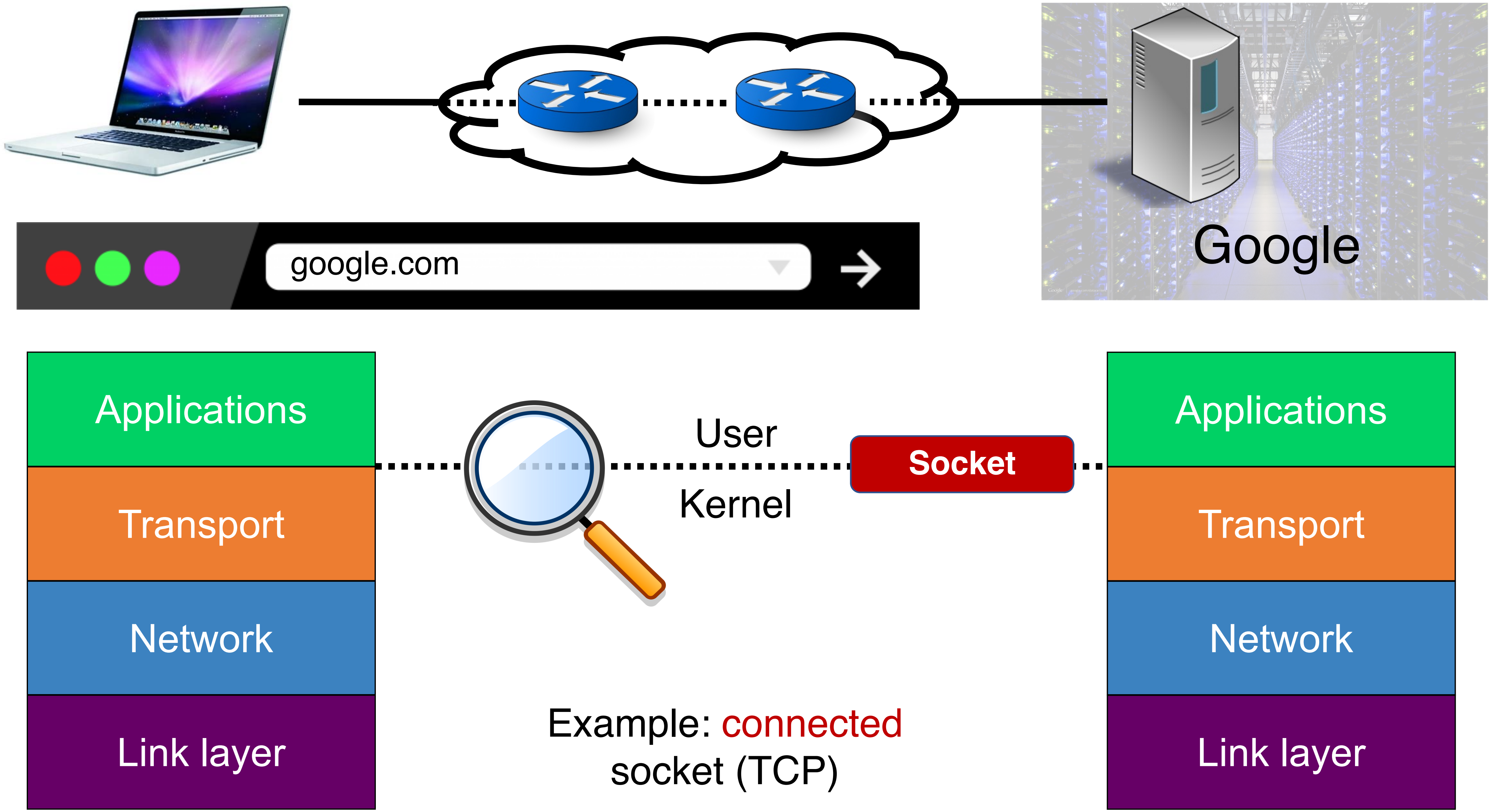
(3) Providing guarantees for applications

- How should endpoints provide guarantees to applications?

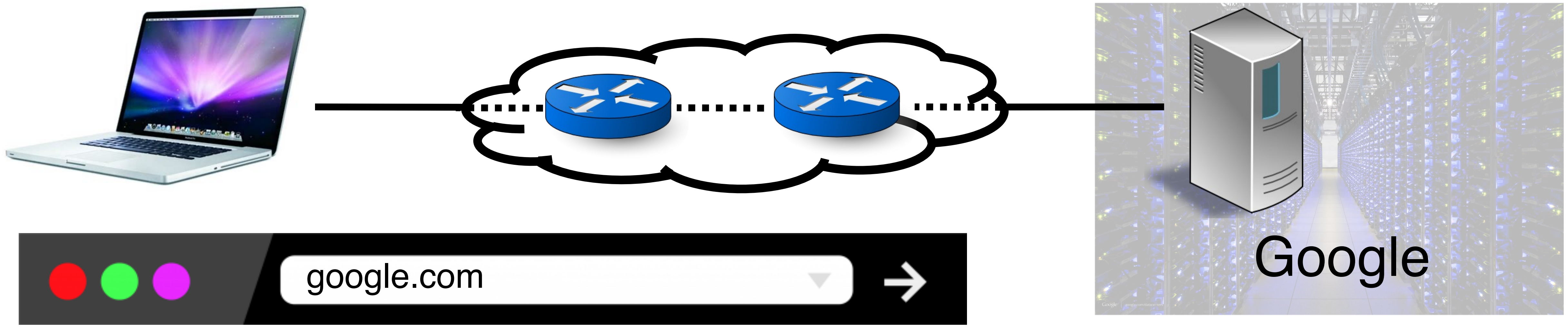


- Transport software on the endpoint oversees implementing guarantees on top of an unreliable network
- Semantics are per "conversation" and agnostic to app data
- Reliable delivery, ordered delivery, fair sharing of resources
- Two popular transports: TCP, UDP
  - (there are others)

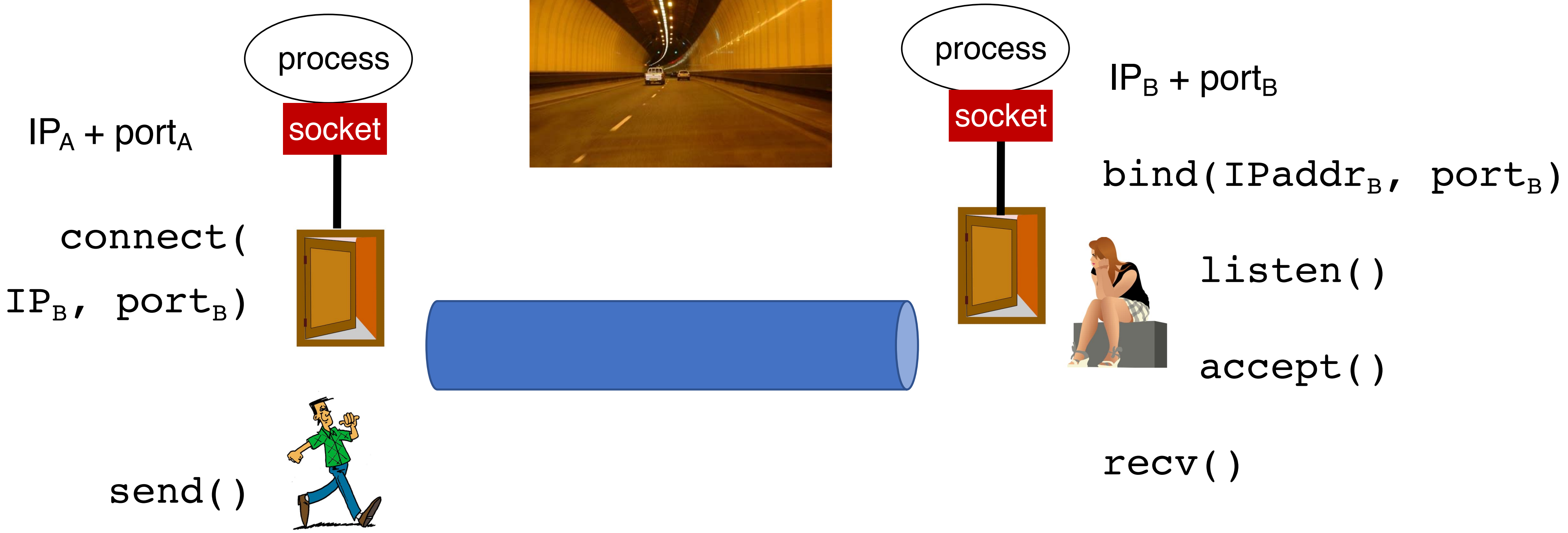
Application-OS interface



Example: connected socket (TCP)


google.com
Google

TCP

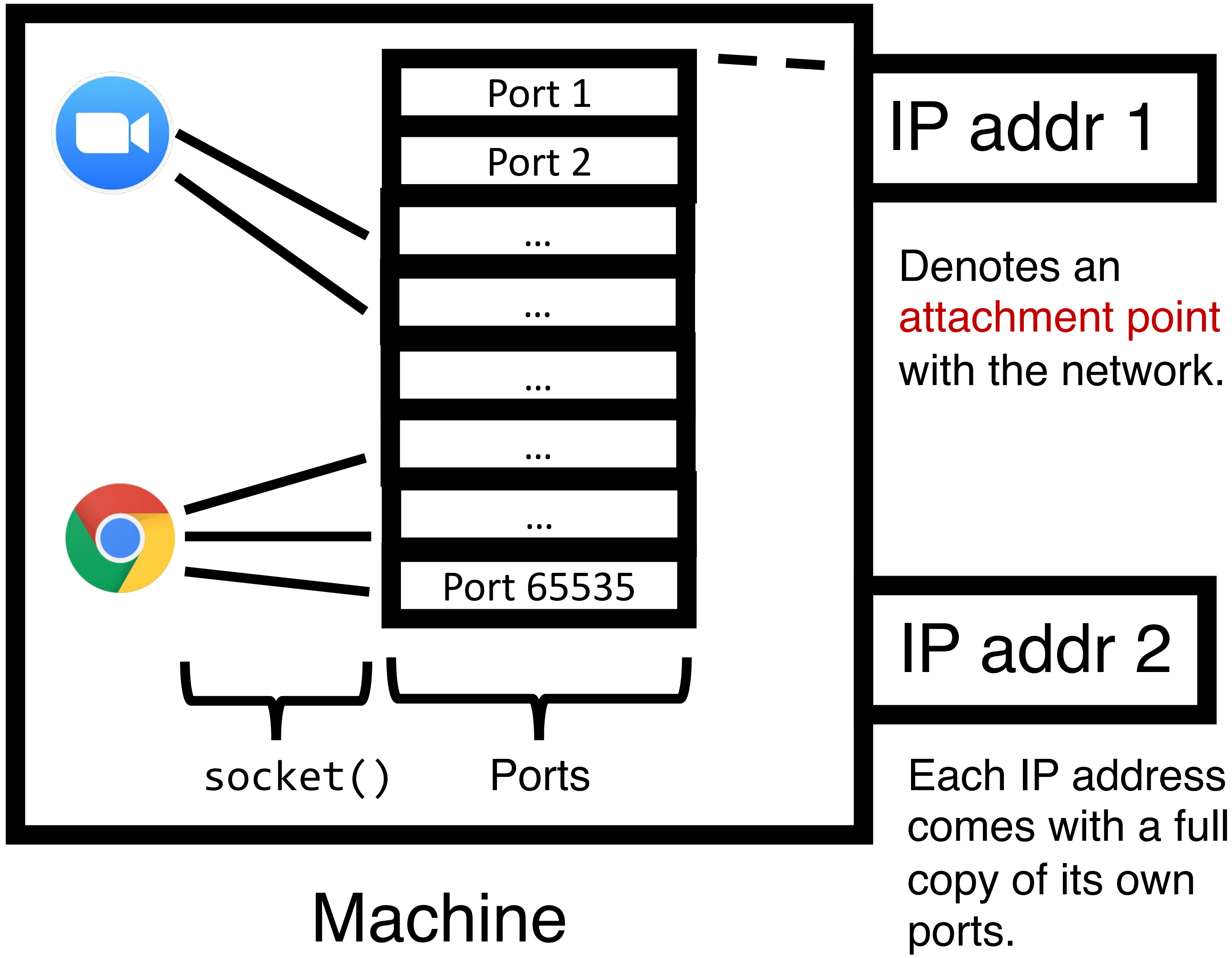

process
socket
IP_A + $port_A$
connect(
IP_B, $port_B$)
send()
process
socket
IP_B + $port_B$
bind(IPaddr$_B$, port$_B$)
listen()
accept()
recv()

Sample code

- Walk through
- What sockets exist on the machine?
    - `ss`

What does transport do?

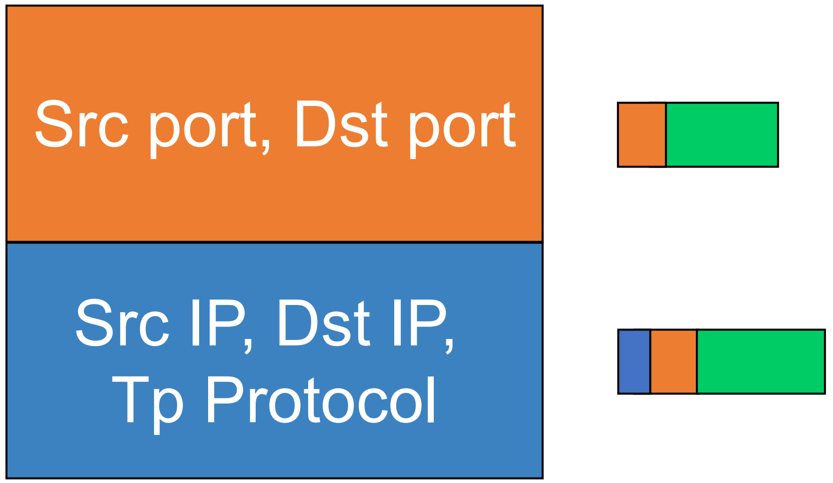
(3.1) App Context



Denotes an attachment point with the network.

Each IP address comes with a full copy of its own ports.

Connection lookup: The operating system does a lookup using these data to determine the right socket and app.



UDP or TCP listening:
(dst IP, dst port, TCP/UDP)

TCP established:
(dst IP, dst port, src IP, src port, TCP)

TCP sockets of different types

Listening (bound but unconnected)

```
# On server side
ls = socket(AF_INET, SOCK_STREAM)
ls.bind(serv_ip, serv_port)
ls.listen() # no accept() yet
```

(dst IP, dst port)

➔

Socket (ss)

Enables new connections to be demultiplexed correctly

Connected (Established)

```
# On server side
cs, addr = ls.accept()

# On client side
connect(serv_ip, serv_port)
```

`accept()` creates a new socket with the 4-tuple (established) mapping

(src IP, dst IP, src port, dst port)

➔

Socket (cs NOT ls)

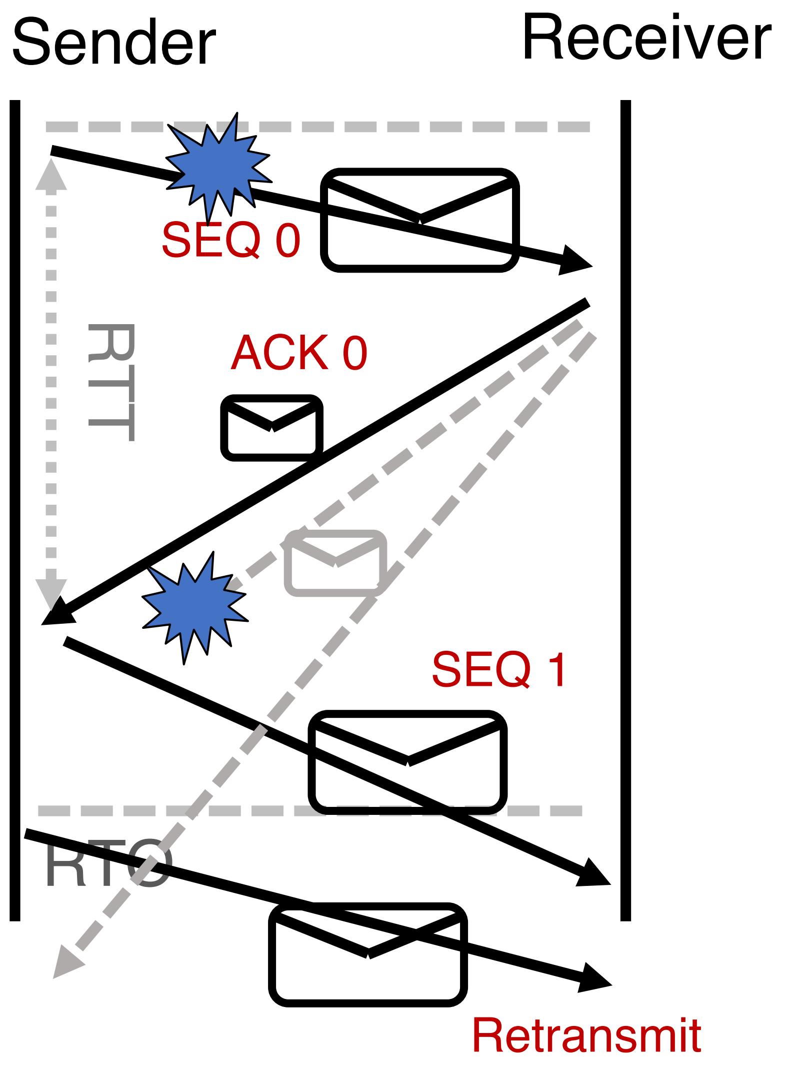
Enables established connections to be demultiplexed correctly

(3.2) Reliability: Stop and Wait. 3 Ideas

- ACKs: Sender sends a single packet, then waits for an ACK to know the packet was successfully received. Then the sender transmits the next packet.

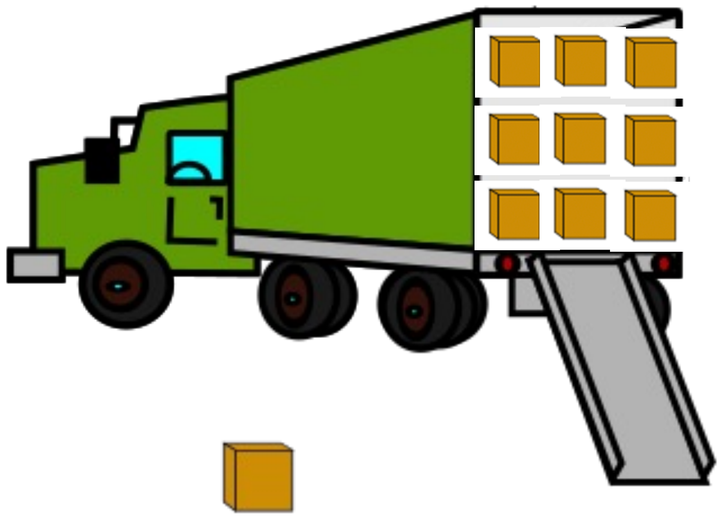
- RTO: If ACK is not received until a timeout, sender retransmits the packet

- Seq: Disambiguate duplicate vs. fresh packets using sequence numbers that change on "adjacent" packets



Stop and wait is reliable, but too slow.

Sending one packet per RTT makes the data transfer rate limited by the time between the endpoints, rather than the bandwidth.



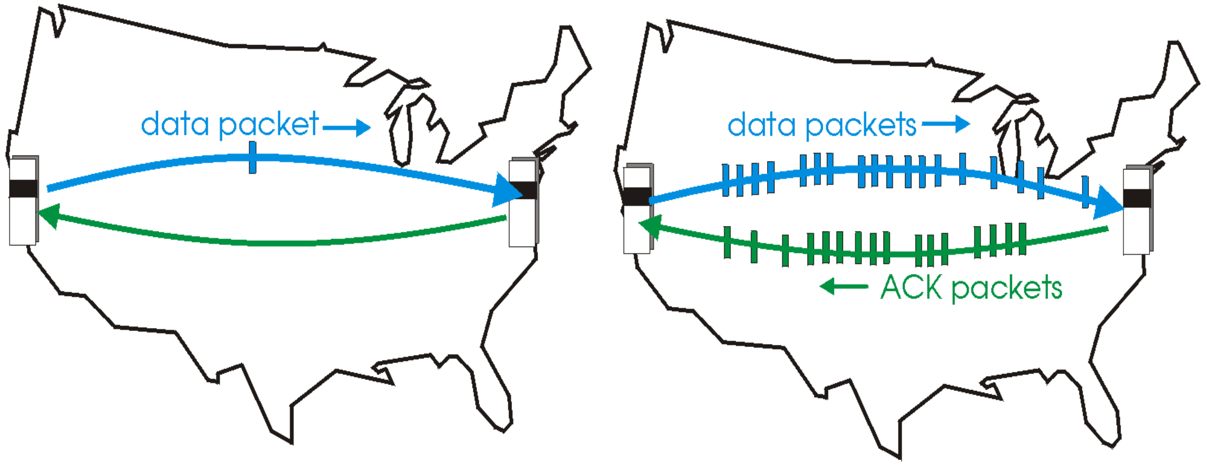
Ensure you got the (one) box safely; make N trips

Ensure you get N boxes safely; make just 1 trip!

Keep many packets in flight

Pipelined reliability

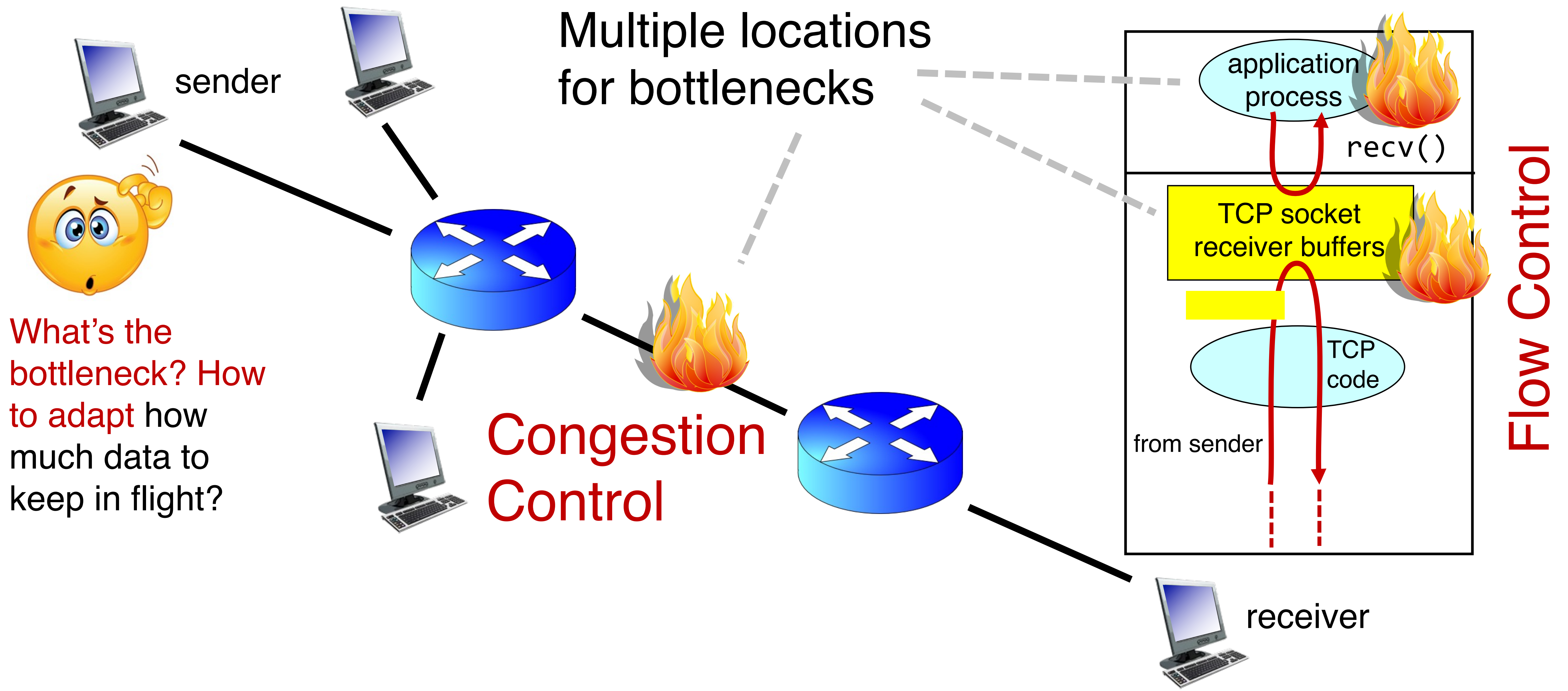
- Data in flight: data that has been sent, but sender hasn't yet received ACKs from the receiver
  - Note: can refer to packets in flight or bytes in flight
- New packets sent at the same time as older ones still in flight
- New packets sent at the same time as ACKs are returning
- More data moving in same time!
- Improves throughput
  - Rate of data transfer
- Window
  - How big should the window be?



(a) a stop-and-wait protocol in operation

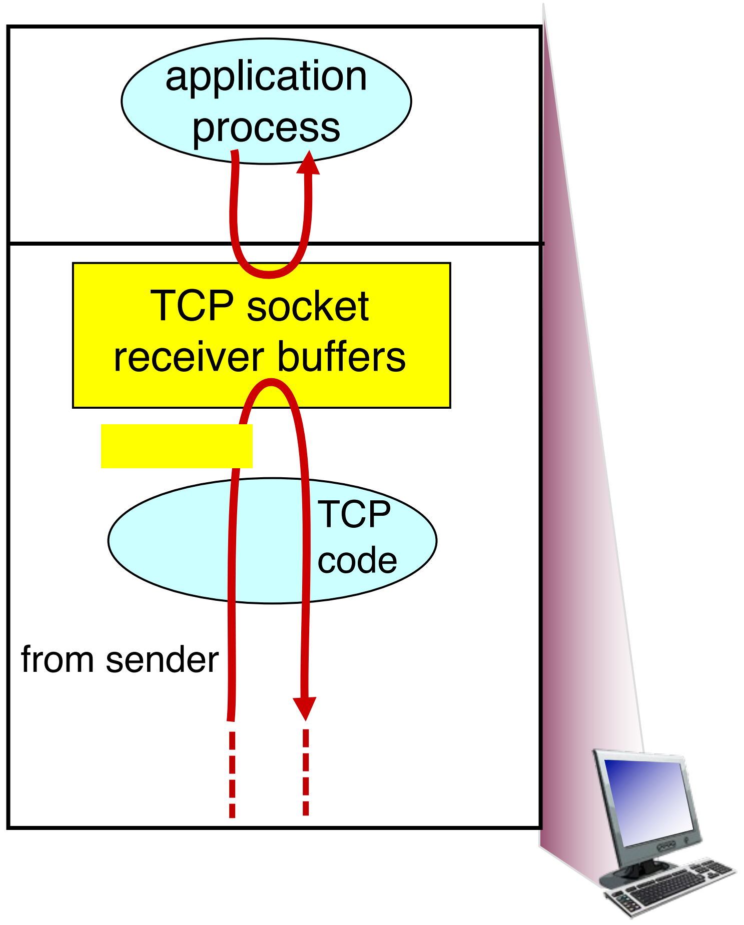
(b) a pipelined protocol in operation

We want to increase throughput, but …


sender
Multiple locations for bottlenecks
application process
recv()
TCP socket receiver buffers
TCP code
from sender
Flow Control
What's the bottleneck? How to adapt how much data to keep in flight?
Congestion Control
receiver

(3.3) Flow Control

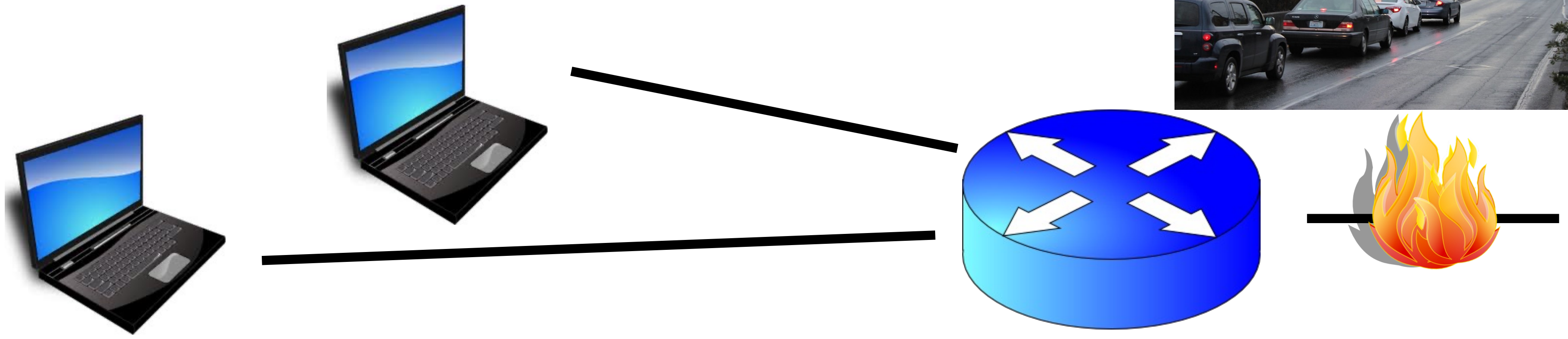
- Have a TCP sender only send as much as the free buffer space available at the receiver.
- *Amount of free buffer varies over time!*
- TCP implements flow control
- Receiver's ACK contains the amount of data the sender can transmit without running out the receiver's socket buffer
- This number is called the advertised window size
- Receiver buffer must be large enough



receiver network stack

(3.4) Congestion control

- How quickly should endpoints send data?



- Known as the congestion control problem
- Congestion control algorithms at source endpoints react to remote network congestion.
- Key question: How to vary the sending rate based on network signals?

A key consequence of the Internet architecture:

Place trust and intelligence in endpoints.

Congestion control is a distributed algorithm (running at endpoints) which attempts to achieve an efficient and fair distribution of bottleneck link resources.

Feedback Control

Finding the right congestion window

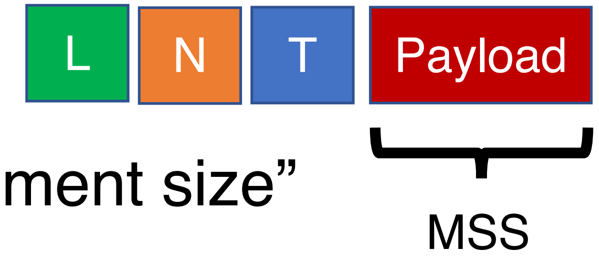
- There is an unknown bottleneck link rate that the sender must match

- 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

- Congestion window (`cwnd`): amount of data in flight

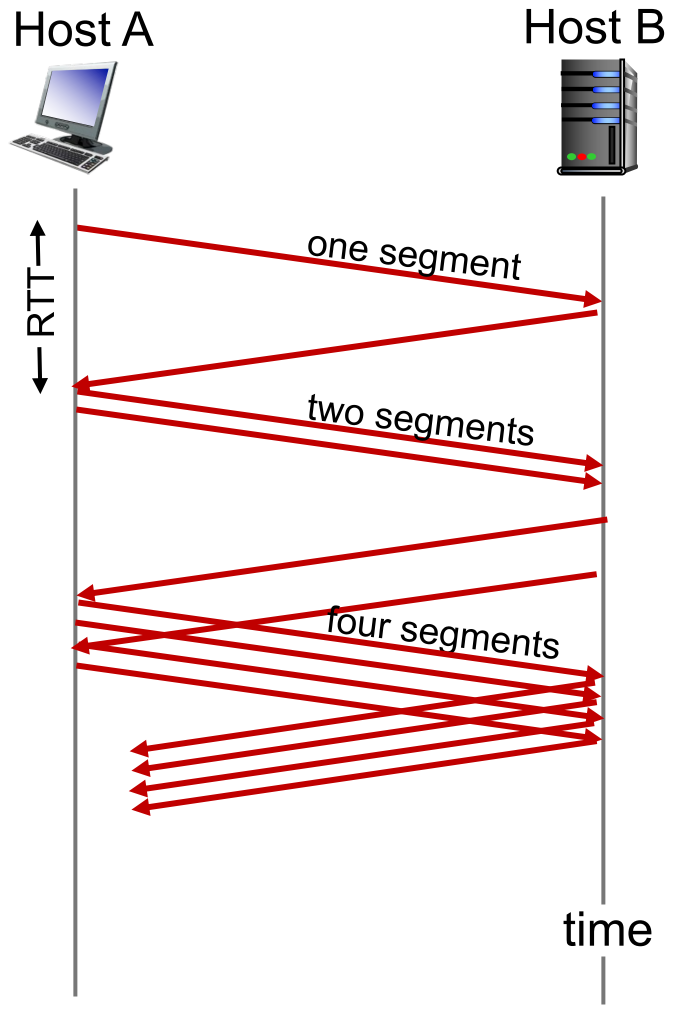
Quickly finding a rate: TCP slow start



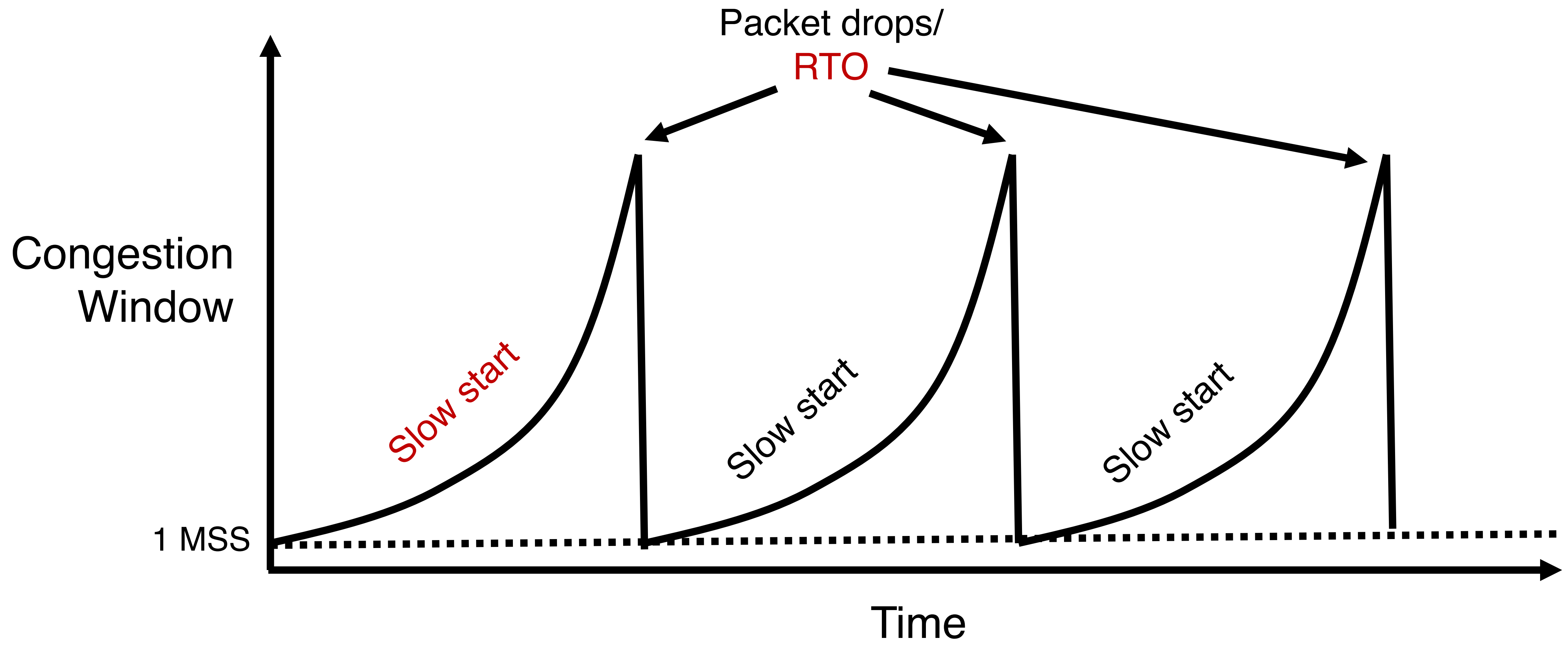
- Initially `cwnd` = 1 MSS
  - MSS is “maximum segment size”
- Upon receiving an ACK of each MSS, increase the `cwnd` by 1 MSS
- Effectively, double `cwnd` every RTT

▪ 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

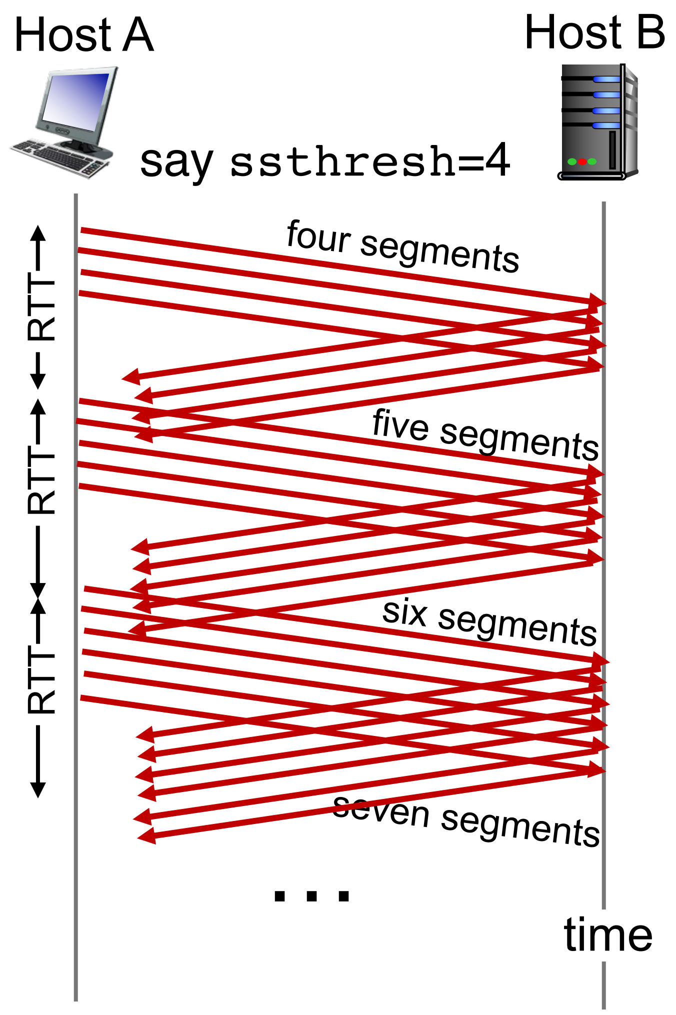
- Congestion window increases too rapidly
    - Example: suppose the "right" window size `cwnd` is 17
    - `cwnd` would go from 16 to 32 and then dropping down to 1
    - Result: massive packet drops

- 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 speed of sending data

- Instead, perform finer adjustments of `cwnd`: congestion avoidance

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



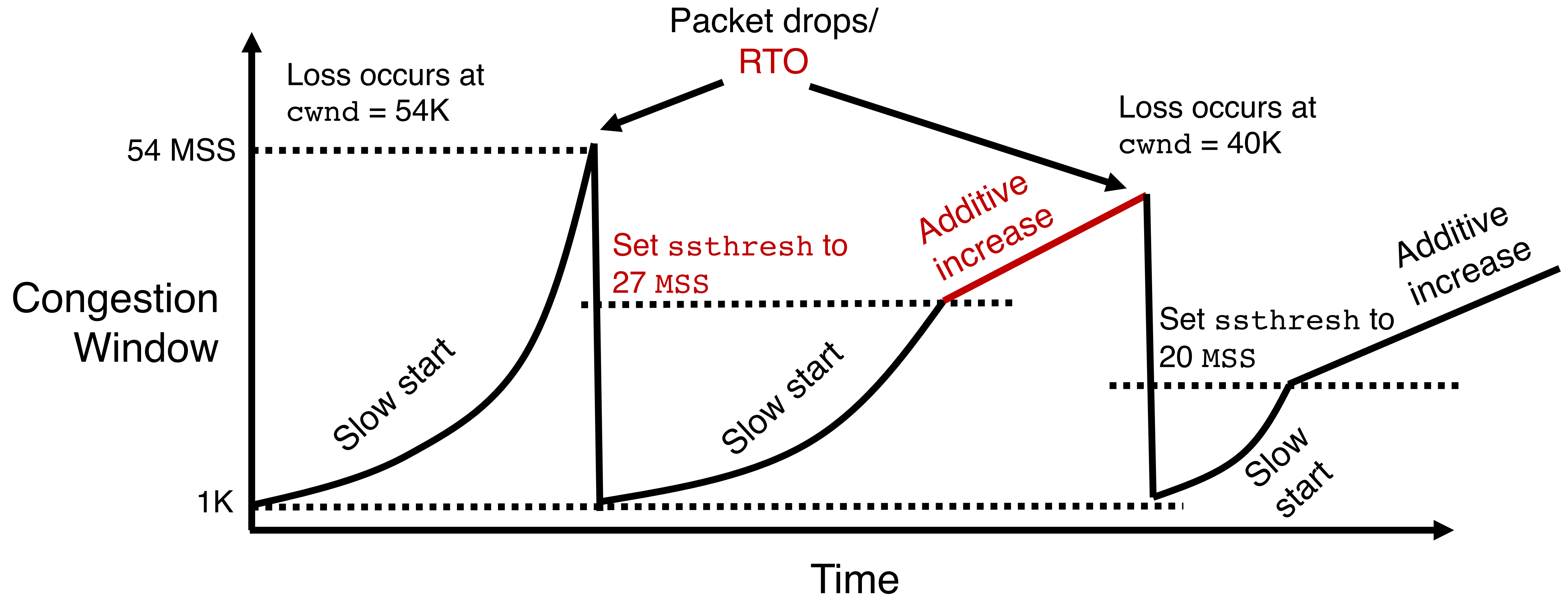
TCP New Reno: Additive increase

- 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`

AI is slow.
Persistent connections
Large window sizes
Different laws to evolve congestion window



Sample code & demo