UNIVERSITY OF COPENHAGEN

Computer Networking II

David Marchant

Based on slides compiled by Marcos Vaz Salles, with adaptions by Vivek Shah and Michael Kirkedel Thomsen

Reviewing the OSI model

Application Layer
Transport Layer
Network Layer
Link Layer
Physical Layer

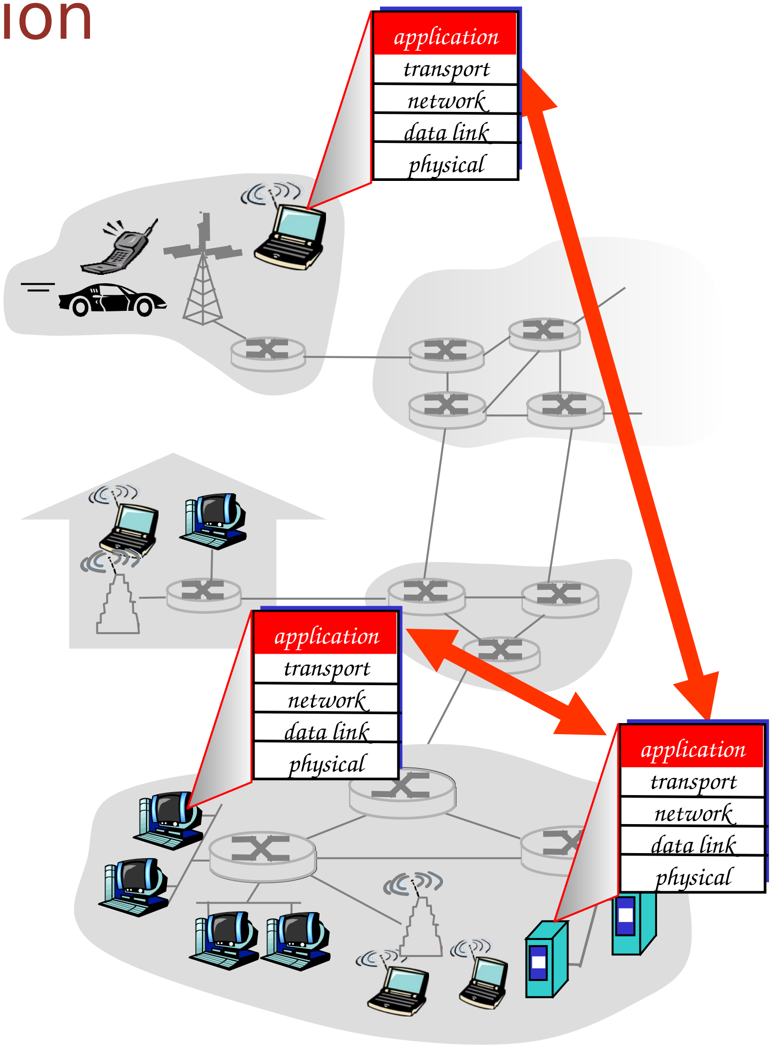
Creating a network application

write programs that

- run on (different) *end systems*
- communicate over network
- e.g., web server software communicates with browser software

No need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



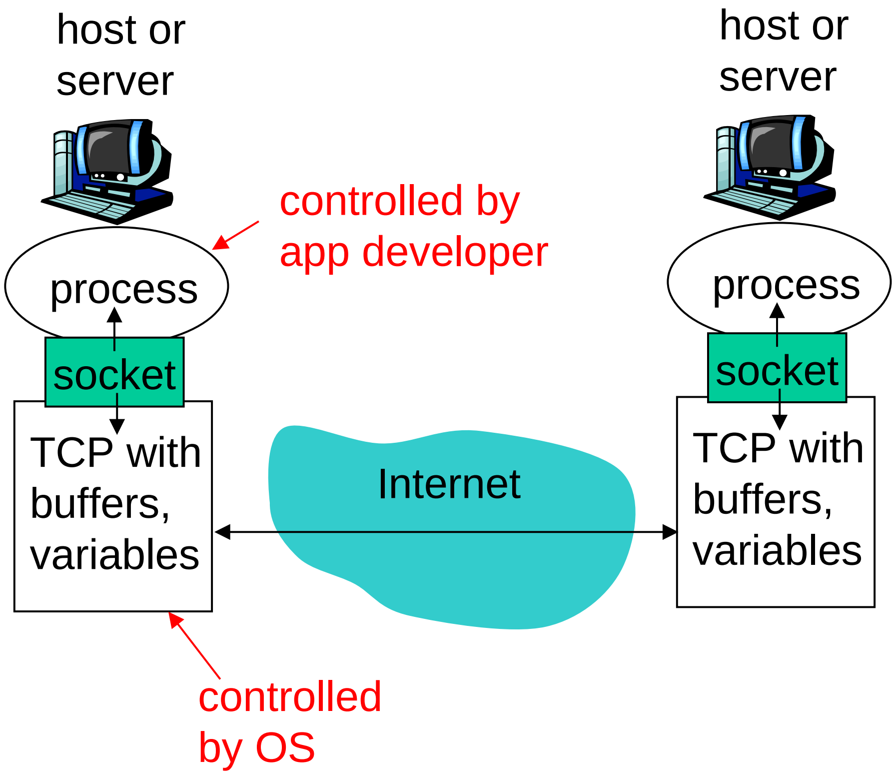
Source: Kurose & Ross

Sockets

process sends/receives messages to/from its socket

socket analogous to door

- sending process shoves message out door
- sending process relies on transport infrastructure on other side of door which brings message to socket at receiving process



API: (1) choice of transport protocol; (2) ability to fix a few parameters (more on this in next lecture!)

Source: Kurose & Ross (partial)

Internet transport protocols services

TCP service:

- *connection-oriented:* setup required between client and server processes
- *reliable transport:* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing, minimum throughput guarantees, security

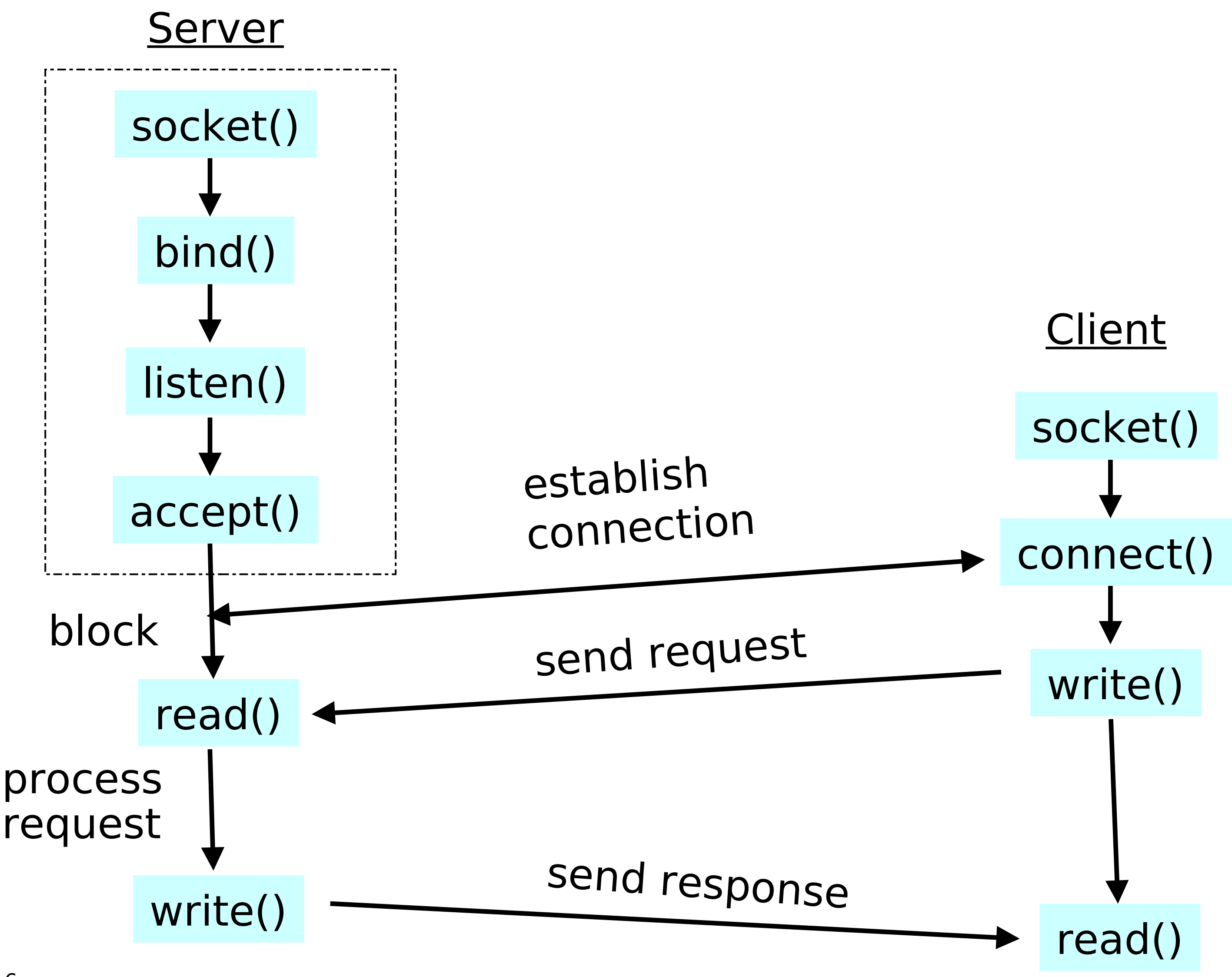
UDP service:

- unreliable data transfer between sending and receiving process
- does not provide: connection setup, reliability, flow control, congestion control, timing, throughput guarantee, or security

Q: why bother? Why is there a UDP?

Source: Kurose & Ross

Client-Server TCP Sockets



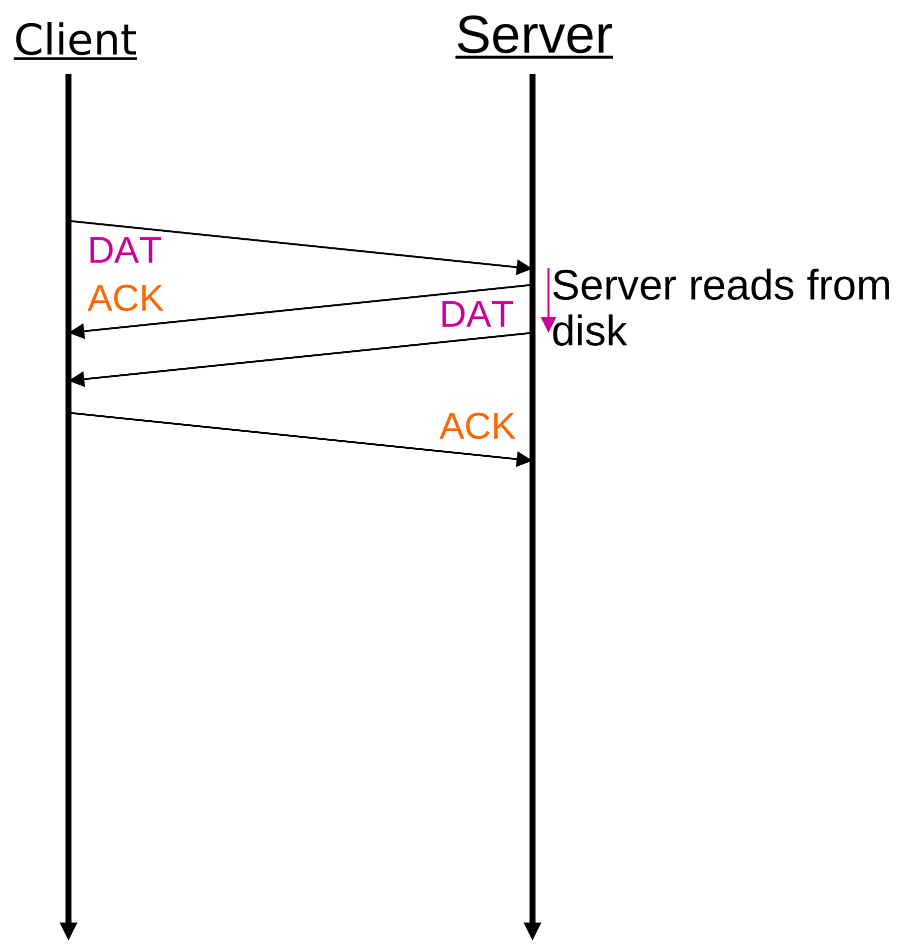
Source: Freedman (partial)

Single Transfer Example

Source: Freedman

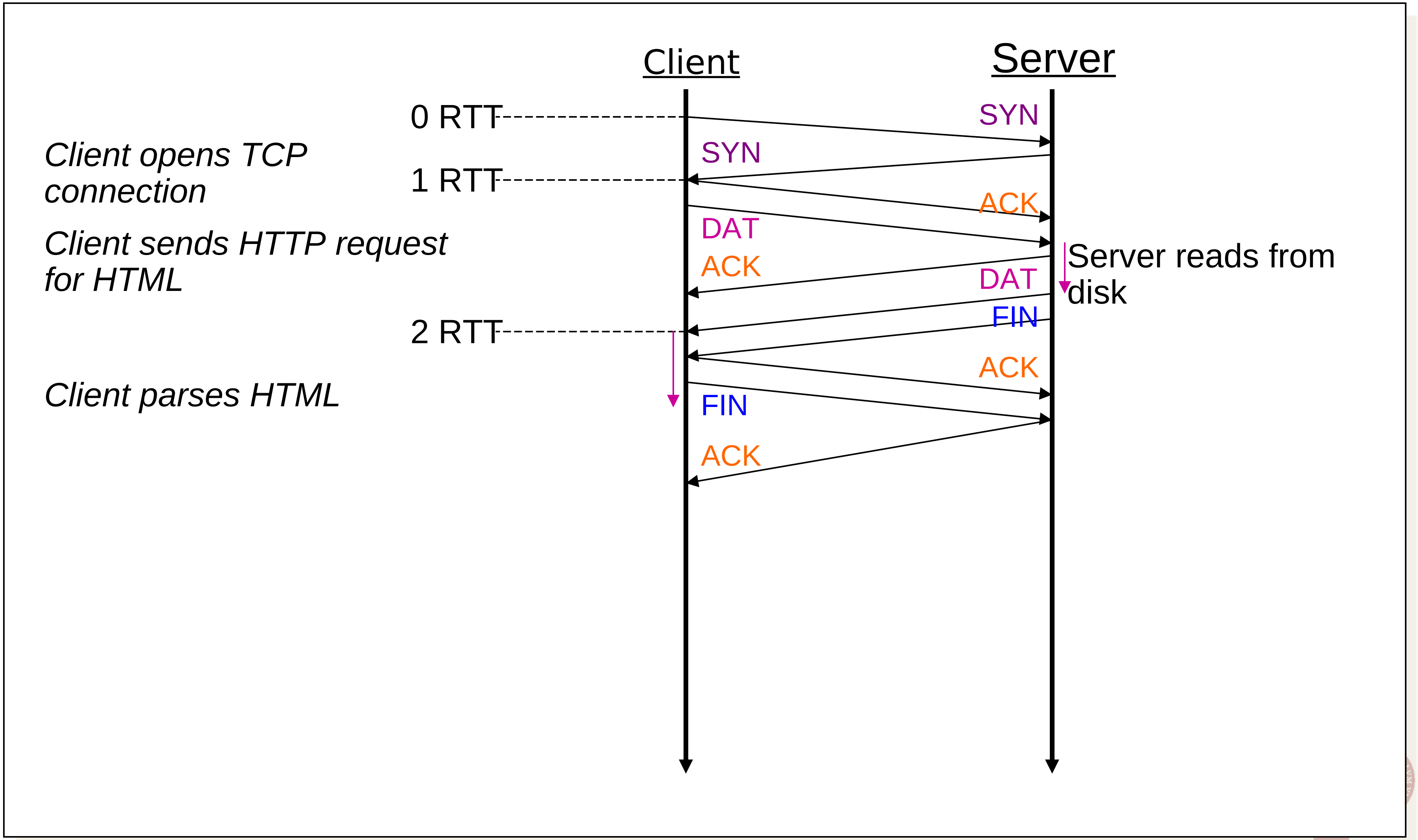
Client sends HTTP request for HTML

Client parses HTML



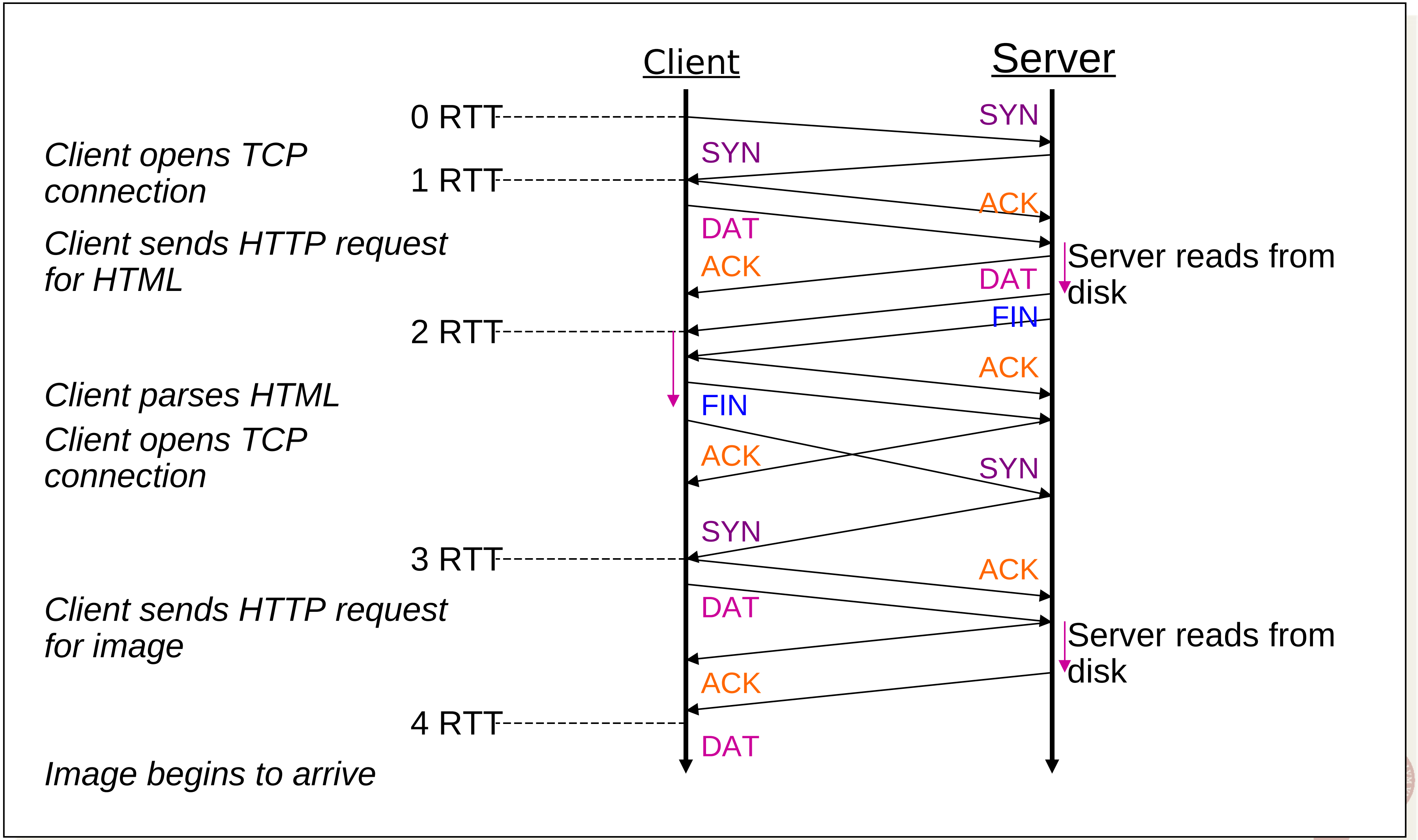
Single Transfer Example

Source: Freedman



Single Transfer Example

Source: Freedman



Problems with simple model

Multiple connection setups

- Three-way handshake each time (TCP "synchronizing" stream)

Lots of extra connections

- Increases server state/processing
- Server forced to keep connection state

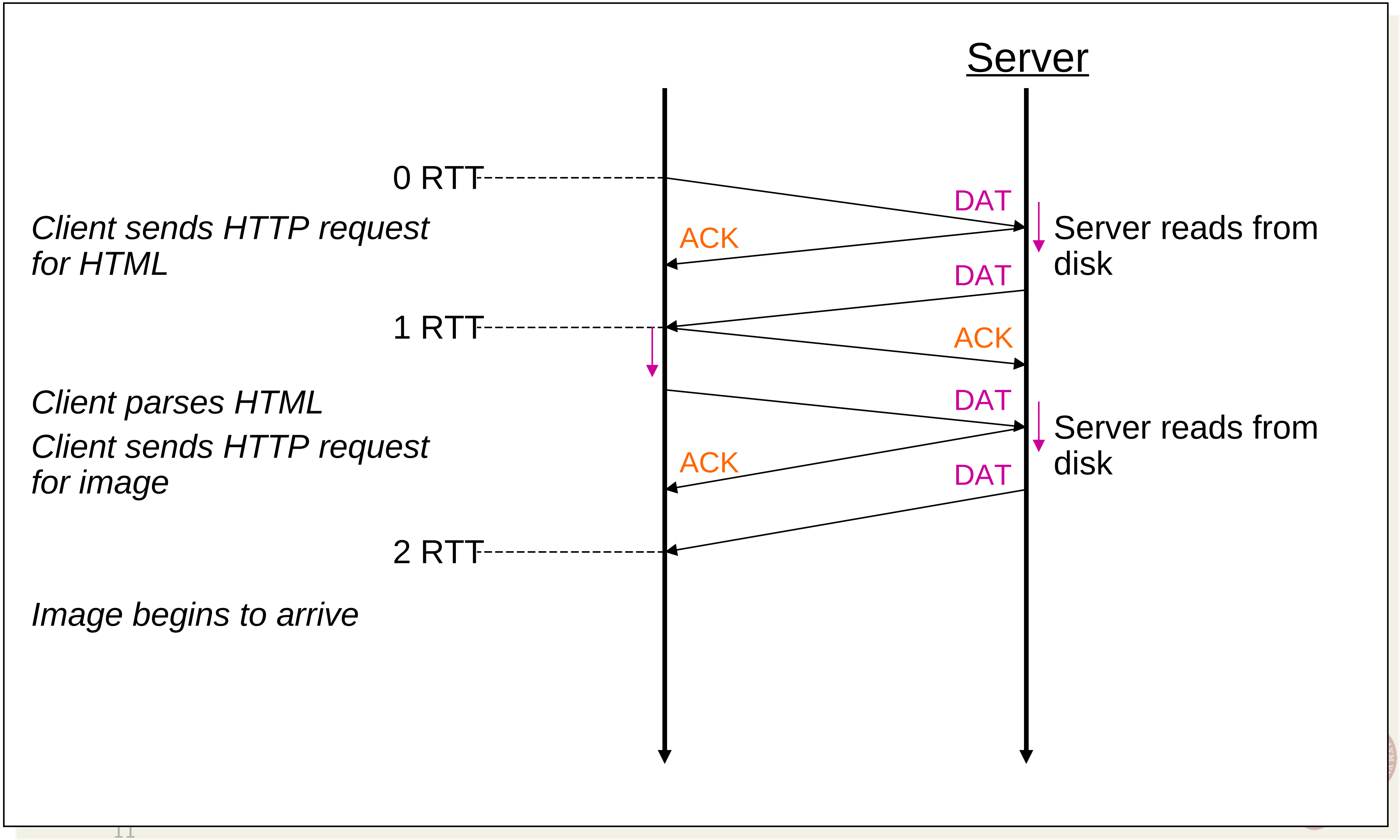
Later we will see also that

- Short transfers are hard on stream protocol (TCP)
  - How much data should it send at once?
  - Congestion avoidance: Takes a while to "ramp up" to high sending rate (TCP "slow start")
  - Loss recovery is poor when not "ramped up"

Source: Freedman (partial)

Persistent Connection Example

Source: Freedman



Persistent HTTP

Non-persistent HTTP issues:

- Requires 2 RTTs per object
- OS must allocate resources for each TCP connection
- But browsers often open parallel TCP connections to fetch referenced objects

Persistent HTTP:

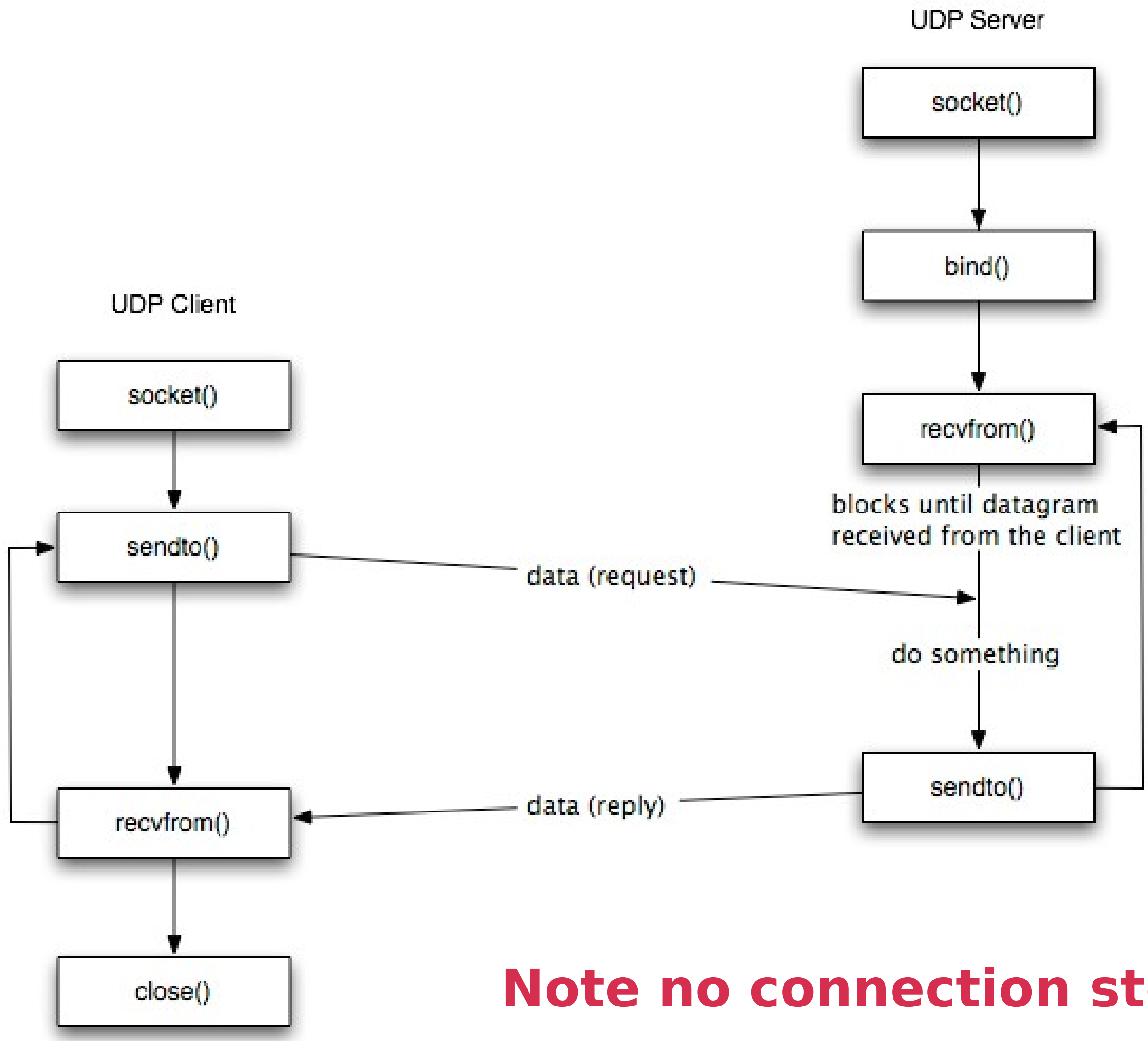
- Server leaves connection open after sending response
- Subsequent HTTP messages between same client/server are sent over connection

Source: Freedman

What about UDP?

Source: Freedman

Socket Programming Using UDP



Note no connection step

Source: Campbell

Hierarchical Names

Host name: www.cs.princeton.edu

- Domain: registrar for each top-level domain (e.g., .edu)
- Host name: local administrator assigns to each host

IP addresses: 128.112.7.156

- Prefixes: ICANN, regional Internet registries, and ISPs
- Hosts: static configuration, or dynamic using DHCP

(more on DHCP later in the course ☺)

Source: Freedman (partial)

Separating Names and IP Addresses

Names are easier (for us!) to remember

- www.cnn.com vs. 64.236.16.20

IP addresses can change underneath

- Move www.cnn.com to 173.15.201.39
- E.g., renumbering when changing providers

Name could map to multiple IP addresses

- www.cnn.com to multiple replicas of the Web site

Map to different addresses in different places

- Address of a nearby copy of the Web site
- E.g., to reduce latency, or return different content

Multiple names for the same address

- E.g., aliases like ee.mit.edu and cs.mit.edu

Source: Freedman

Outline: Domain Name System

Computer science concepts underlying DNS

- Indirection: names in place of addresses
- Hierarchy: in names, addresses, and servers
- Caching: of mappings from names to/from addresses

DNS software components

- DNS resolvers
- DNS servers

DNS queries

- Iterative queries
- Recursive queries

DNS caching based on time-to-live (TTL)

Source: Freedman

Strawman Solution: Central Server

All you need is to map names!

Central server

- One place where all mappings are stored
- All queries go to the central server

- Is this a good solution?
- What would be the potential drawbacks?

Source: Freedman (partial)

Domain Name System (DNS)

Properties of DNS

- Hierarchical name space divided into zones
- Distributed over a collection of DNS servers

Hierarchy of DNS servers

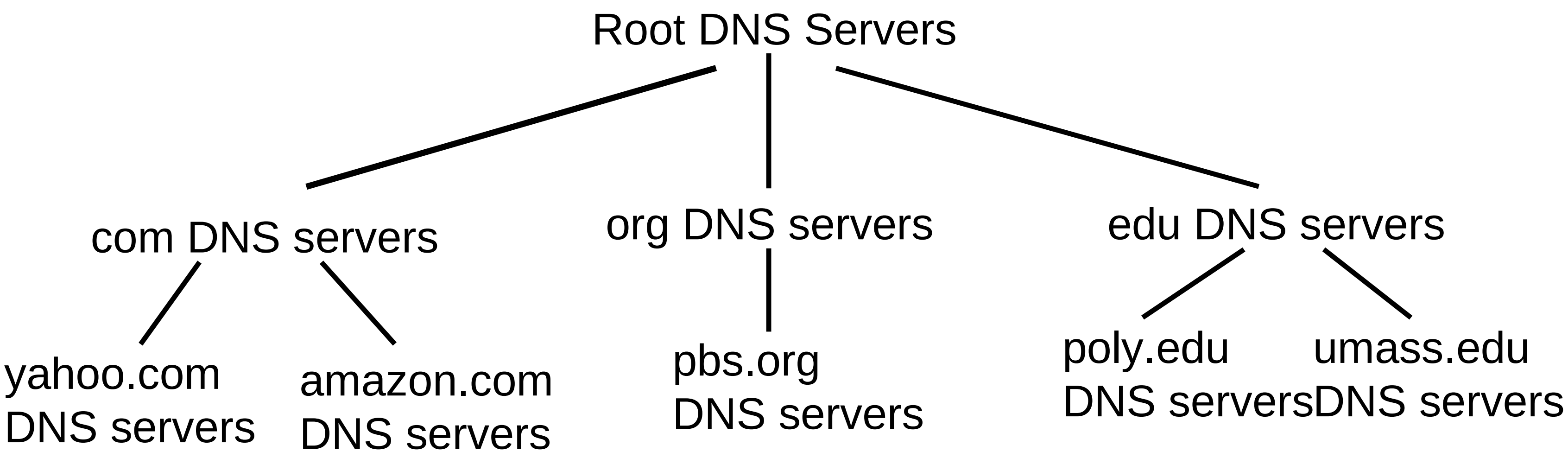
- Root servers
- Top-level domain (TLD) servers
- Authoritative DNS servers

Performing the translations

- Local DNS servers
- Resolver software

Source: Freedman

Distributed, Hierarchical Database



client wants IP for www.amazon.com; 1st approx:

client queries a root server to find com DNS server

client queries com DNS server to get amazon.com DNS server

client queries amazon.com DNS server to get IP address for www.amazon.com

Source: Kurose & Ross

DNS Name Resolution Example

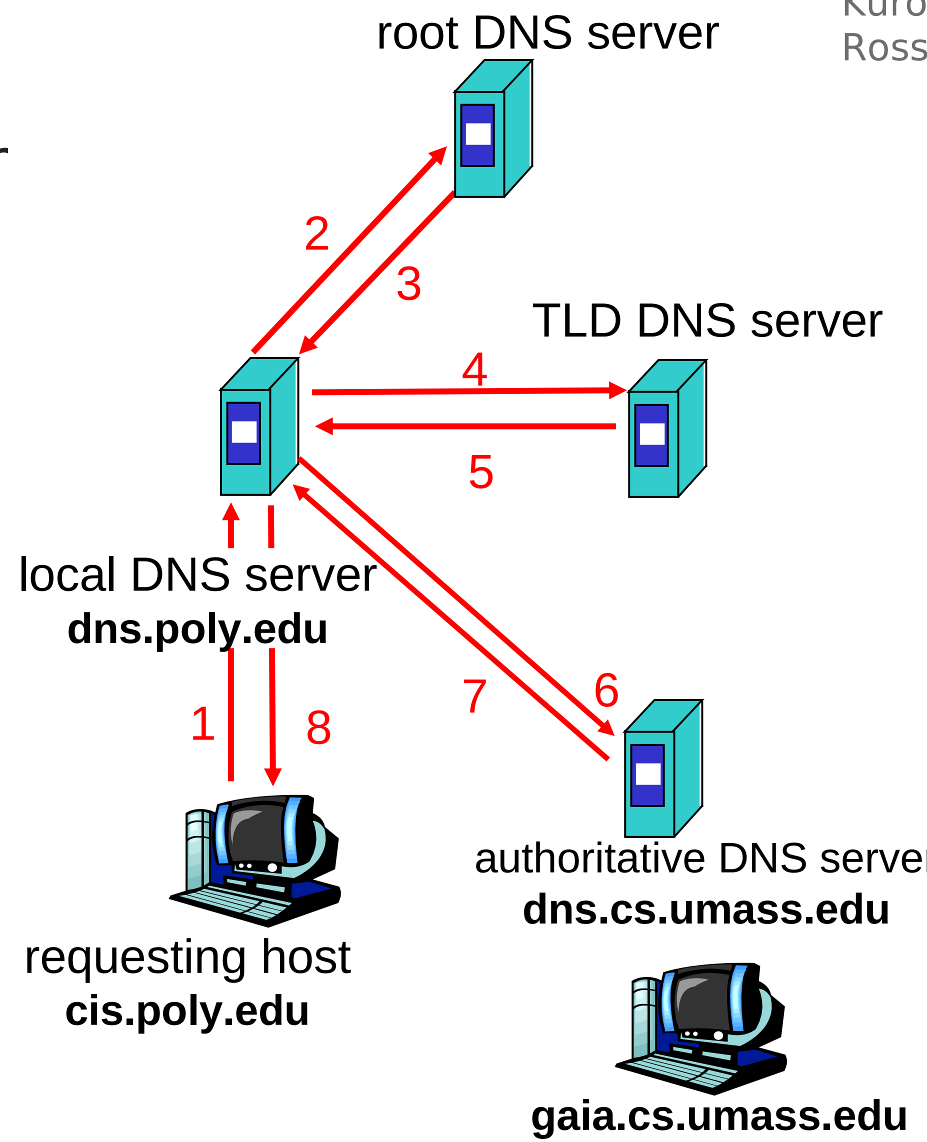
Source: Kurose & Ross

host at cis.poly.edu wants IP address for gaia.cs.umass.edu

Iterated query

contacted server replies with name of server to contact

"I don't know this name, but ask this server"

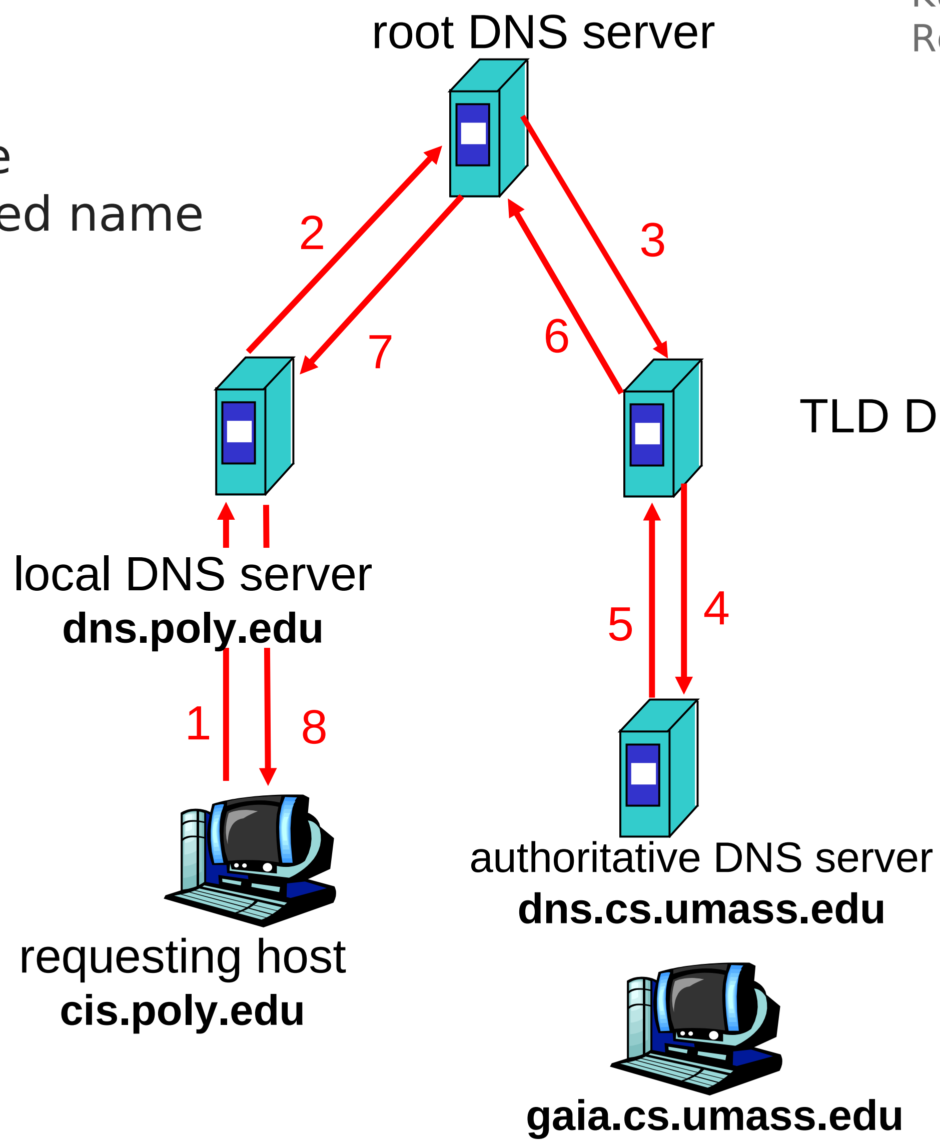


DNS Name Resolution Example

Source: Kurose & Ross

Recursive query

- puts burden of name resolution on contacted name server
- heavy load?



DNS Caching

Performing all these queries take time

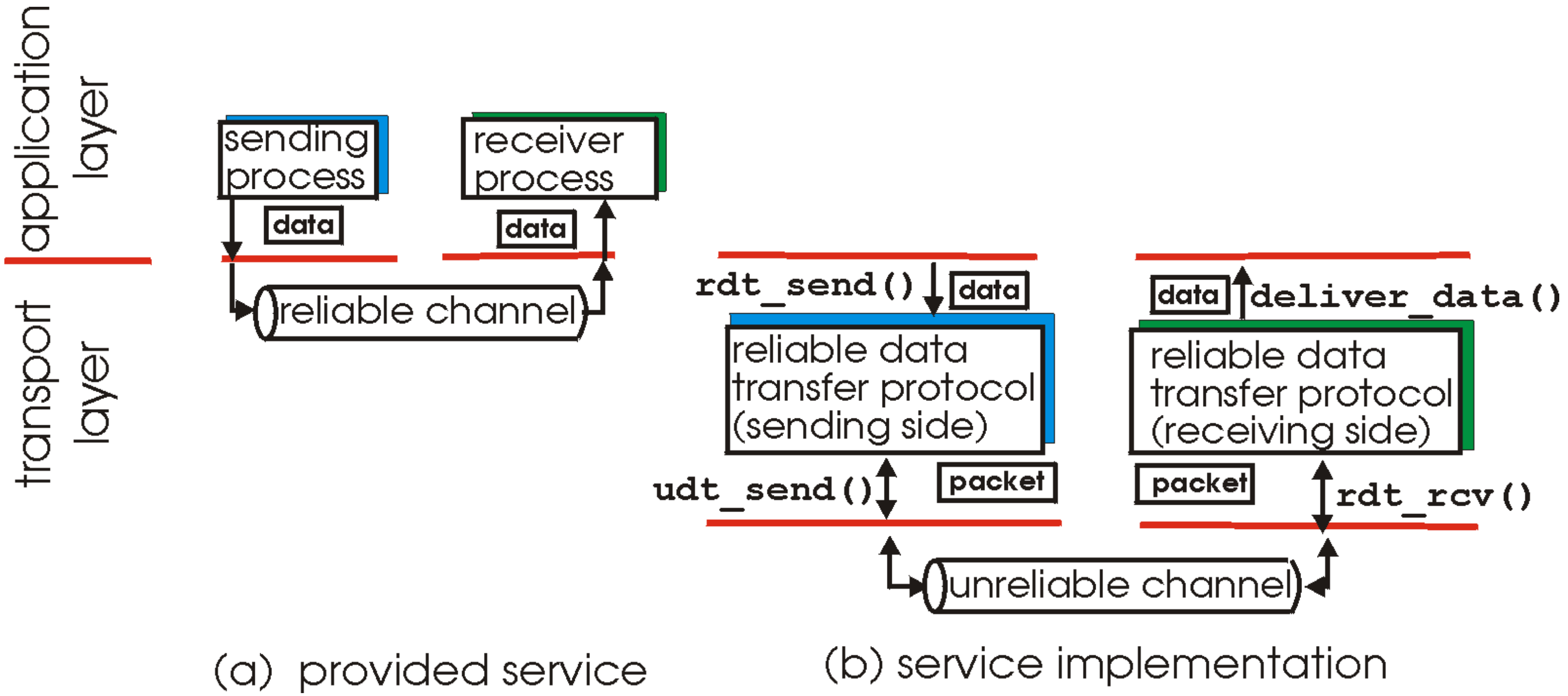
- And all this before the actual communication takes place
- E.g., 1-second latency before starting Web download

Caching can substantially reduce overhead

- The top-level servers very rarely change
- Popular sites (e.g., www.cnn.com) visited often
- Local DNS server often has the information cached

Source: Freedman (partial)

Reliable Data Transfer



Source: Kurose & Ross

- What can go wrong on the unreliable channel?
- How can you deal with it?
  - Suppose you want to transfer TCP segments, reliably and in order! ☺

Challenges of Reliable Data Transfer

- Over a perfectly reliable channel: Done
- Over a channel with bit errors
  - Receiver detects errors and requests re-transmission
- Over a lossy channel with bit errors
  - Some data missing, others corrupted
  - Receiver cannot easily detect loss
- Over a channel that may reorder packets
  - Receiver cannot easily distinguish loss vs. out-of-order

Source: Freedman

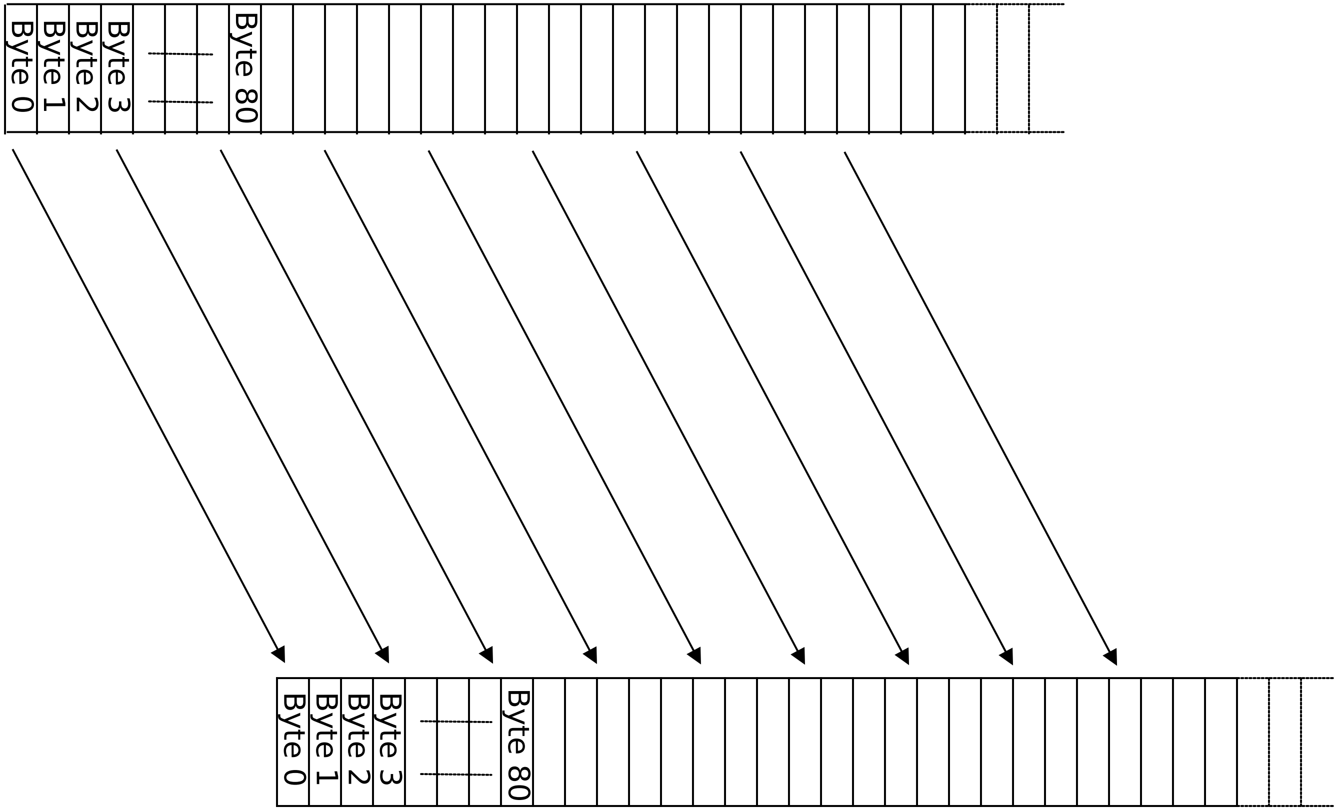
TCP Support for Reliable Delivery

- **Detect bit errors:** checksum
  - Used to detect corrupted data at the receiver
  - …leading the receiver to drop the packet
- **Detect missing data:** sequence number
  - Used to detect a gap in the stream of bytes
  - … and for putting the data back in order
- **Recover from lost data:** retransmission
  - Sender re-transmits lost or corrupted data
  - Two main ways to detect lost packets

Source: Freedman

TCP “Stream of Bytes” Service

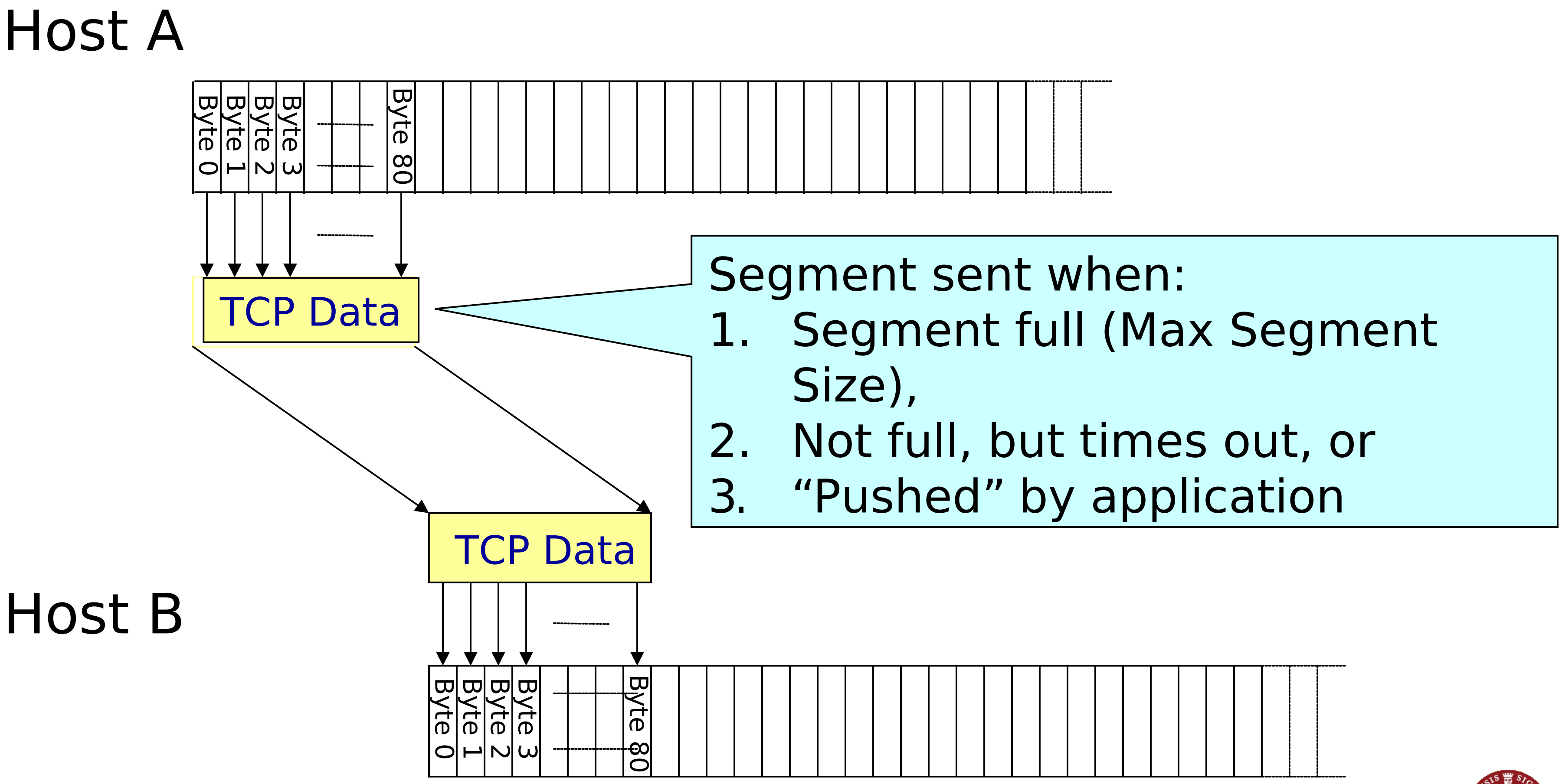
Host A



Host B

Source: Freedman

… Emulated Using TCP “Segments”



Source: Freedman

TCP Segment

IP packet

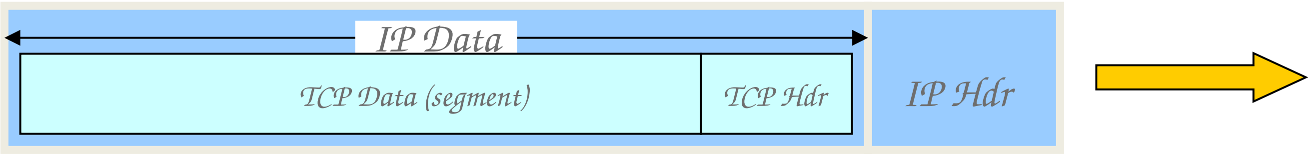
- No bigger than Maximum Transmission Unit (MTU)
- E.g., up to 1500 bytes on an Ethernet link

TCP packet

- IP packet with a TCP header and data inside
- TCP header is typically 20 bytes long

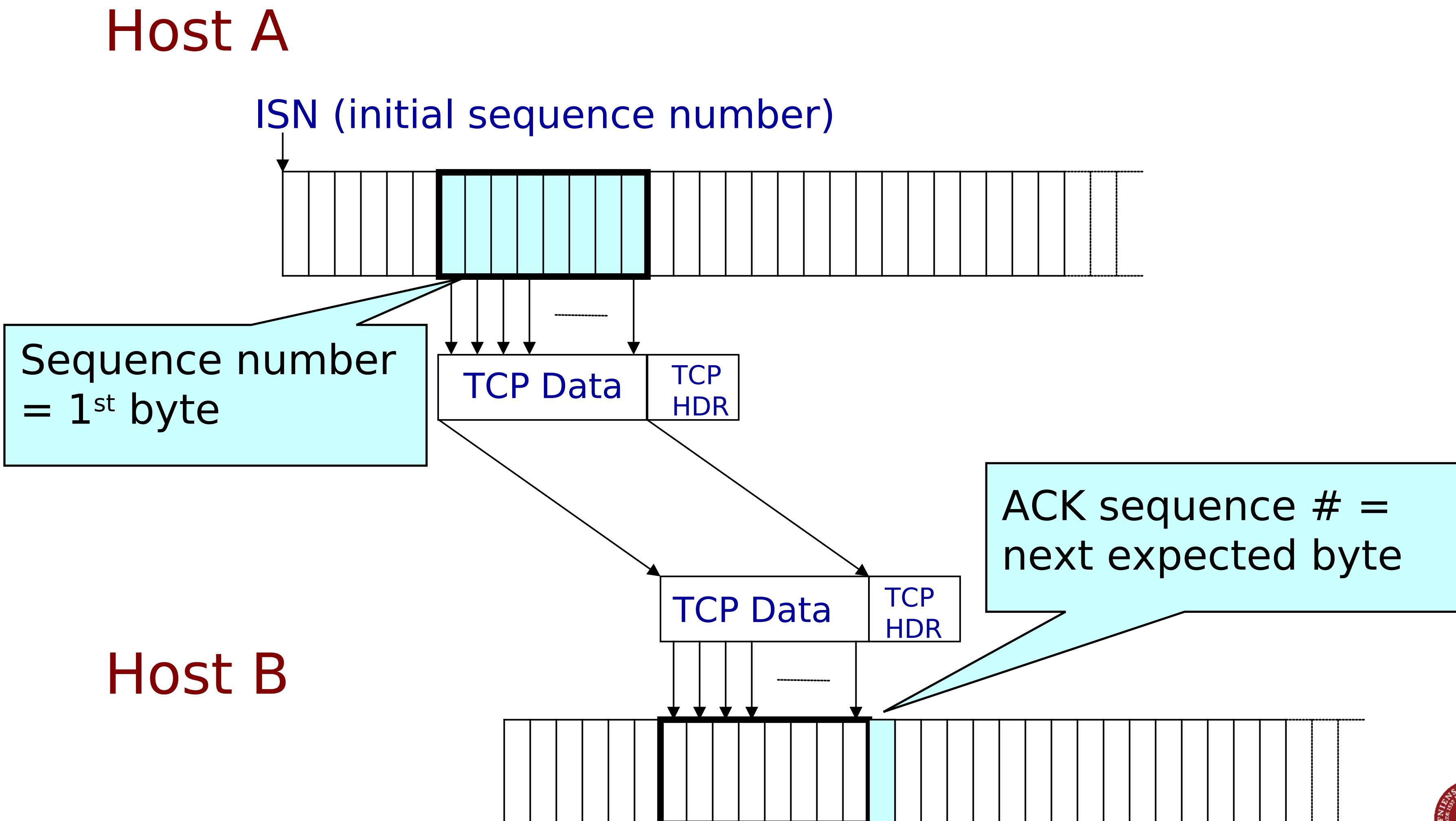
TCP segment

- No more than Maximum Segment Size (MSS) bytes
- E.g., up to 1460 consecutive bytes from the stream



Source: Freedman

TCP Acknowledgements

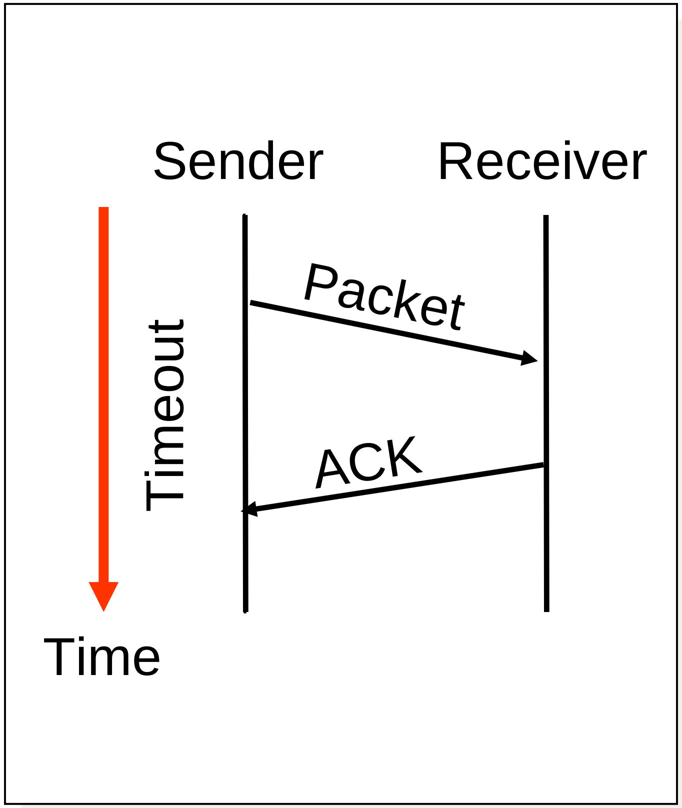


Source: Freedman

Automatic Repeat Request (ARQ)

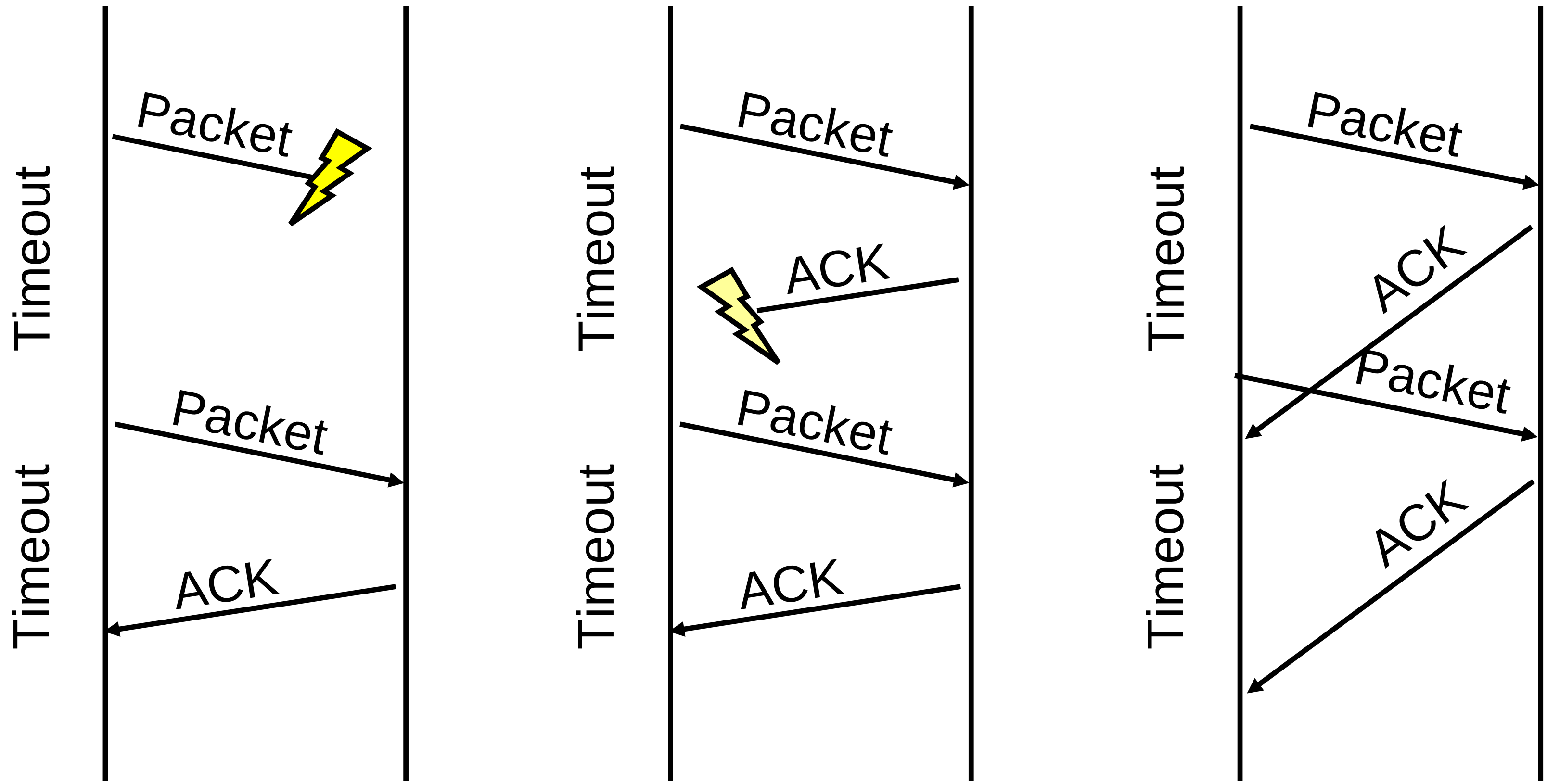
- Receiver sends ACK when it receives packet
- Sender waits for ACK.
- If ACK not received within some timeout period, resend packet

- “stop and wait”
  - One packet at a time…



Source: Freedman (partial)

Reasons for Retransmission



Packet lost

ACK lost
DUPLICATE PACKET

Early timeout
DUPLICATE PACKETS

Source: Freedman

TCP Fast Retrasmit

- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

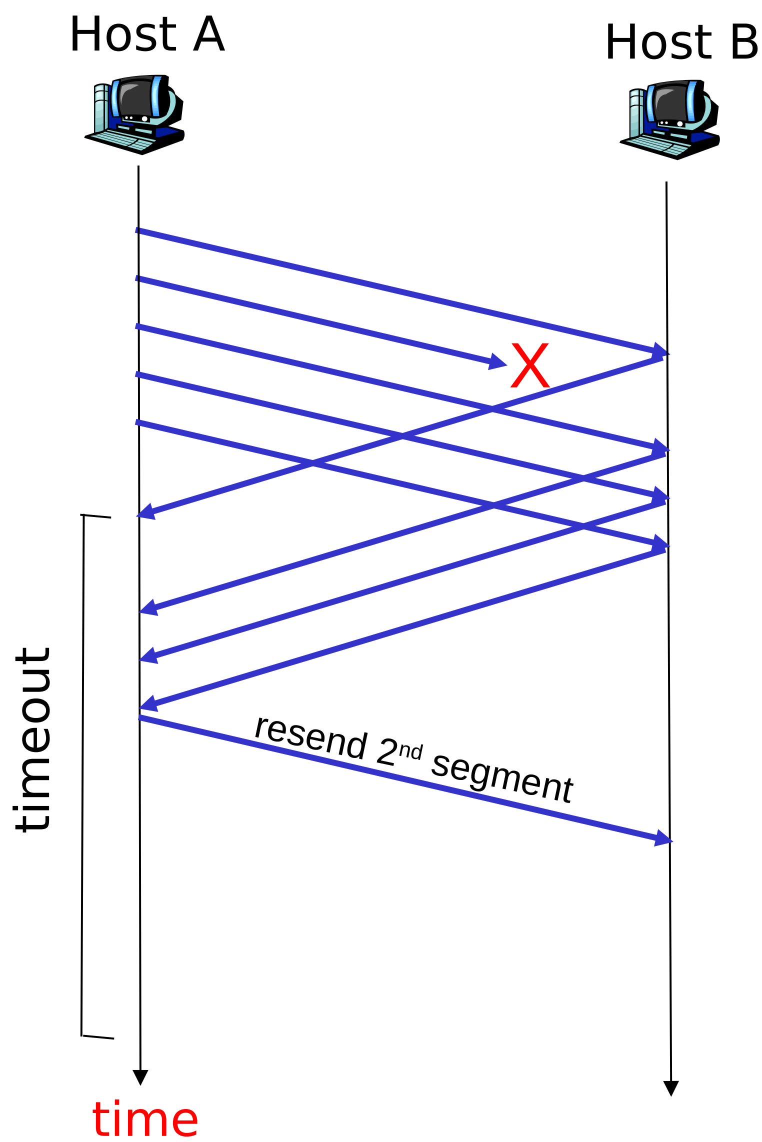
- if sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

Source: Kurose & Ross

TCP Fast Retransmit

Resending a segment after triple duplicate ACK

Triple duplicate ACK works as a logical NACK



Source: Kurose & Ross (partial)

Effectiveness of Fast Retransmit

When does Fast Retransmit work best?

- Long transfers: High likelihood of many pkts in flight
- Large window: High likelihood of many packets in flight
- Low loss burstiness: Higher likelihood that later pkts arrive

Implications for Web traffic

- Most Web objects are short (e.g., 10 packets)
- So, often aren't many packets in flight
- … making fast retransmit less likely to "kick in"
- … another reason for persistent connections!

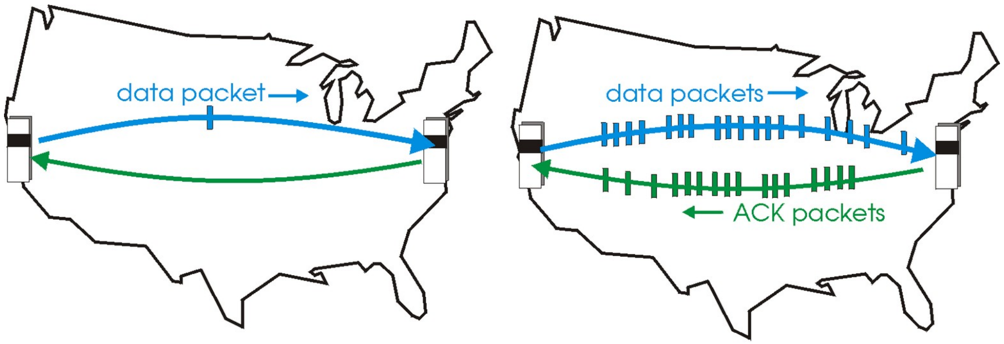
Source: Freedman

Increasing TCP throughput

- **Problem:** Stop-and-wait + timeouts are inefficient
  - Only one TCP segment "in flight" at time ✔
- **Solution:** Send multiple packets at once

Problem: How many w/o overwhelming receiver?

Solution: Determine "window size" dynamically



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

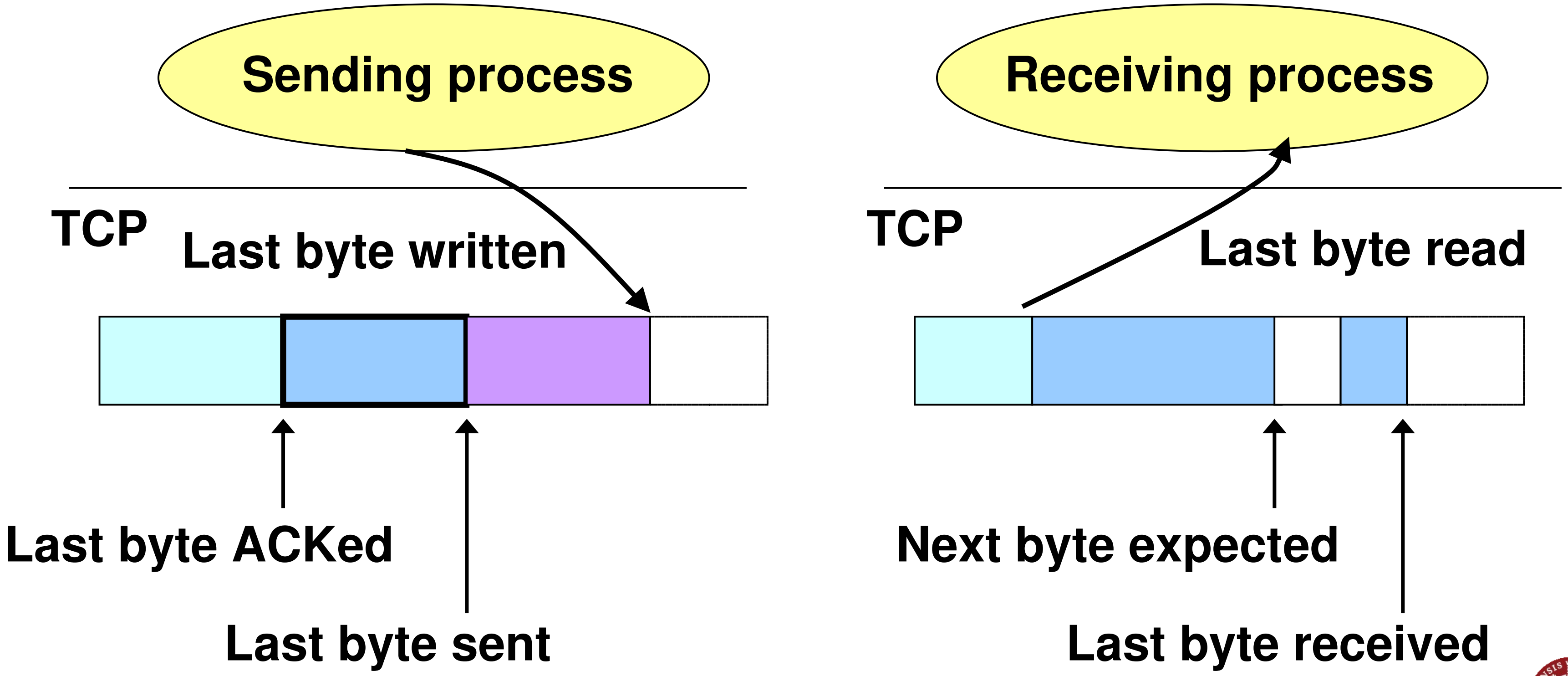
Image Source: Kurose & Ross

Source: Freedman (partial)

Flow Control: Sliding Window

Allow a larger amount of data “in flight”

- Sender can get ahead of receiver, though not *too far*



Source: Freedman (partial)

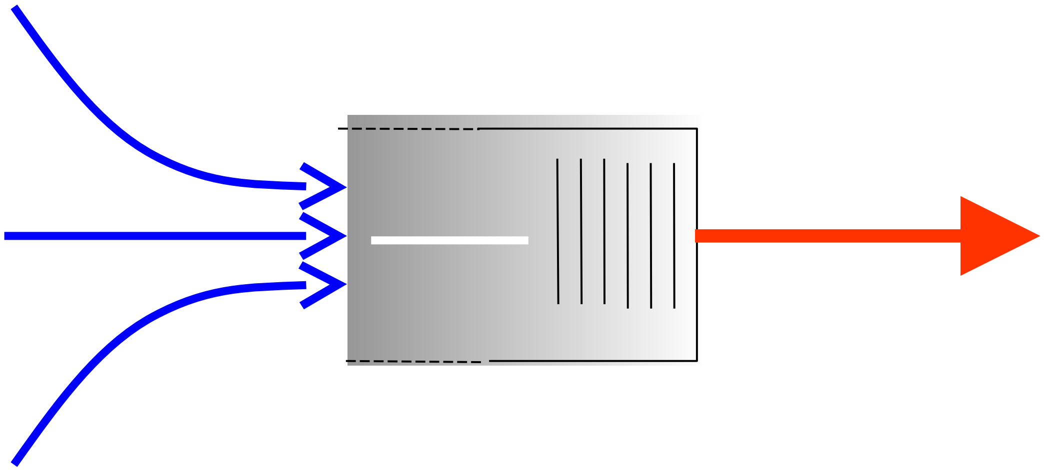
Congestion is Unavoidable

Two packets arrive at same time

- Router can only transmit one: must buffer or drop other

If many packets arrive in short period of time

- Router cannot keep up with the arriving traffic
- Buffer may eventually overflow



Source: Freedman

The Problem of Congestion

What is congestion?

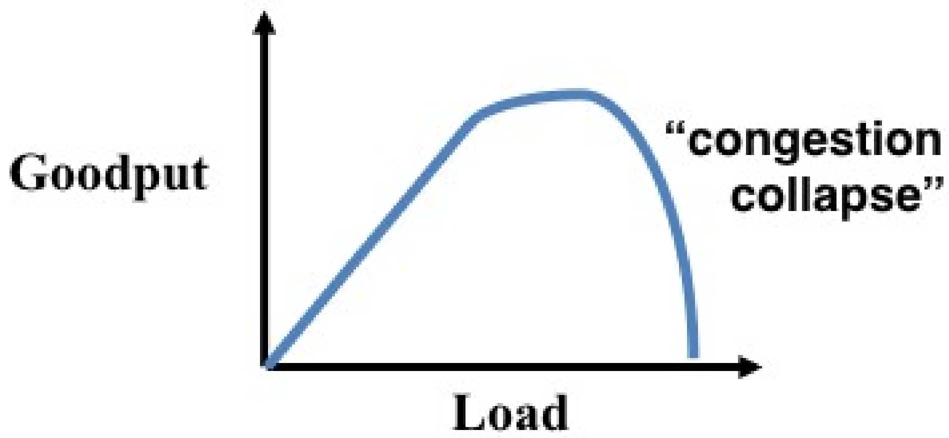
- Load is higher than capacity

What do IP routers do?

- Drop the excess packets

Why is this bad?

- Wasted bandwidth for re-transmissions



Increase in load that results in a *decrease* in useful work done.

Source: Freedman

Ways to Deal With Congestion

Ignore the problem

- Many dropped (and retransmitted) packets
- Can cause congestion collapse

Reservations, like in circuit switching

- Pre-arrange bandwidth allocations
- Requires negotiation before sending packets

Pricing

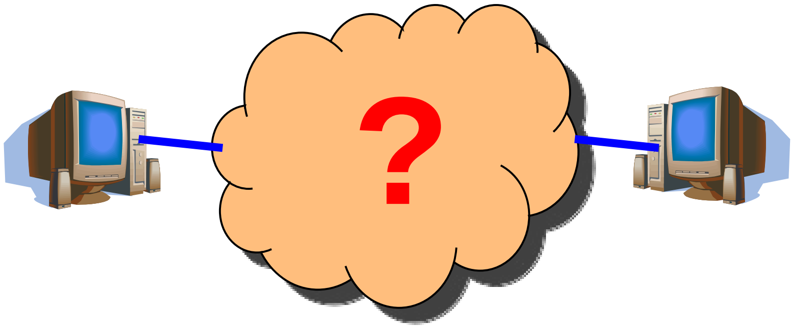
- Don't drop packets for the high-bidders
- Requires a payment model, and low-bidders still dropped

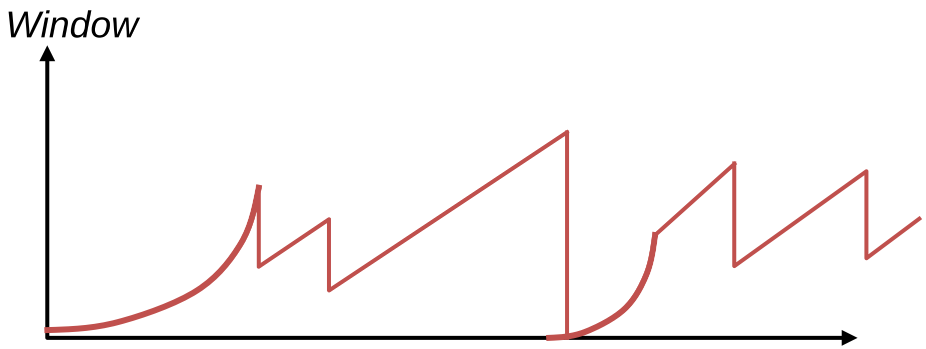
Dynamic adjustment (TCP)

- Every sender infers the level of congestion
- Each adapts its sending rate "for the greater good"

Source: Freedman

Inferring From Implicit Feedback





- What does the end host see?
- What can the end host change?
- What if conditions change?

- TCP keeps congestion window, as in the graph
- Can you explain behavior? Why are there increases and drops?
- Why is there a "sawtooth"?

Source: Freedman (partial)

TCP Congestion Window

Each TCP sender maintains a congestion window

- Max number of bytes to have in transit (not yet ACK'd)

Adapting the congestion window

- Decrease upon losing a packet: backing off
- Increase upon success: optimistically exploring
- Always struggling to find right transfer rate

Tradeoff

- Pro: avoids needing explicit network feedback
- Con: continually under- and over-shoots "right" rate

Source: Freedman

Additive Increase, Multiplicative Decrease (AIMD)

How much to adapt?

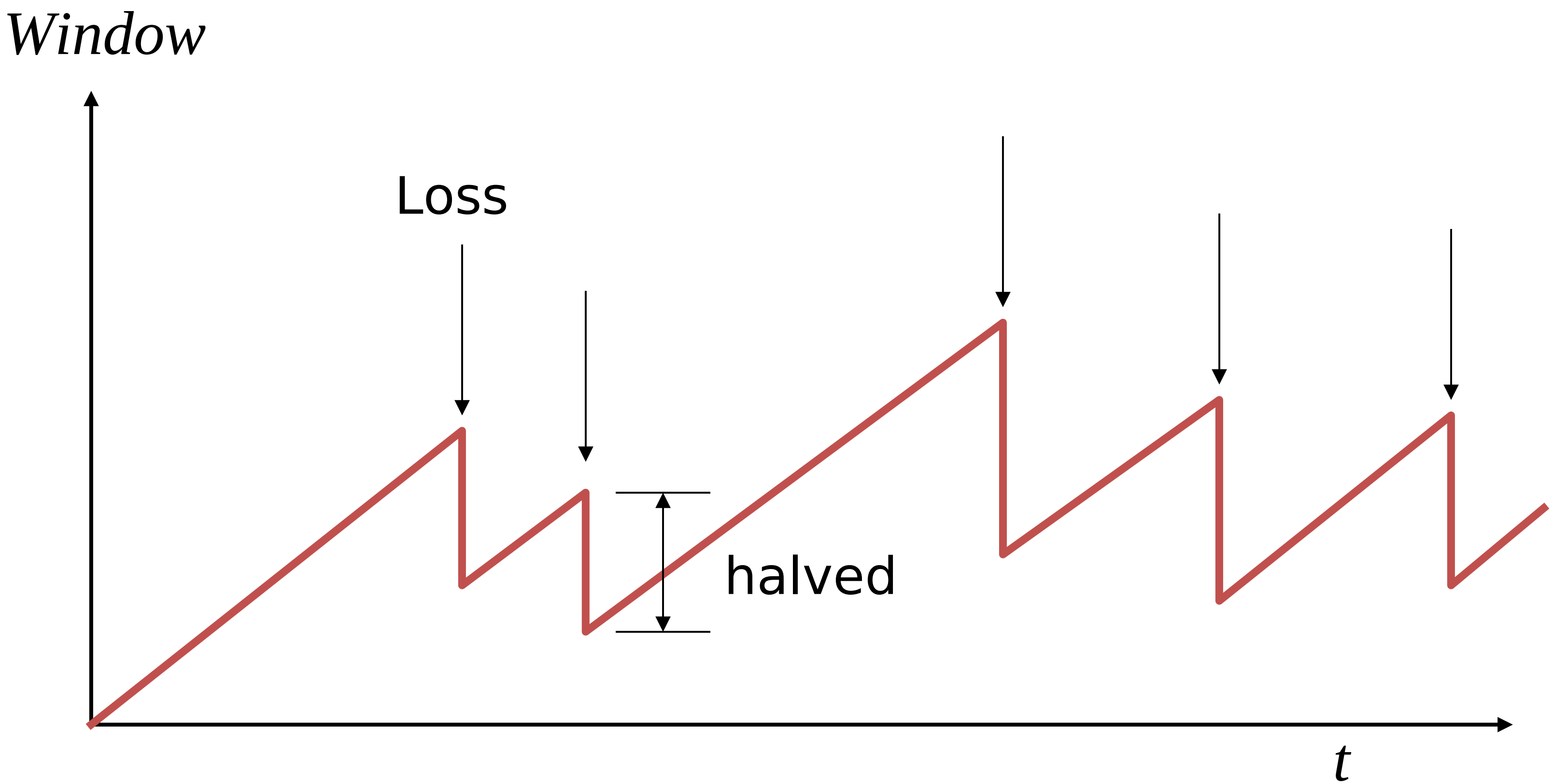
- Additive increase: On success of last window of data, increase window by 1 Max Segment Size (MSS)
- Multiplicative decrease: On loss of packet, divide congestion window in half

Much quicker to slow than speed up!

- Over-sized windows (causing loss) are much worse than under-sized windows (causing lower throughput)
- AIMD: A necessary condition for stability of TCP

Source: Freedman (partial)

Leads to TCP "Sawtooth"



Source: Freedman

Receiver Window vs. Congestion Window

Flow control

- Keep a *fast sender* from overwhelming *a slow receiver*

Congestion control

- Keep a *set of senders* from overloading the *network*

Different concepts, but similar mechanisms

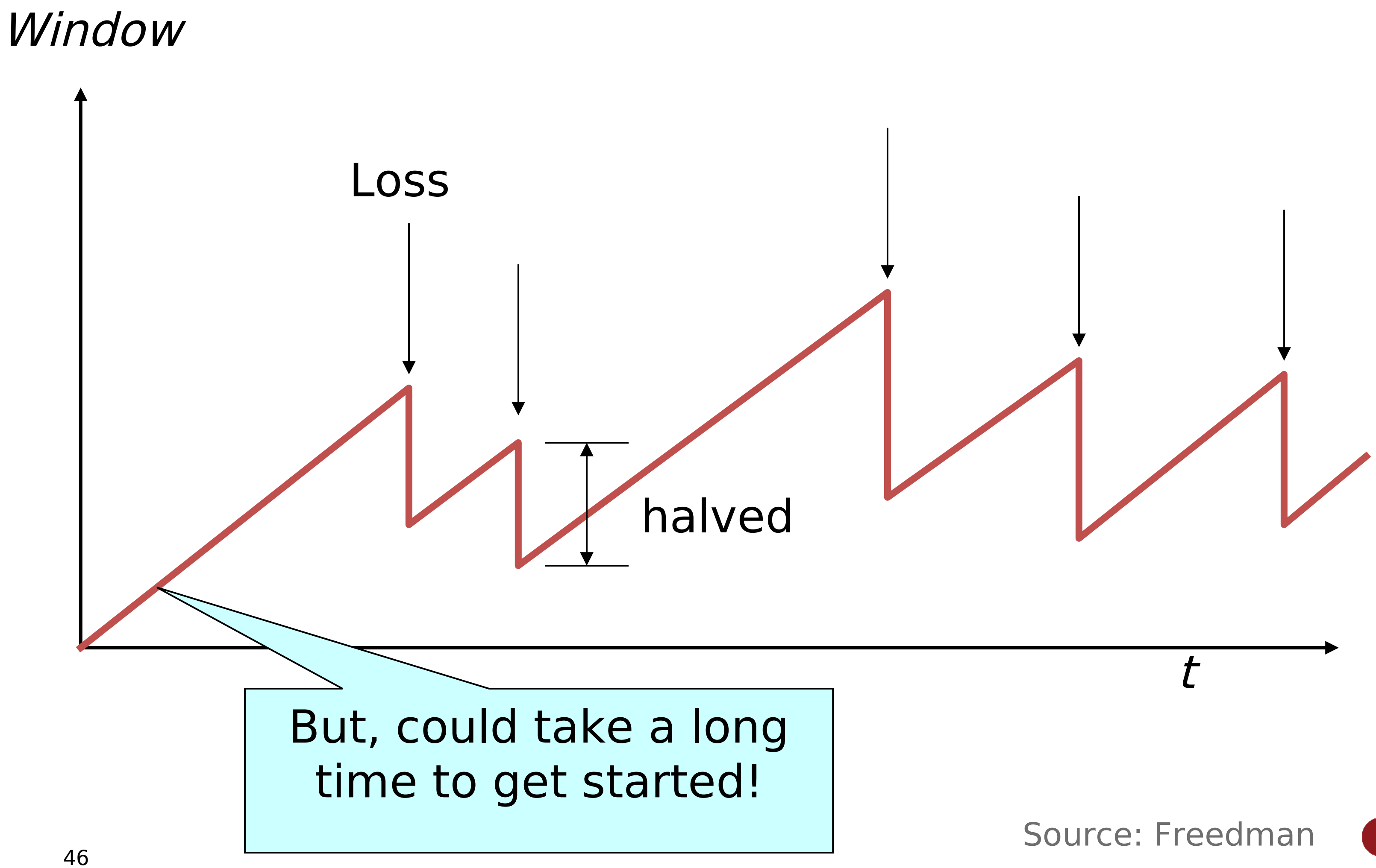
- TCP flow control: receiver window
- TCP congestion control: congestion window
- Sender TCP window =

  min { congestion window, receiver window }

Source: Freedman

How Should a New Flow Start?

Start slow (a small CWND) to avoid overloading network



Source: Freedman

“Slow Start” Phase

Start with a small congestion window

- Initially, CWND is 1 MSS
- So, initial sending rate is MSS / RTT

Could be pretty wasteful

- Might be much less than actual bandwidth
- Linear increase takes a long time to accelerate

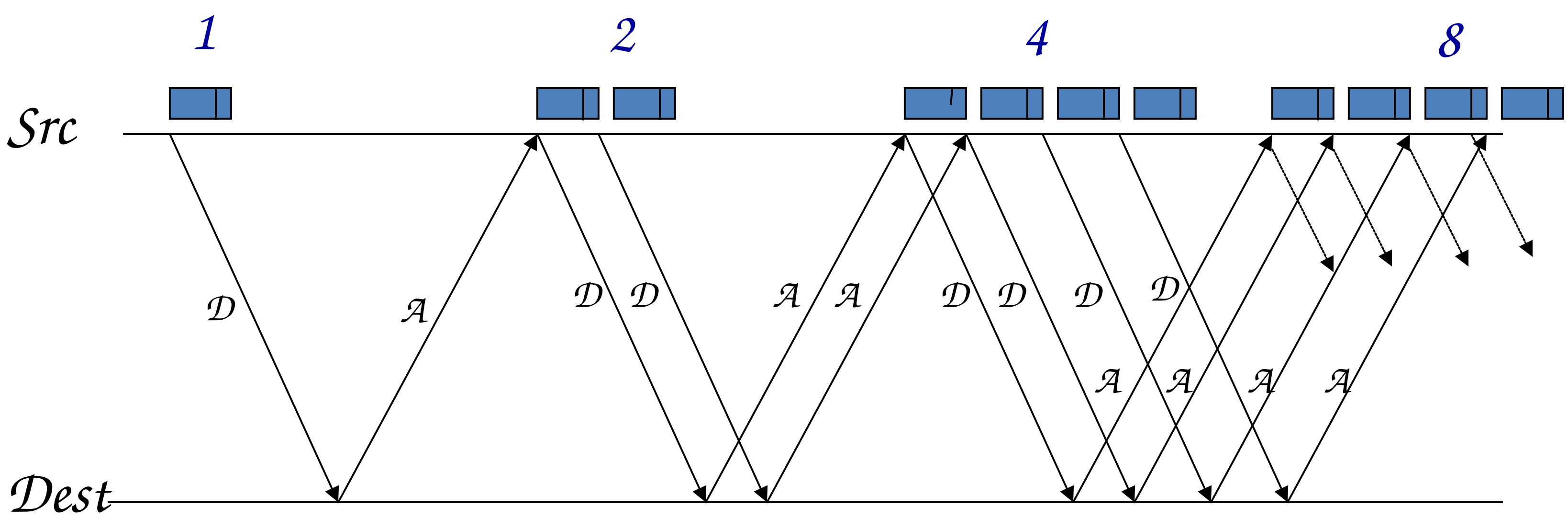
Slow-start phase (really “fast start”)

- Sender starts at a slow rate (hence the name)
- … but increases rate exponentially until the first loss

Source: Freedman

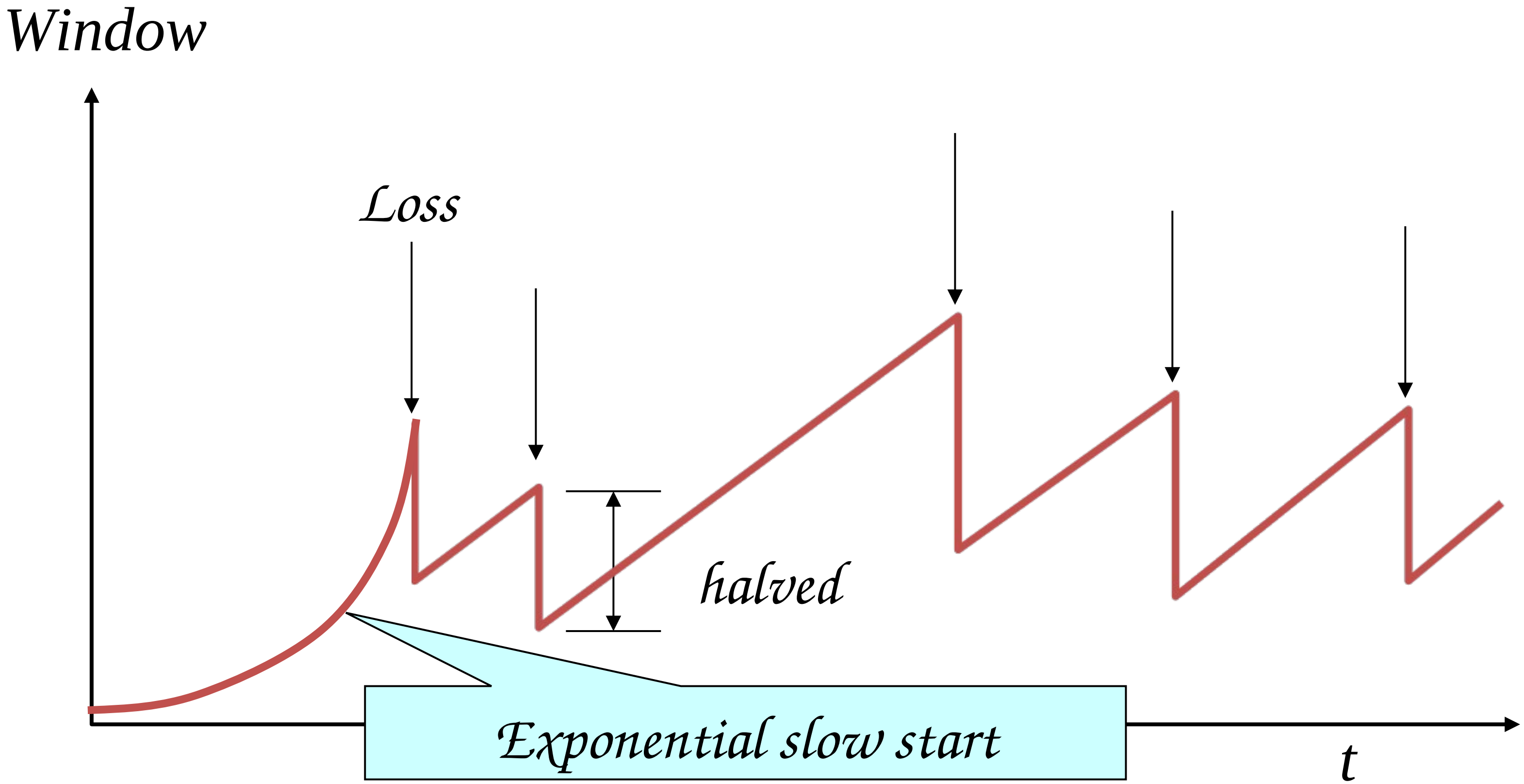
Slow Start in Action

Double CWND per round-trip time



Source: Freedman

Slow Start and the TCP Sawtooth



- So-called because TCP originally had no congestion control
  - Source would start by sending an entire receiver window
  - Led to congestion collapse!

Source: Freedman

Two Kinds of Loss in TCP

Timeout

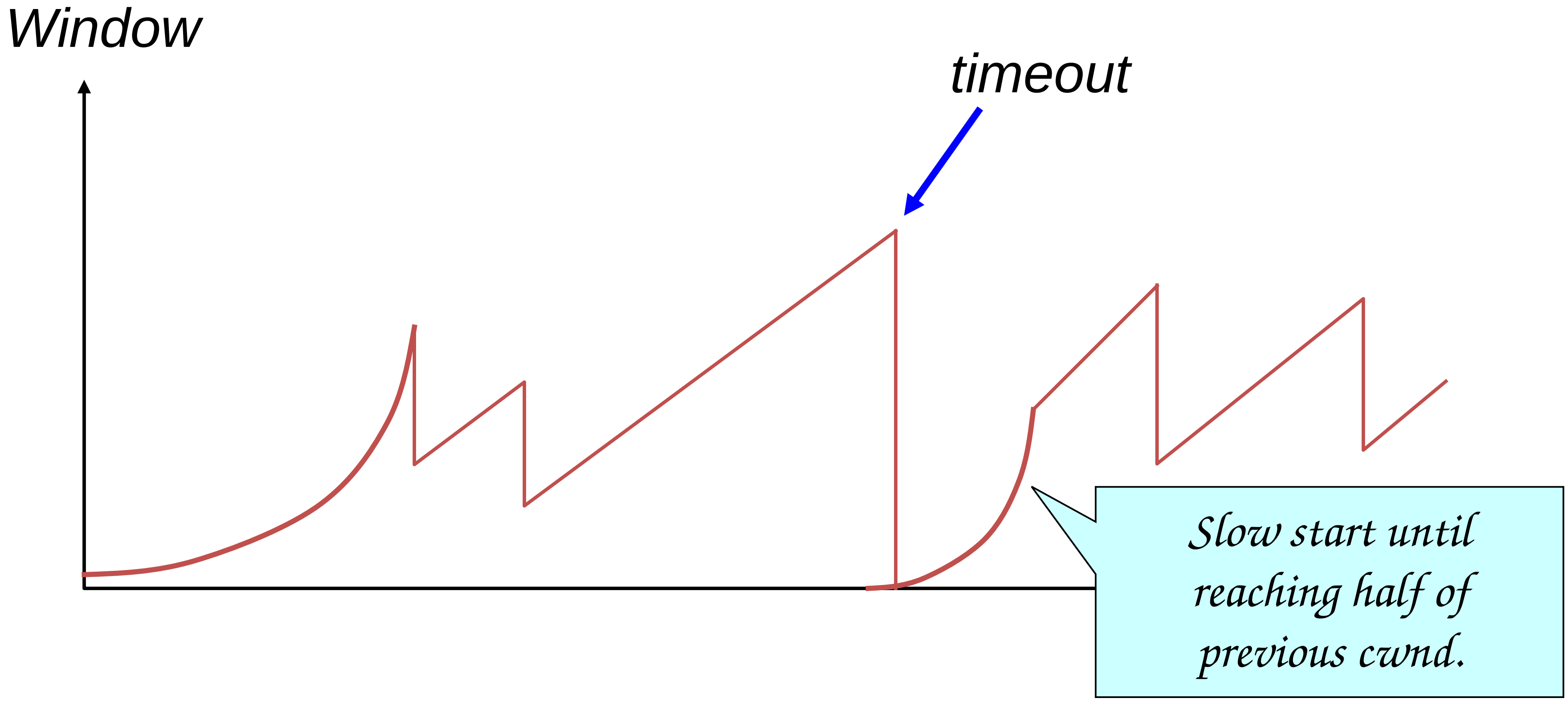
- Packet n is lost and detected via a timeout
  - When? n is last packet in window, or all packets in flight lost
- After timeout, blasting entire CWND would cause another burst
- Better to start over with a low CWND

Triple duplicate ACK

- Packet n is lost, but packets n+1, n+2, etc. arrive
  - How detected? Multiple ACKs that receiver waiting for n
  - When? Later packets after n received
- After triple duplicate ACK, sender quickly resends packet n
- Do a multiplicative decrease and keep

Source: Freedman

Repeating Slow Start After Timeout



Slow-start restart: Go back to CWND of 1, but take advantage of knowing the previous value of CWND.

Source: Freedman

Repeating Slow Start After Idle Period

Suppose a TCP connection goes idle for a while

Eventually, the network conditions change

- Maybe many more flows are traversing the link

Dangerous to start transmitting at the old rate

- Previously-idle TCP sender might blast network
- … causing excessive congestion and packet loss

So, some TCP implementations repeat slow start

- Slow-start restart after an idle period

Source: Freedman

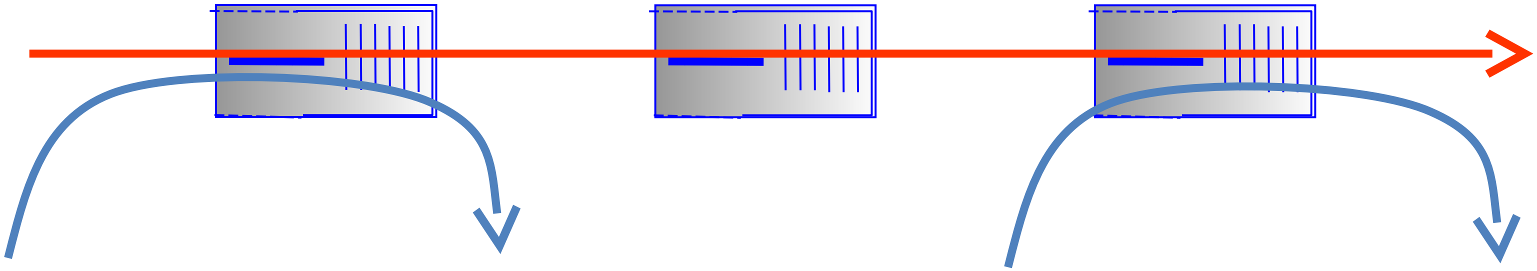
TCP Achieves Some Notion of Fairness

Effective utilization is not only goal

- We also want to be *fair* to various flows
- … but what does *that* mean?

Simple definition: equal shares of the bandwidth

- N flows that each get 1/N of the bandwidth?
- But, what if flows traverse different paths?
- Result: bandwidth shared in proportion to RTT



Source: Freedman

What About Cheating?

Some folks are more fair than others

- Running multiple TCP connections in parallel (BitTorrent)
- Modifying the TCP implementation in the OS
  - Some cloud services start TCP at > 1 MSS
- Use the User Datagram Protocol

What is the impact

- Good guys slow down to make room for you
- You get an unfair share of the bandwidth

Possible solutions?

- Routers detect cheating and drop excess packets?
- Per user/customer fairness?
- Peer pressure?

Source: Freedman

Summary

UDP

- basic multiplexing
- checksums

TCP & reliable transfer

- Segments, sequence numbers, automatic repeat requests
- Timeout estimation
- Pipelining, cumulative ACK, fast retransmit
- Flow control: receiver window
- Congestion control: congestion window, AIMD, slow start, slow start restart

Intro to A4

- Task is to implement a HTTP Server
- Must be implemented in Python
- No handout, but examples in exercises that may act as inspiration
- HTTP protocol defined in RFC 2616
- Do not implement all, only GET, HEAD and support for a few headers

HTTP Request Example

GET / HTTP/1.1

Host: sns.cs.princeton.edu

Accept: */*

Accept-Language: en-us

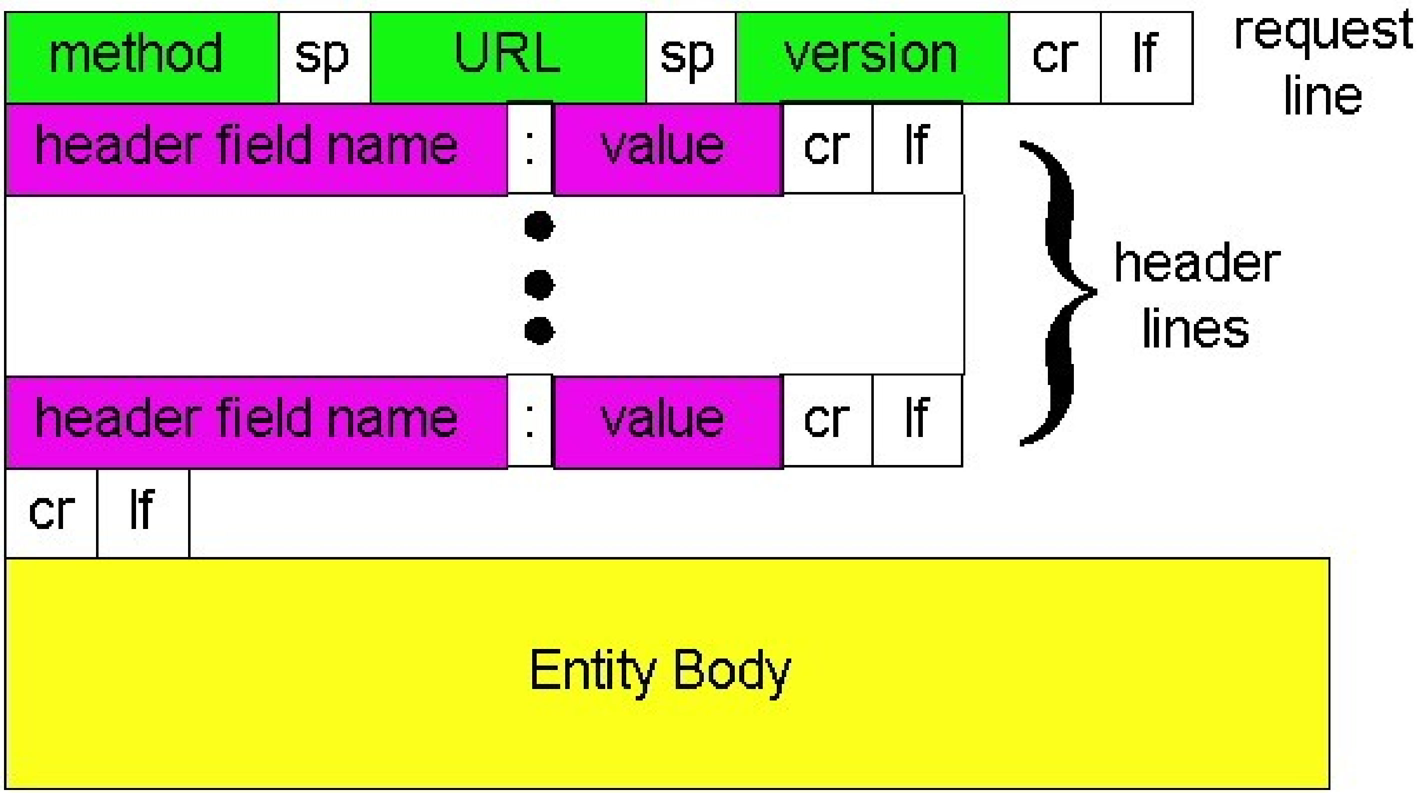
Accept-Encoding: gzip, deflate

User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.2.13) Gecko/20101203 Firefox/3.6.13

Connection: Keep-Alive

Source: Freedman

HTTP Request



Source: Freedman

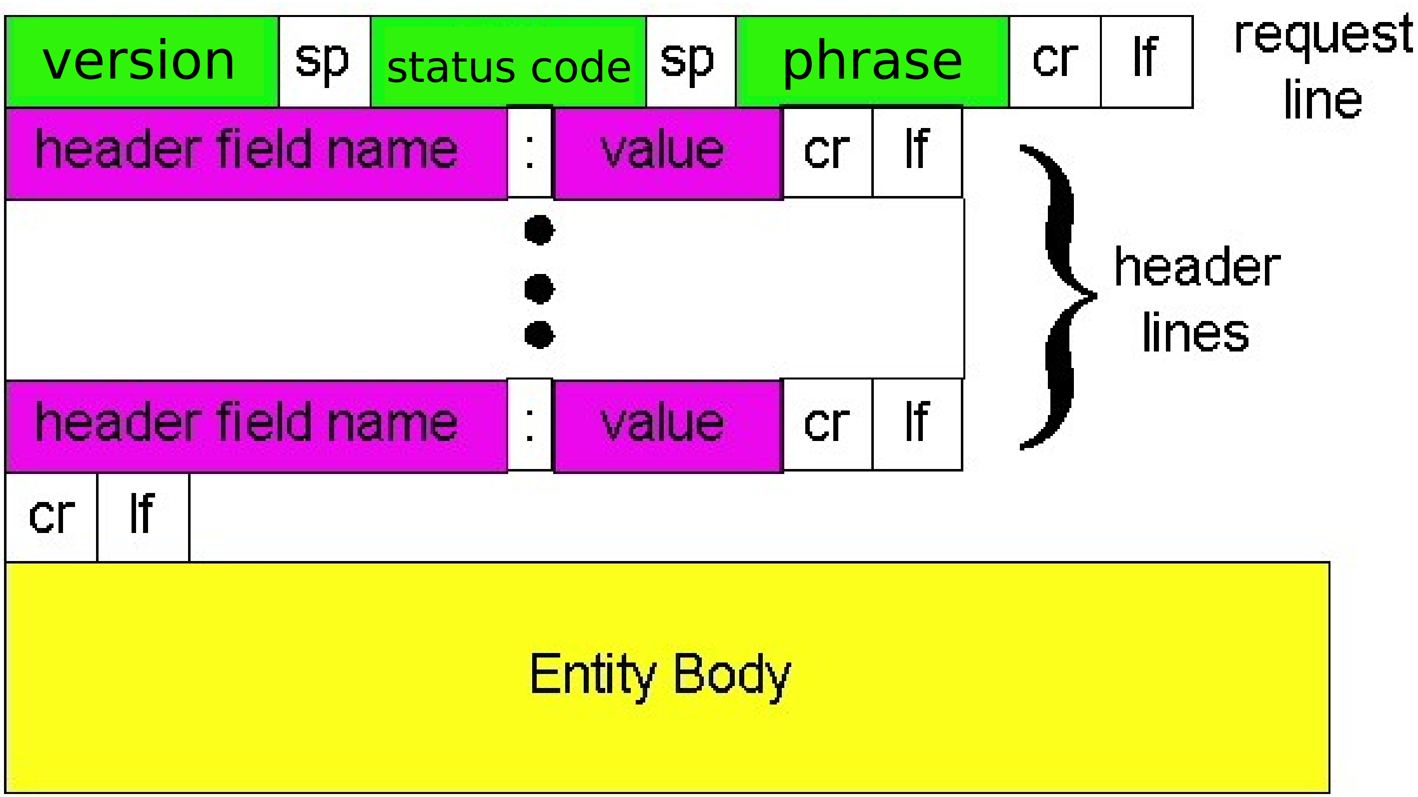
HTTP Response Example

HTTP/1.1 200 OK
Date: Wed, 02 Feb 2011 04:01:21 GMT
Server: Apache/2.2.3 (CentOS)
X-Pingback: http://sns.cs.princeton.edu/xmlrpc.php
Last-Modified: Wed, 01 Feb 2011 12:41:51 GMT
ETag: "7a11f-10ed-3a75ae4a"
Accept-Ranges: bytes
Content-Length: 4333
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" dir="ltr" lang="en-US">

Source: Freedman

HTTP Response



Source: Freedman