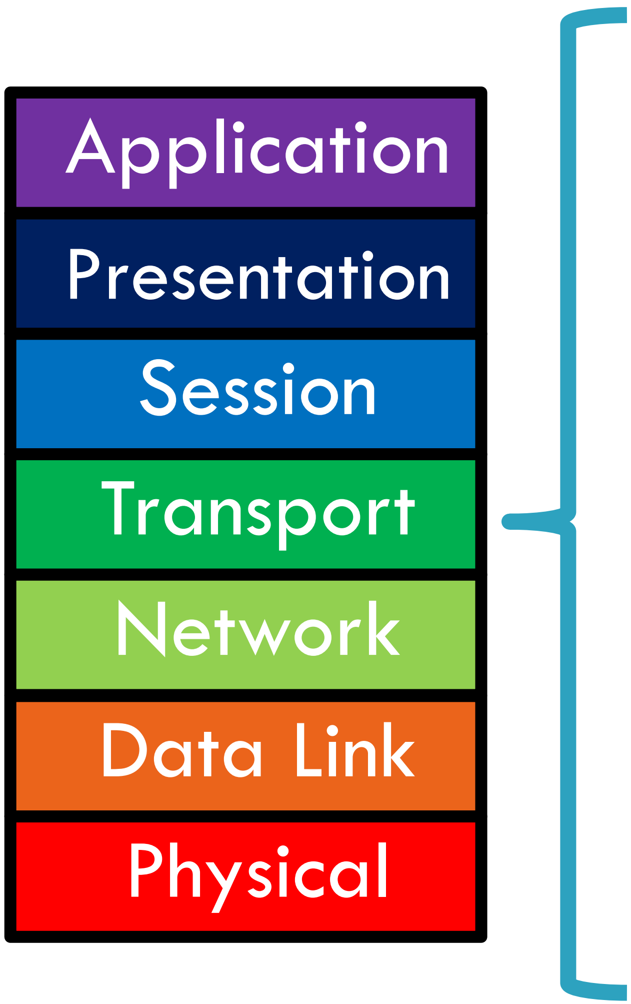
Computer Networks

Lecture 11-12: Transport layer Part II
DNS, HTTP

Based on slides from D. Choffnes Northeastern U. and P. Gill from StonyBrook University
Revised Autumn 2015 by S. Laki

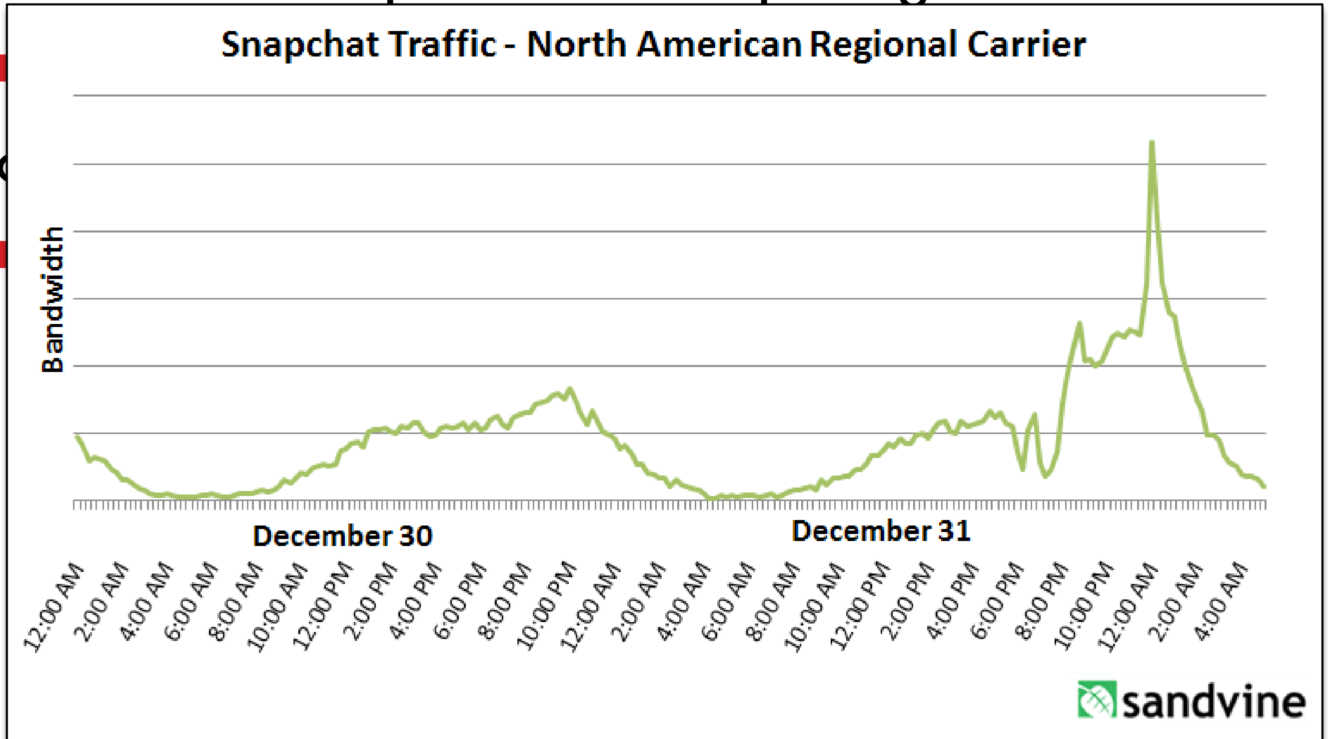
Transport Layer



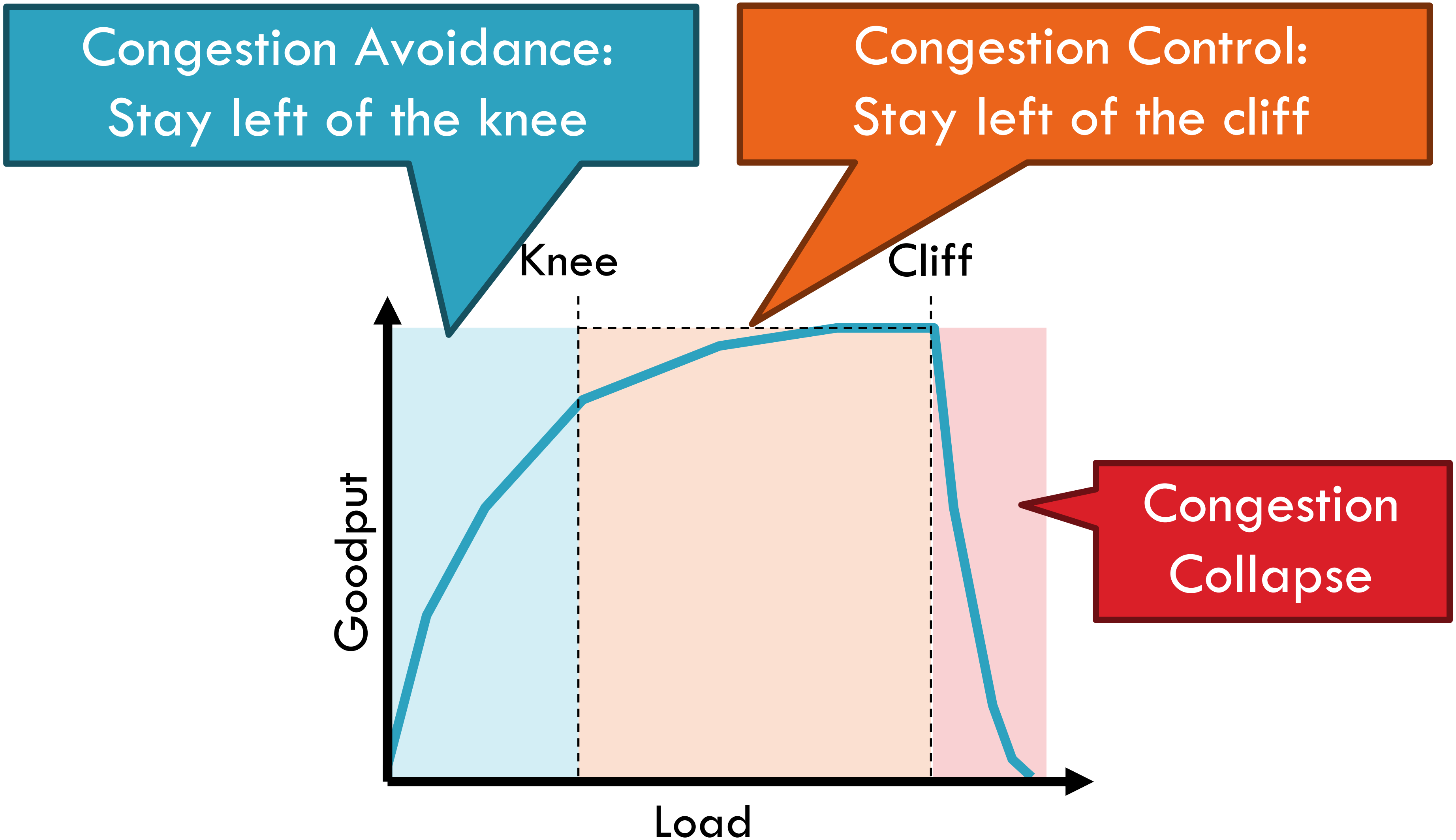
- Function:
  - Demultiplexing of data streams
- Optional functions:
  - Creating long lived connections
  - Reliable, in-order packet delivery
  - Error detection
  - Flow and congestion control
- Key challenges:
  - Detecting and responding to congestion
  - Balancing fairness against high utilization

What is Congestion?

- Load on the network is higher than capacity
  - Capacity is not uniform across networks
    - Modem vs. Cellular vs. Cable vs. Fiber Optics
  - There are multiple flows competing for bandwidth
  - L



Cong. Control vs. Cong. Avoidance



TCP Congestion Control

- Each TCP connection has a window
  - Controls the number of unACKed packets
- Sending rate is ~ window/RTT
- Idea: vary the window size to control the send rate
- Introduce a congestion window at the sender
  - Congestion control is sender-side problem

Two Basic Components

1. Detect congestion
   - Packet dropping is most reliably signal
     - Delay-based methods are hard and risky
   - How do you detect packet drops? ACKs
     - Timeout after not receiving an ACK
     - Several duplicate ACKs in a row (ignore for now)
2. Rate adjustment algorithm
   - Modify *cwnd*
   - Probe for bandwidth
   - Responding to congestion

Rate Adjustment

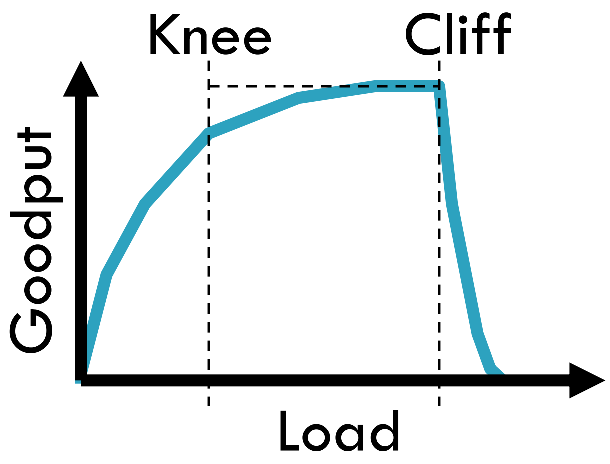
- Recall: TCP is ACK clocked
  - Congestion = delay = long wait between ACKs
  - No congestion = low delay = ACKs arrive quickly
- Basic algorithm
  - Upon receipt of ACK: increase cwnd
    - Data was delivered, perhaps we can send faster
    - *cwnd* growth is proportional to RTT
  - On loss: decrease cwnd
    - Data is being lost, there must be congestion
- Question: increase/decrease functions to use? !!!!

Implementing Congestion Control

- Maintains three variables:
  - *cwnd*: congestion window
  - *adv_wnd*: receiver advertised window
  - *ssthresh*: threshold size (used to update *cwnd*)
- For sending, use: $wnd = min(cwnd, adv\_wnd)$
- Two phases of congestion control
  1. Slow start ($cwnd < ssthresh$)
     - Probe for bottleneck bandwidth
  2. Congestion avoidance ($cwnd >= ssthresh$)
     - AIMD

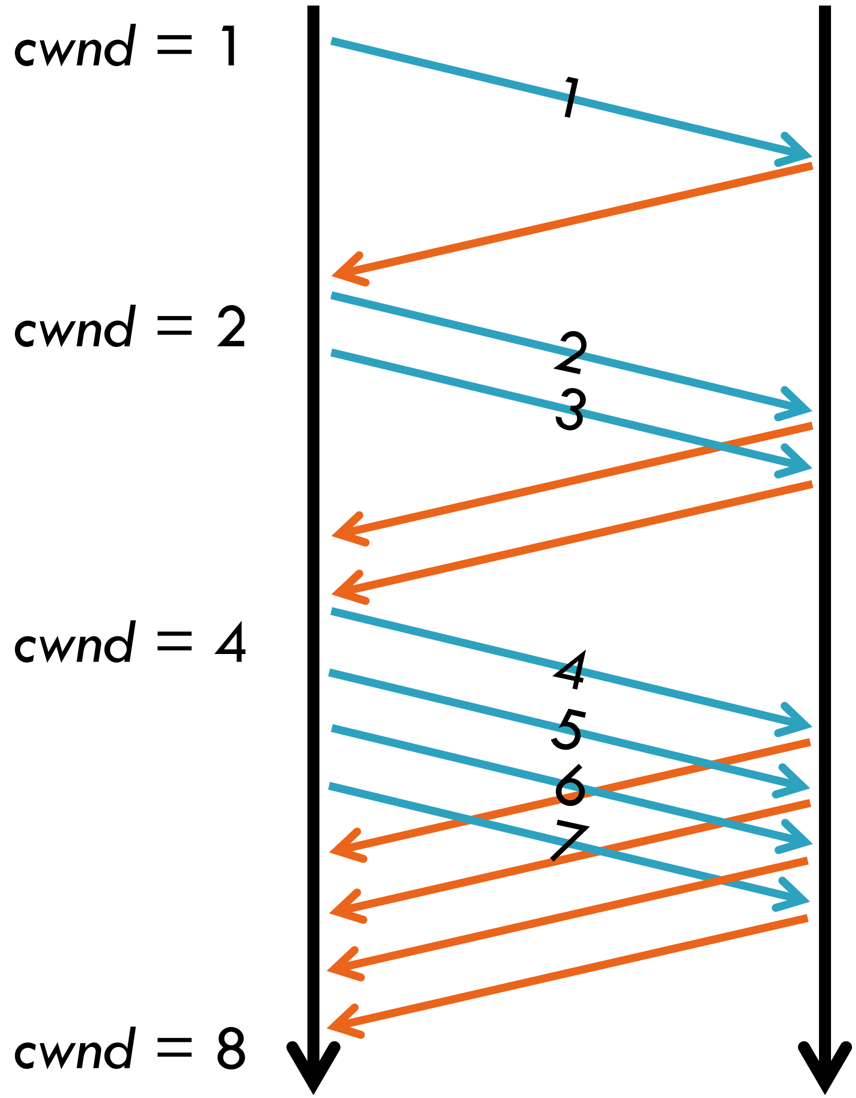
Slow Start

- Goal: reach knee quickly
- Upon starting (or restarting) a connection
  - *cwnd* =1
  - *ssthresh* = *adv_wnd*
  - Each time a segment is ACKed, *cwnd*++
- Continues until…
  - *ssthresh* is reached
  - Or a packet is lost
- Slow Start is not actually slow
  - *cwnd* increases exponentially



Slow Start Example

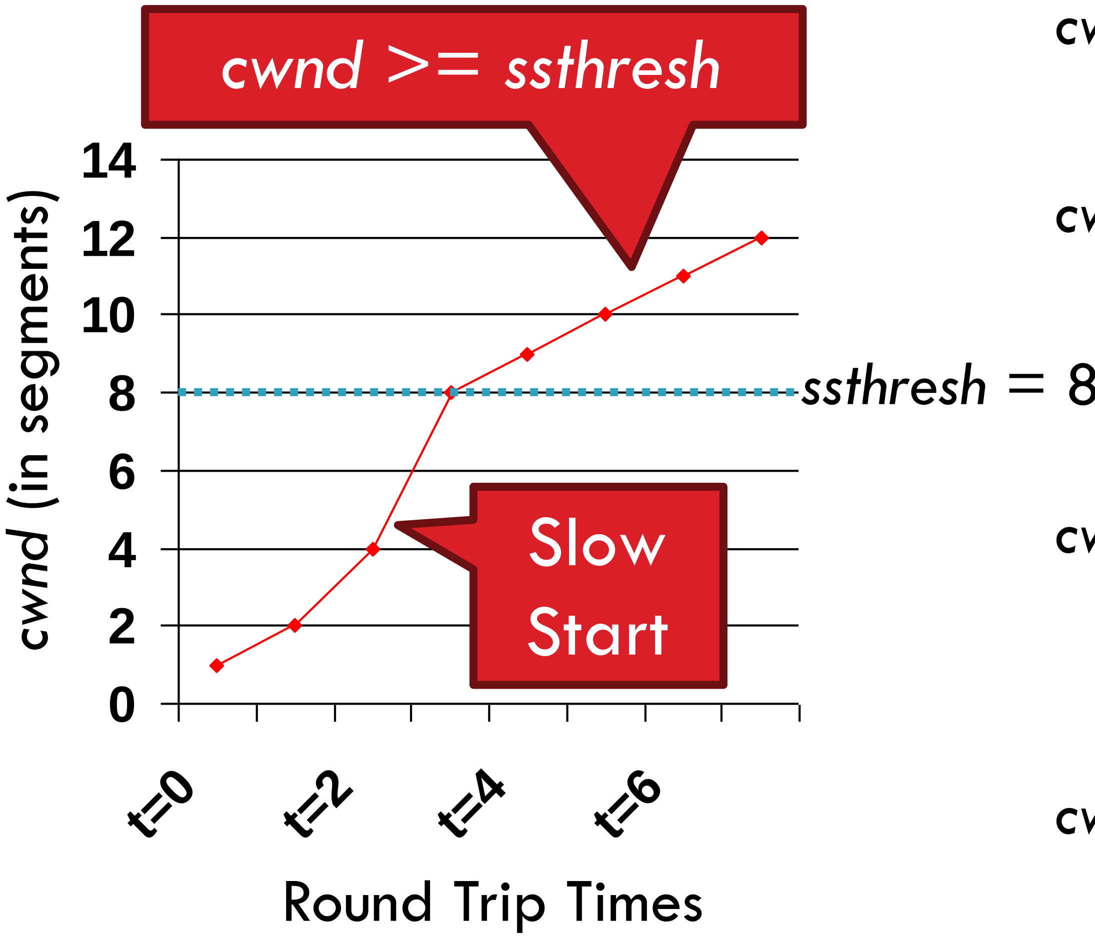
- *cwnd* grows rapidly
- Slows down when…
  - *cwnd* >= *ssthresh*
  - Or a packet drops

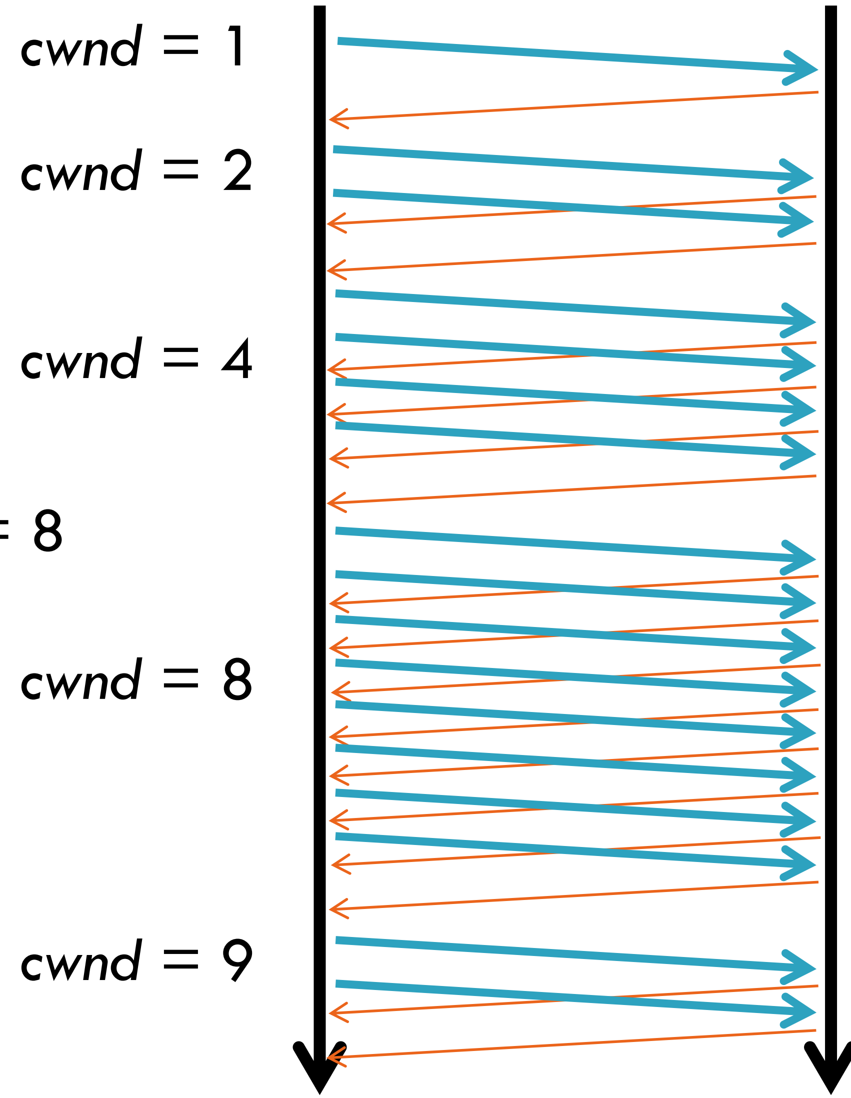


Congestion Avoidance

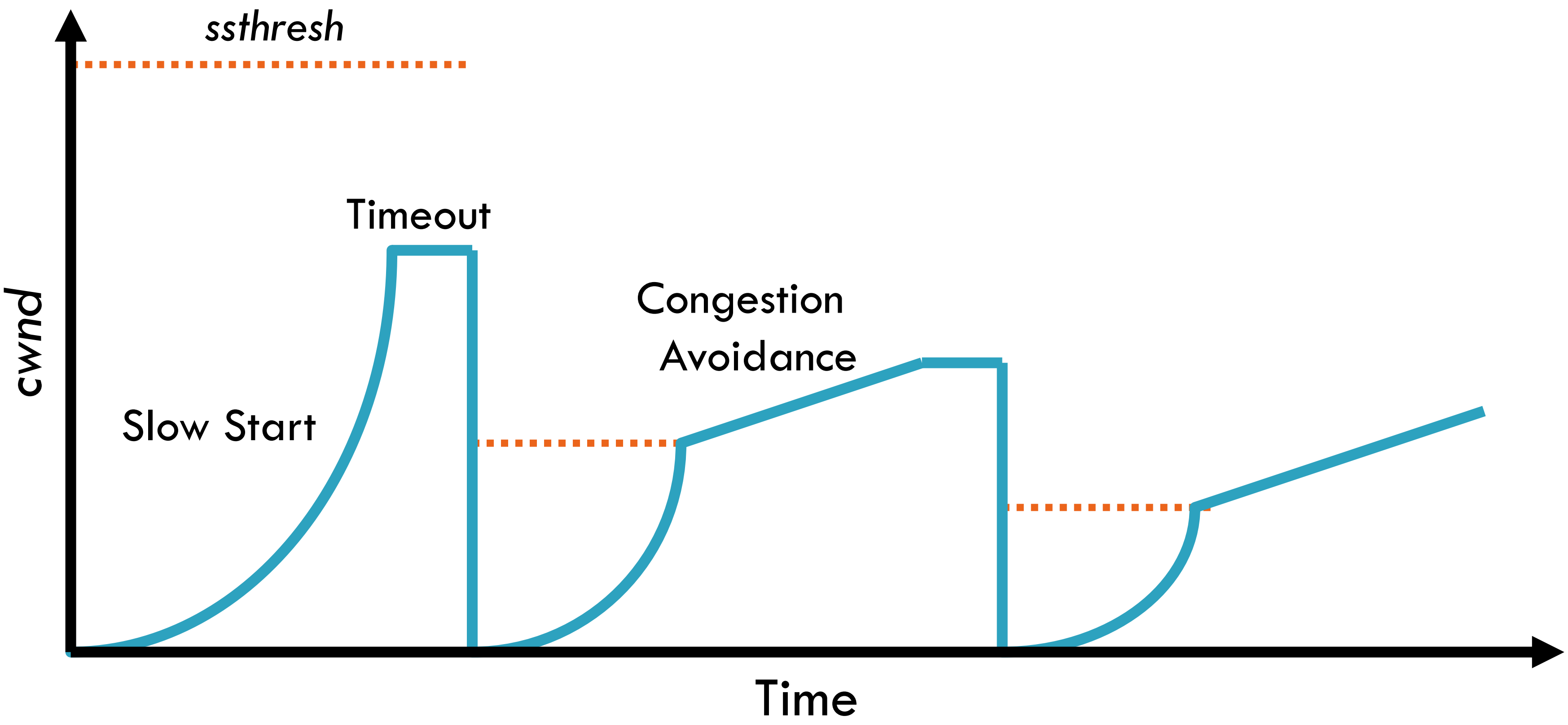
- Additive Increase Multiplicative Decrease (AIMD) mode
- *ssthresh* is lower-bound guess about location of the knee
- **If** *cwnd* >= *ssthresh* **then**
  each time a segment is ACKed
  increment *cwnd by 1/cwnd* (*cwnd* += *1/cwnd*).
- So *cwnd* is increased by one only if all segments have been acknowledged

Congestion Avoidance Example





The Big Picture – TCP Tahoe (the original TCP)



Outline

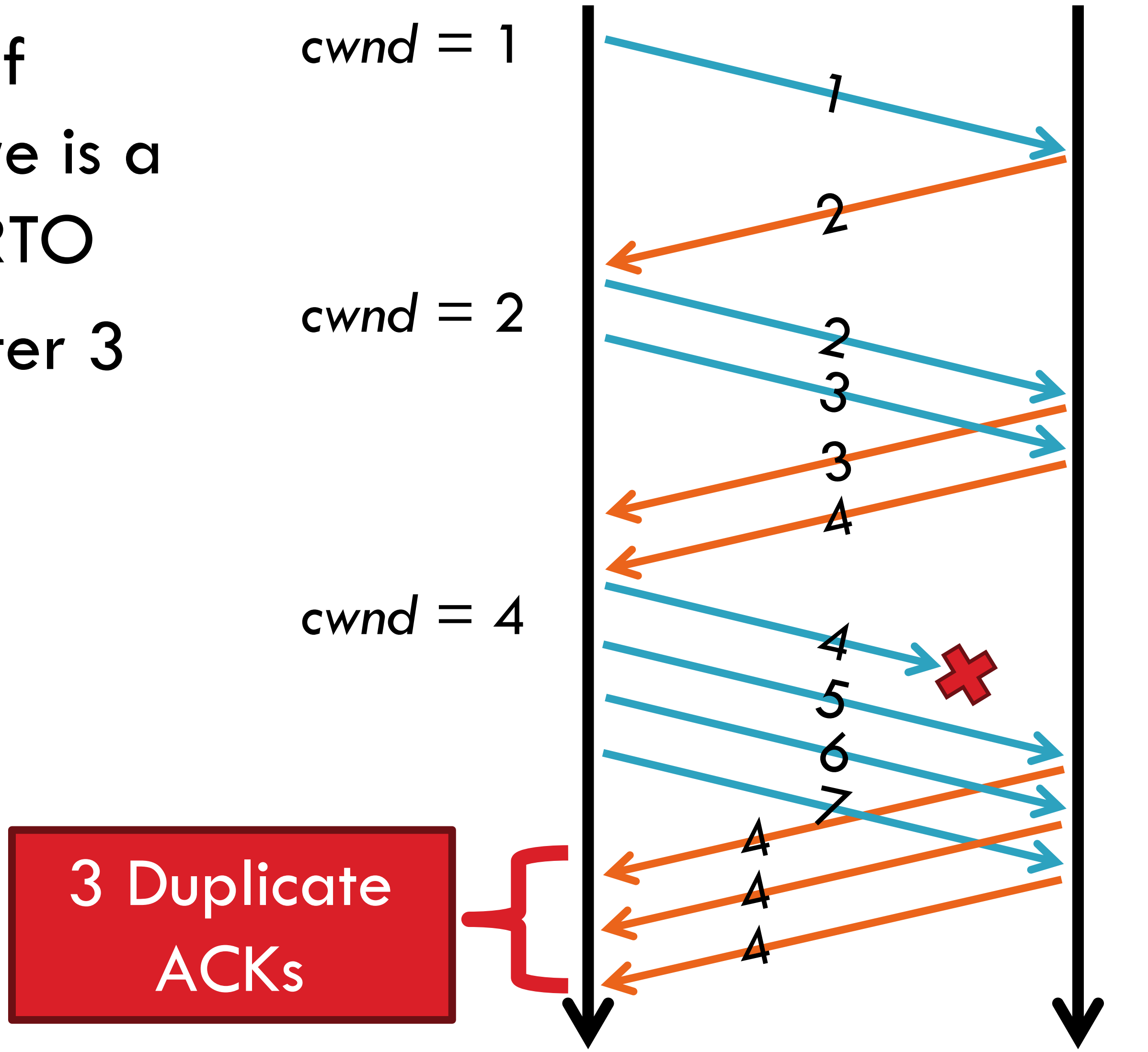
- UDP
- TCP
- Congestion Control
- **Evolution of TCP**
- Problems with TCP

The Evolution of TCP

- Thus far, we have discussed TCP Tahoe
  - Original version of TCP
- However, TCP was invented in 1974!
  - Today, there are many variants of TCP
- Early, popular variant: TCP Reno
  - Tahoe features, plus…
  - Fast retransmit
    - 3 duplicate ACKs? -> retransmit (don't wait for RTO)
  - Fast recovery
    - On loss: set cwnd = cwnd/2 (ssthresh = new cwnd value)

TCP Reno: Fast Retransmit

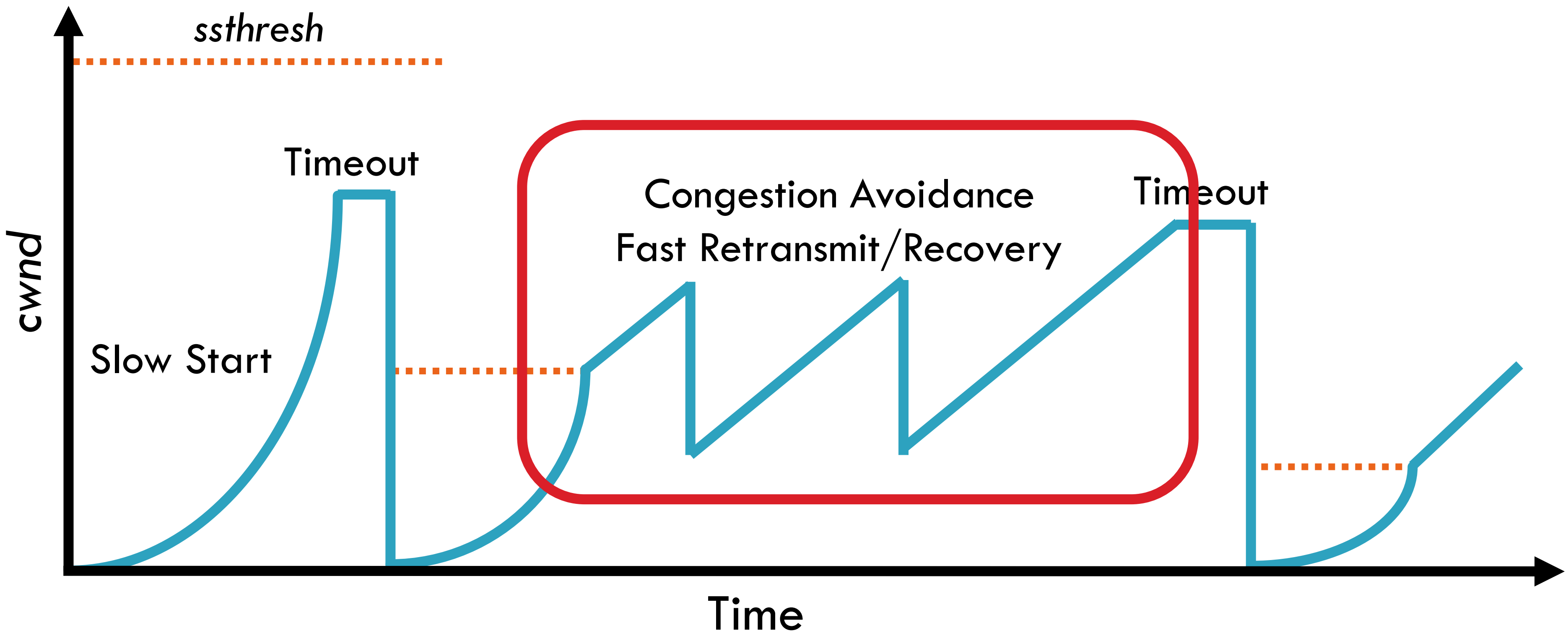
- Problem: in Tahoe, if segment is lost, there is a long wait until the RTO
- Reno: retransmit after 3 duplicate ACKs



TCP Reno: Fast Recovery

- After a fast-retransmit set *cwnd* to *cwnd*/2
  - Also reset ssthresh to the new halved cwnd value
  - i.e. don't reset *cwnd* to 1
  - Avoid unnecessary return to slow start
  - Prevents expensive timeouts
- But when RTO expires still do *cwnd* = 1
  - Return to slow start, same as Tahoe
  - Indicates packets aren't being delivered at all
  - i.e. congestion must be really bad

Fast Retransmit and Fast Recovery



- At steady state, *cwnd* oscillates around the optimal window size
- TCP always forces packet drops

Many TCP Variants…

- Tahoe: the original
  - Slow start with AIMD
  - Dynamic RTO based on RTT estimate
- Reno:
  - fast retransmit (3 dupACKs)
  - fast recovery (cwnd = cwnd/2 on loss)
- NewReno: improved fast retransmit
  - Each duplicate ACK triggers a retransmission
  - Problem: >3 out-of-order packets causes pathological retransmissions
- Vegas: delay-based congestion avoidance
- And many, many, many more…

TCP in the Real World

- What are the most popular variants today?
  - Key problem: TCP performs poorly on high bandwidth-delay product networks (like the modern Internet)
  - Compound TCP (Windows)
    - Based on Reno
    - Uses two congestion windows: delay based and loss based
    - Thus, it uses a *compound* congestion controller
  - TCP CUBIC (Linux)
    - Enhancement of BIC (Binary Increase Congestion Control)
    - Window size controlled by cubic function
    - Parameterized by the time *T* since the last dropped packet

High Bandwidth-Delay Product

- Key Problem: TCP performs poorly when
  - The capacity of the network (bandwidth) is large
  - The delay (RTT) of the network is large
  - Or, when bandwidth * delay is large
    - b * d = maximum amount of in-flight data in the network
    - a.k.a. the bandwidth-delay product
- Why does TCP perform poorly?
  - Slow start and additive increase are slow to converge
  - TCP is ACK clocked
    - i.e. TCP can only react as quickly as ACKs are received
    - Large RTT → ACKs are delayed → TCP is slow to react

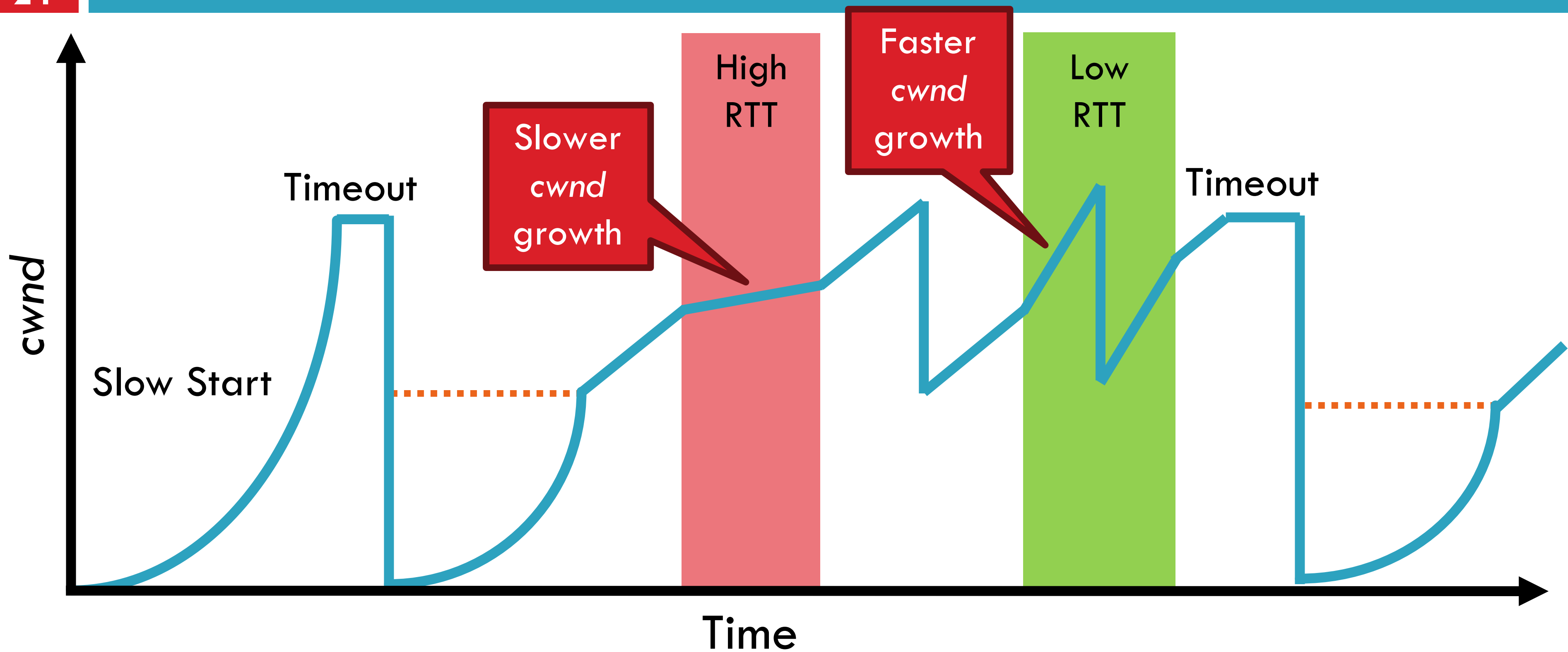
Goals

- Fast window growth
  - Slow start and additive increase are too slow when bandwidth is large
  - Want to converge more quickly
- Maintain fairness with other TCP varients
  - Window growth cannot be too aggressive
- Improve RTT fairness
  - TCP Tahoe/Reno flows are not fair when RTTs vary widely
- Simple implementation

Compound TCP Implementation

- Default TCP implementation in Windows
- Key idea: split *cwnd* into two separate windows
  - Traditional, loss-based window
  - New, delay-based window
- $wnd = \min(cwnd + dwnd, adv\_wnd)$
  - *cwnd* is controlled by AIMD
  - *dwnd* is the delay window
- Rules for adjusting *dwnd:*
  - If RTT is increasing, decrease *dwnd* (*dwnd* >= 0)
  - If RTT is decreasing, increase *dwnd*
  - Increase/decrease are proportional to the rate of change

Compound TCP Example



- Aggressiveness corresponds to changes in RTT
- Advantages: fast ramp up, more fair to flows with different RTTs
- Disadvantage: must estimate RTT, which is very challenging

TCP CUBIC Implementation

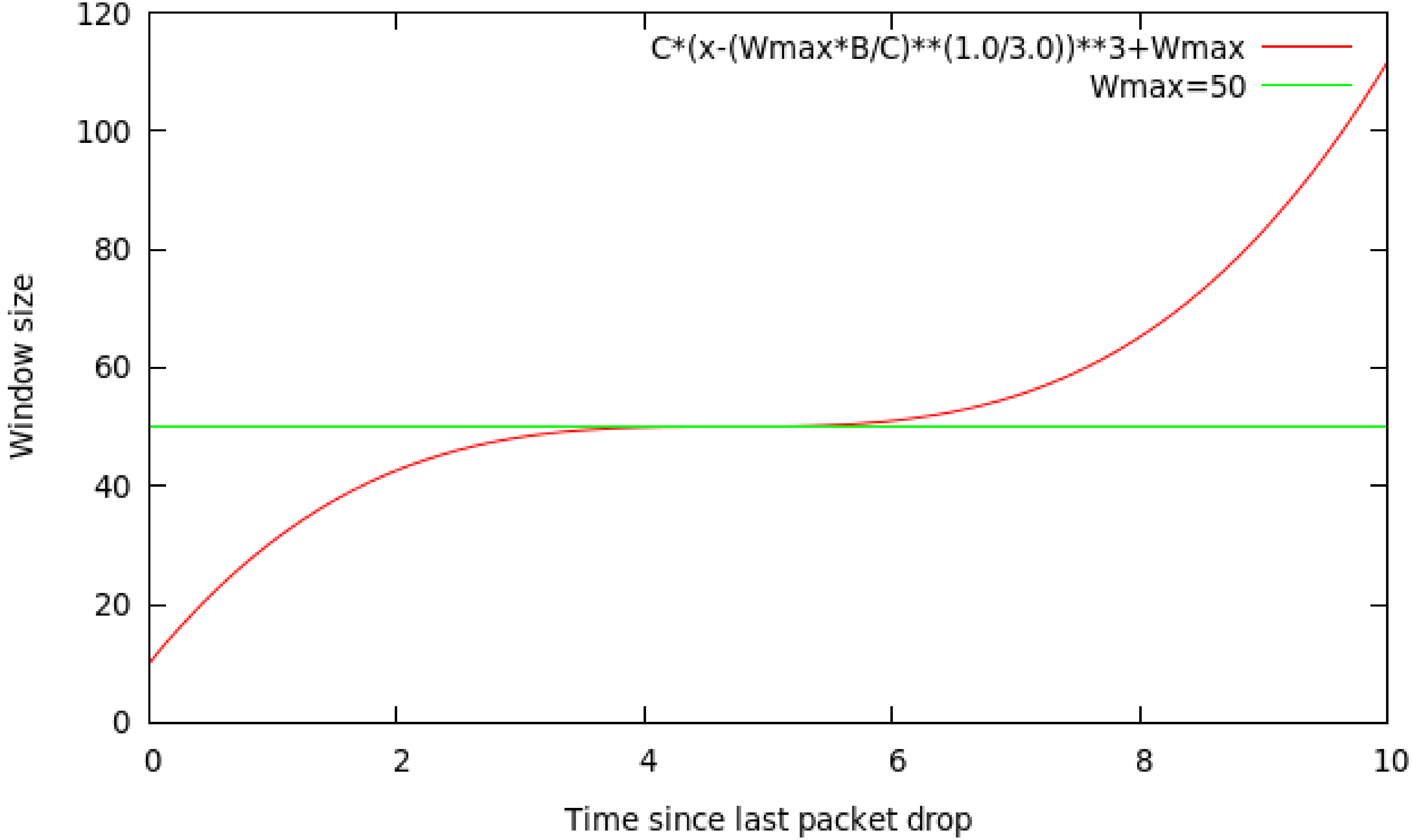
- Default TCP implementation in Linux
- Replace AIMD with cubic function

$$W_{cubic} = C(T - K)^3 + W_{max} \qquad (1)$$

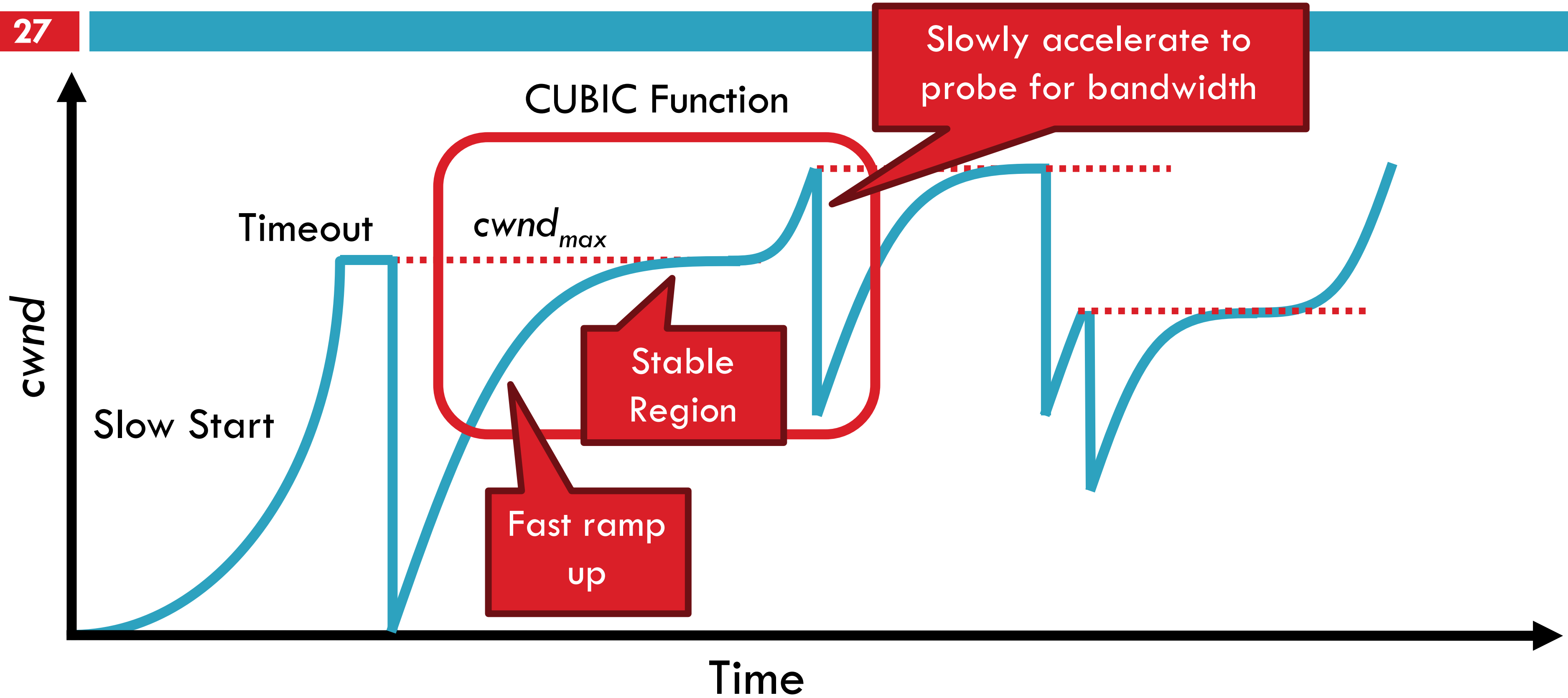
$$\text{C is a scaling constant, and K} = \sqrt[3]{\frac{W_{max}\beta}{C}}$$

  - B → a constant fraction for multiplicative increase
  - T → time since last packet drop
  - W_max → cwnd when last packet dropped

TCP CUBIC Implementation



TCP CUBIC Example



- Less wasted bandwidth due to fast ramp up
- Stable region and slow acceleration help maintain fairness
  - Fast ramp up is more aggressive than additive increase
  - To be fair to Tahoe/Reno, CUBIC needs to be less aggressive

Outline

- UDP
- TCP
- Congestion Control
- Evolution of TCP
- Problems with TCP

Issues with TCP

- The vast majority of Internet traffic is TCP
- However, many issues with the protocol
  - Poor performance with small flows
  - Really poor performance on wireless networks
  - Susceptibility to denial of service

Small Flows

- Problem: TCP is biased against short flows
  - 1 RTT wasted for connection setup (SYN, SYN/ACK)
  - *cwnd* always starts at 1
- Vast majority of Internet traffic is short flows
  - Mostly HTTP transfers, <100KB
  - Most TCP flows never leave slow start!
- Proposed solutions (driven by Google):
  - Increase initial *cwnd* to 10
  - TCP Fast Open: use cryptographic hashes to identify receivers, eliminate the need for three-way handshake

Wireless Networks

- Problem: Tahoe and Reno assume loss = congestion
  - True on the WAN, bit errors are very rare
  - False on wireless, interference is very common
- TCP throughput ~ 1/sqrt(drop rate)
  - Even a few interference drops can kill performance
- Possible solutions:
  - Break layering, push data link info up to TCP
  - Use delay-based congestion detection (TCP Vegas)
  - Explicit congestion notification (ECN)

Denial of Service

- Problem: TCP connections require state
  - Initial SYN allocates resources on the server
  - State must persist for several minutes (RTO)
- SYN flood: send enough SYNs to a server to allocate all memory/meltdown the kernel
- Solution: SYN cookies
  - Idea: don’t store initial state on the server
  - Securely insert state into the SYN/ACK packet (sequence number field)
  - Client will reflect the state back to the server

DNS

Layer 8 (The Carbon-based nodes)

- If you want to…
  - Call someone, you need to ask for their phone number
    - You can’t just dial “P R O F G I L L ”
  - Mail someone, you need to get their address first
- What about the Internet?
  - If you need to reach Google, you need their IP
  - Does anyone know Google’s IP?
- Problem:
  - People can’t remember IP addresses
  - Need human readable names that map to IPs

Internet Names and Addresses

- Addresses, e.g. 129.10.117.100
  - Computer usable labels for machines
  - Conform to structure of the network
- Names, e.g. www.northeastern.edu
  - Human usable labels for machines
  - Conform to organizational structure
- How do you map from one to the other?
  - Domain Name System (DNS)

History

- Before DNS, all mappings were in *hosts.txt*
  - */etc/hosts* on Linux
  - *C:\Windows\System32\drivers\etc\hosts* on Windows
- Centralized, manual system
  - Changes were submitted to SRI via email
  - Machines periodically FTP new copies of *hosts.txt*
  - Administrators could pick names at their discretion
  - Any name was allowed
    - alans_server_at_sbu_pwns_joo_lol_kthxbye

Towards DNS

- Eventually, the *hosts.txt* system fell apart
  - Not scalable, SRI couldn't handle the load
  - Hard to enforce uniqueness of names
    - e.g MIT
      - Massachusetts Institute of Technology?
      - Melbourne Institute of Technology?
  - Many machines had inaccurate copies of *hosts.txt*
- Thus, DNS was born

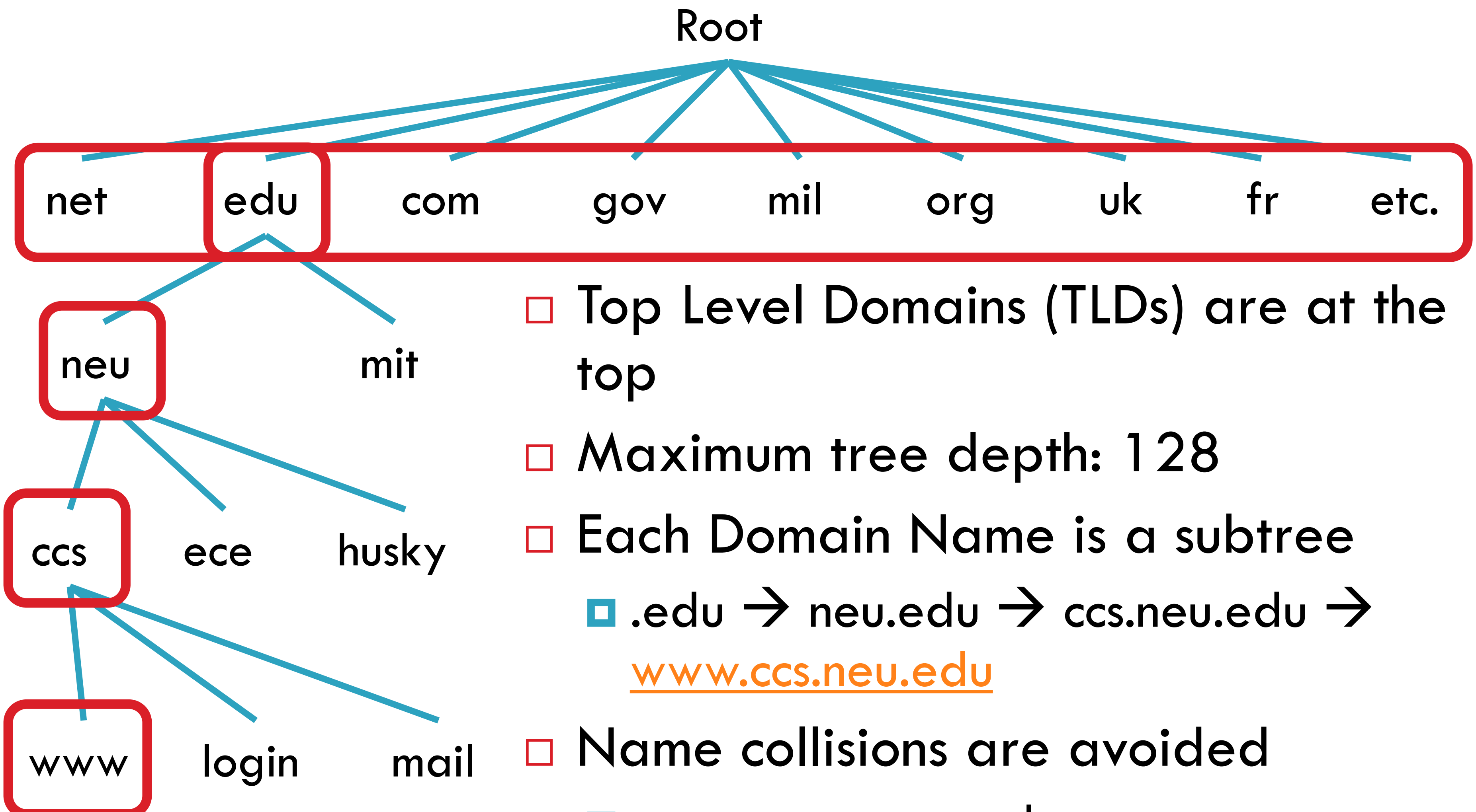
Outline

- DNS Basics
- DNS Security
- DNS and Censorship

DNS at a High-Level

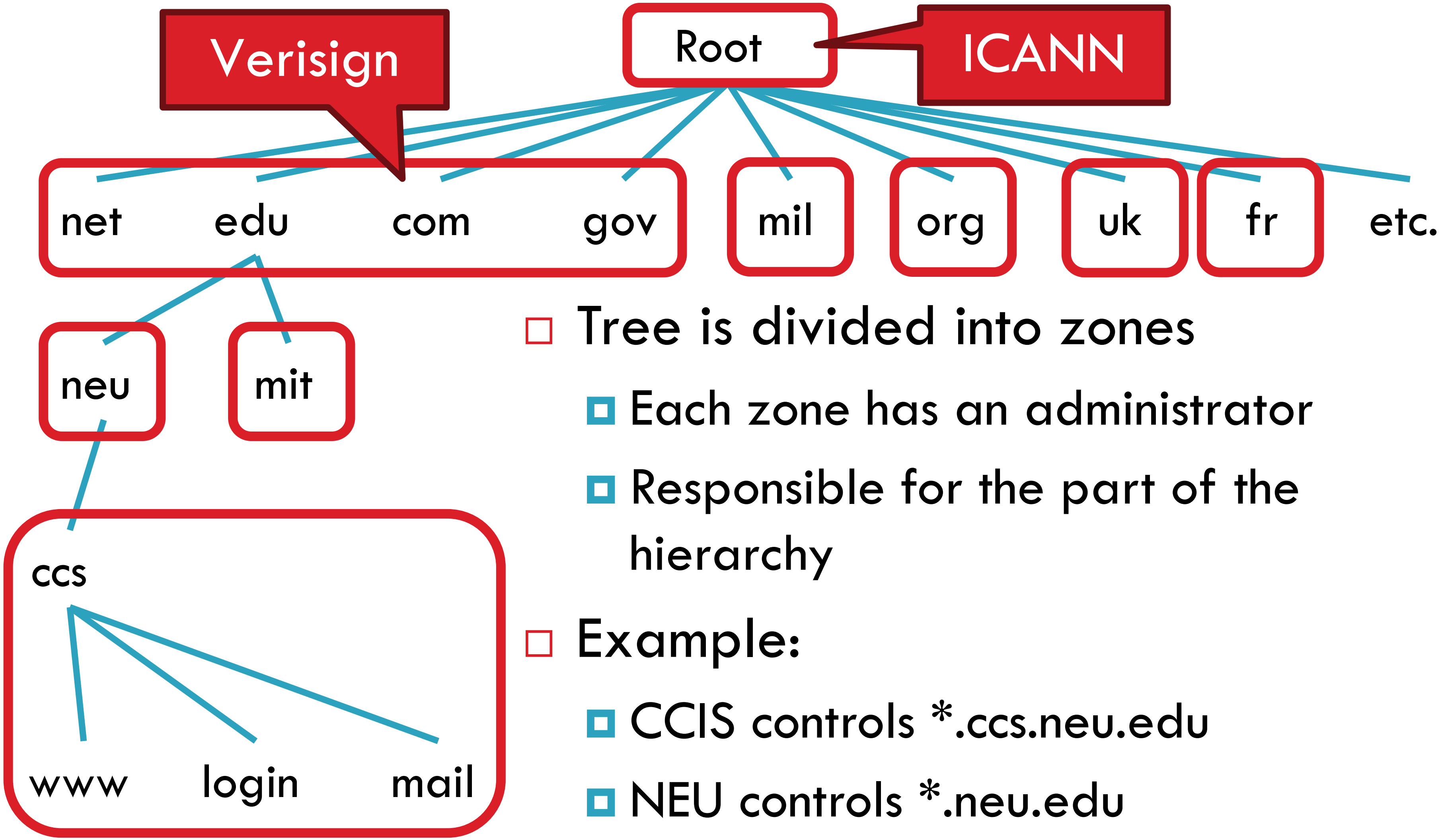
- Domain Name System
- Distributed database
  - No centralization
- Simple client/server architecture
  - UDP port 53, some implementations also use TCP
  - Why?
- Hierarchical namespace
  - As opposed to original, flat namespace
  - e.g. .com → google.com → mail.google.com

Naming Hierarchy



- Top Level Domains (TLDs) are at the top
- Maximum tree depth: 128
- Each Domain Name is a subtree
  - .edu → neu.edu → ccs.neu.edu → www.ccs.neu.edu
- Name collisions are avoided
  - neu.com vs. neu.edu

Hierarchical Administration



- Tree is divided into zones
  - Each zone has an administrator
  - Responsible for the part of the hierarchy
- Example:
  - CCIS controls *.ccs.neu.edu
  - NEU controls *.neu.edu

Server Hierarchy

- Functions of each DNS server:
  - Authority over a portion of the hierarchy
    - No need to store all DNS names
  - Store all the records for hosts/domains in its zone
    - May be replicated for robustness
  - Know the addresses of the root servers
    - Resolve queries for unknown names
- Root servers know about all TLDs
  - The buck stops at the root servers

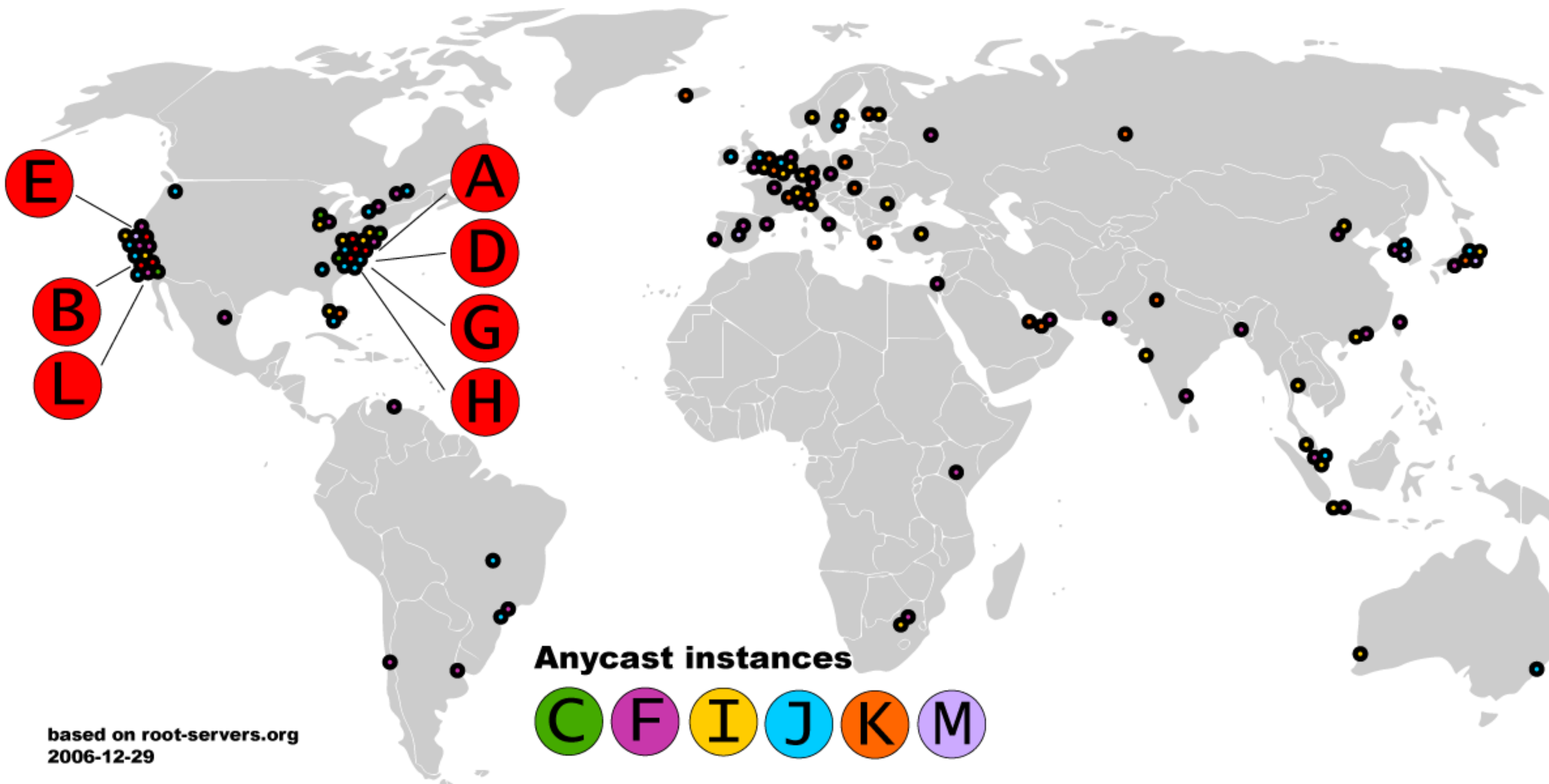
Root Name Servers

- Responsible for the Root Zone File
  - Lists the TLDs and who controls them
  - ~272KB in size

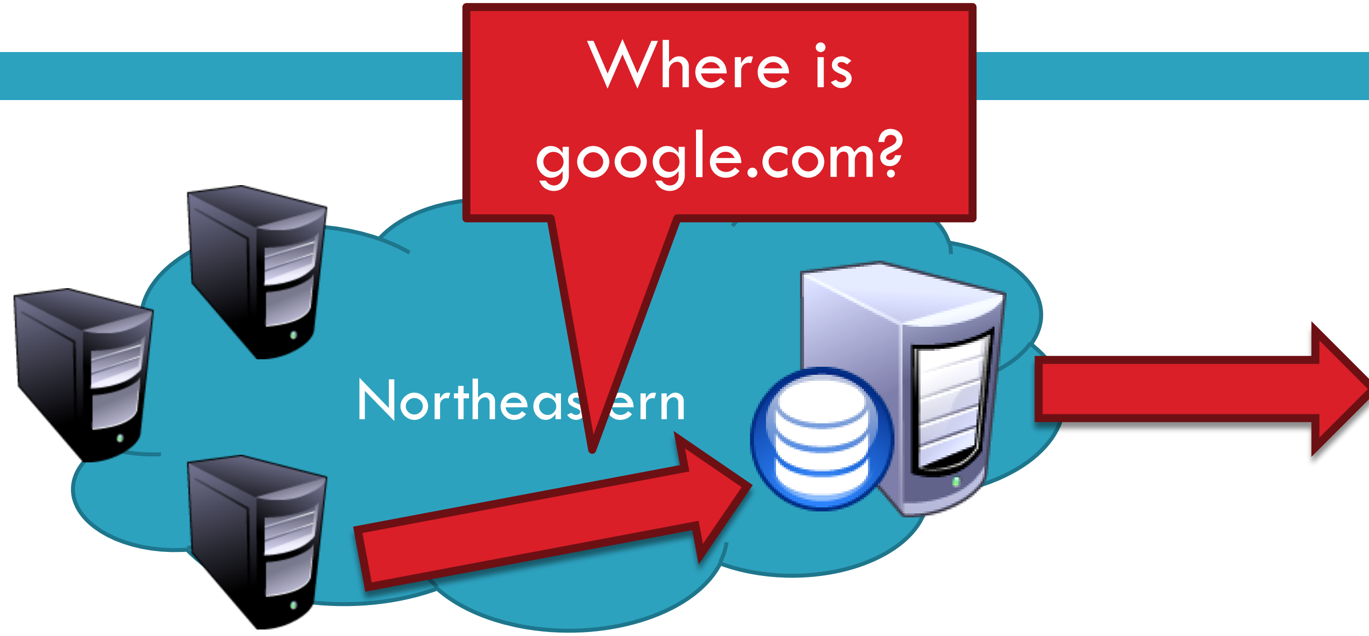
com.	172800	IN	NS	a.gtld-servers.net.
com.	172800	IN	NS	b.gtld-servers.net.
com.	172800	IN	NS	c.gtld-servers.net.

- Administered by ICANN
  - 13 root servers, labeled A→M
  - 6 are anycasted, i.e. they are globally replicated
- Contacted when names cannot be resolved
  - In practice, most systems cache this information

Map of the Roots

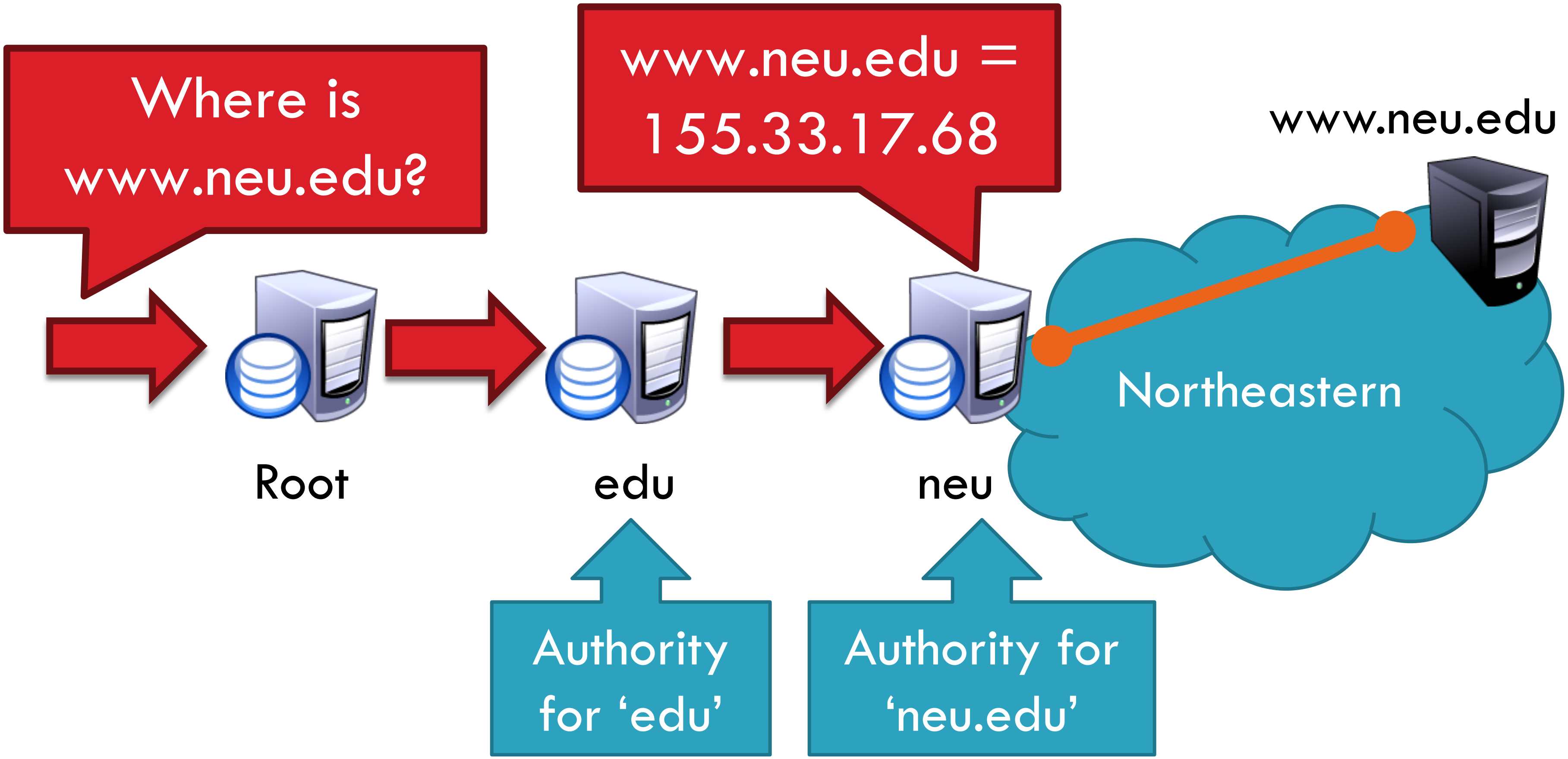


Local Name Servers



- Each ISP/company has a local, default name server
- Often configured via DHCP
- Hosts begin DNS queries by contacting the local name server
- Frequently cache query results

Authoritative Name Servers

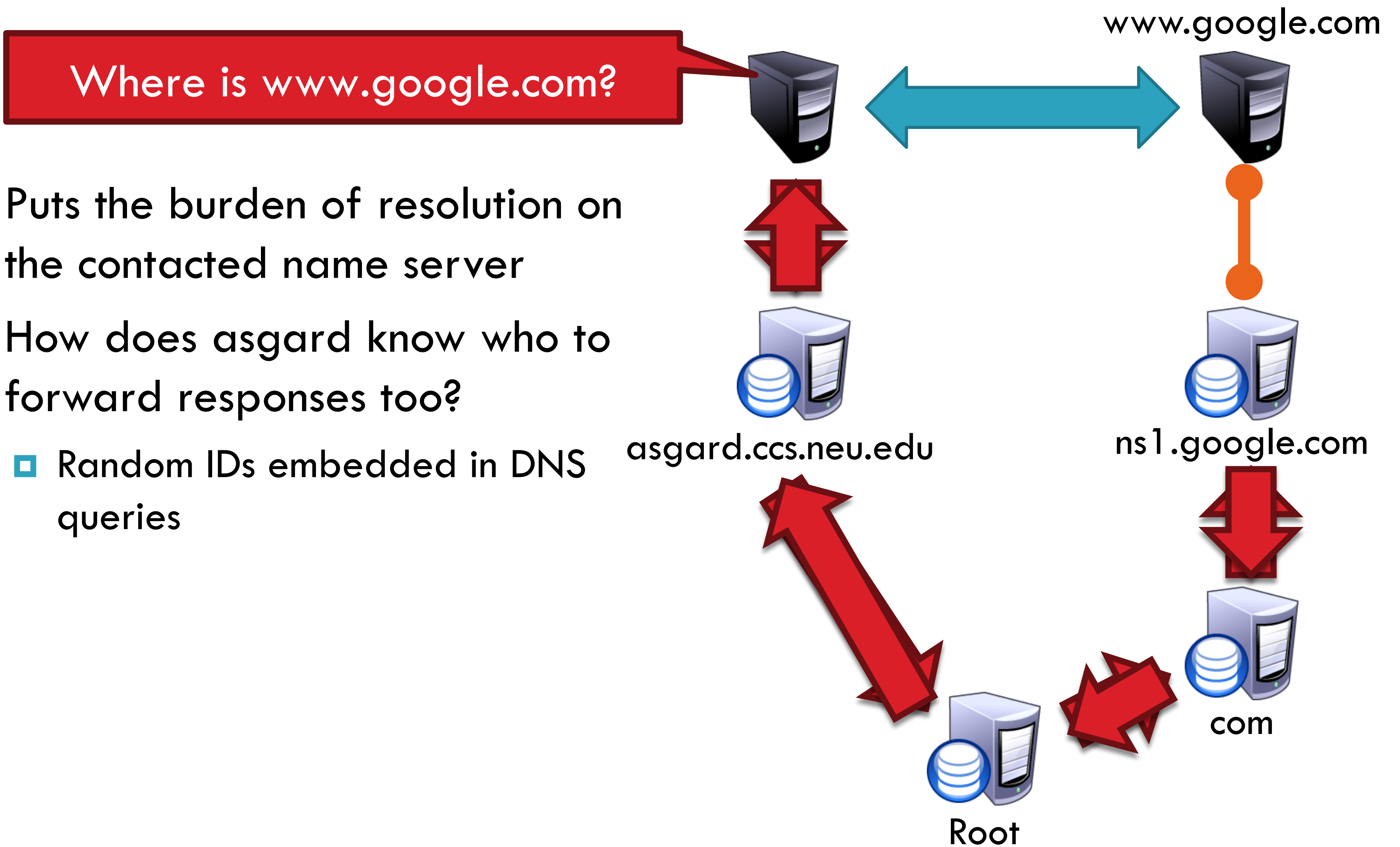


- Stores the name→IP mapping for a given host

Basic Domain Name Resolution

- Every host knows a local DNS server
  - Sends all queries to the local DNS server
- If the local DNS can answer the query, then you're done
  1. Local server is also the authoritative server for that name
  2. Local server has cached the record for that name
- Otherwise, go down the hierarchy and search for the authoritative name server
  - Every local DNS server knows the root servers
  - Use cache to skip steps if possible
    - e.g. skip the root and go directly to .edu if the root file is cached

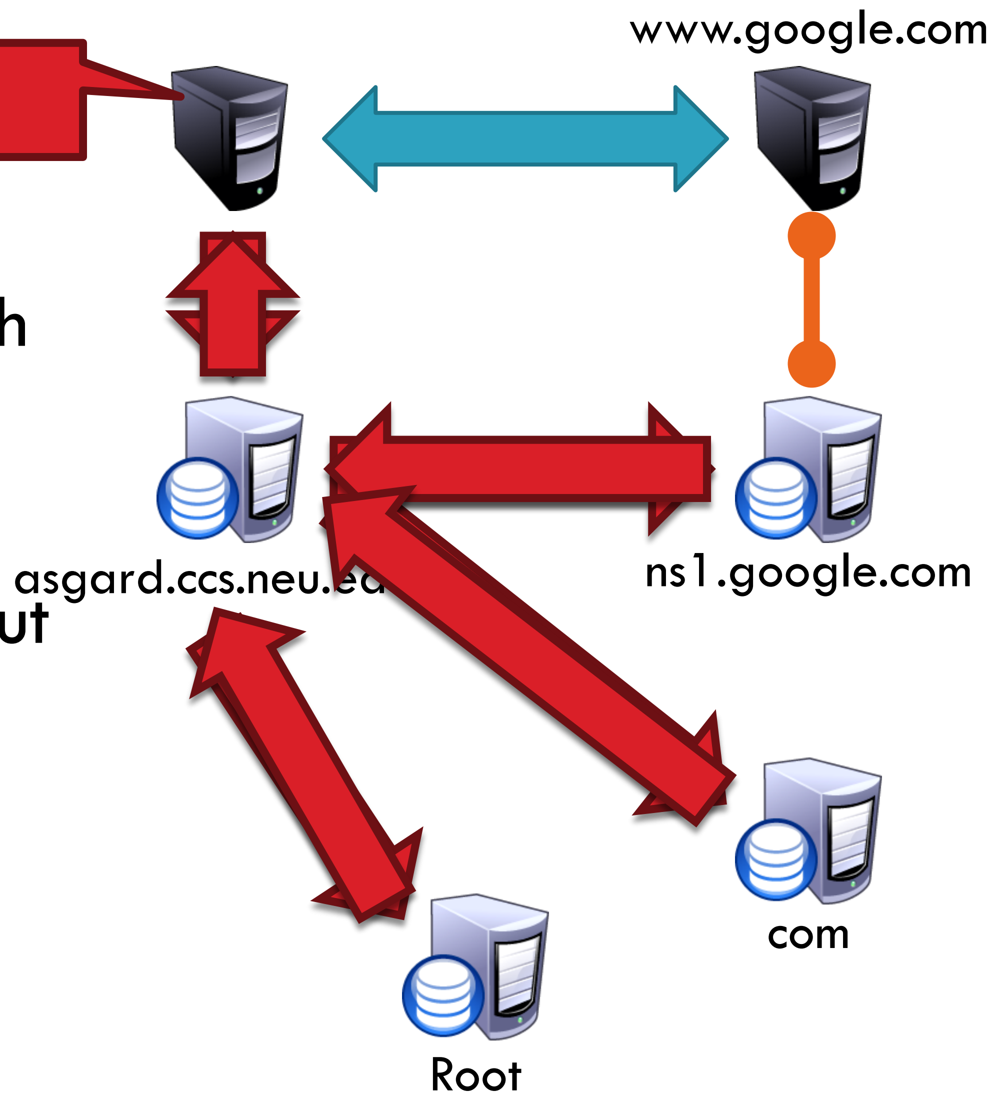
Recursive DNS Query



- Puts the burden of resolution on the contacted name server
- How does asgard know who to forward responses too?
  - Random IDs embedded in DNS queries

Iterated DNS query

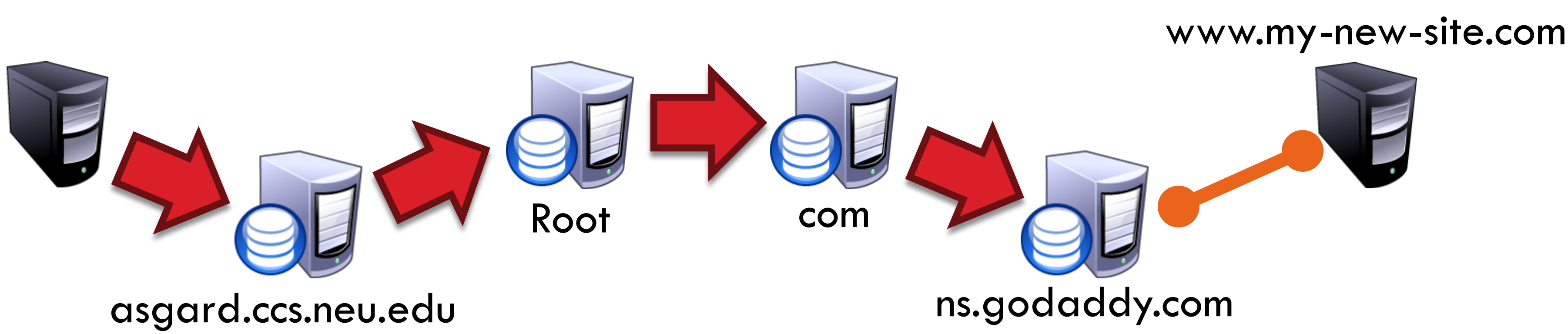
Where is www.google.com?



- Contact server replies with the name of the next authority in the hierarchy
- "I don't know this name, but this other server might"
- This is how DNS works today

DNS Propagation

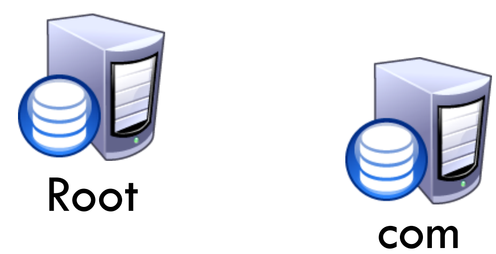
- How many of you have purchased a domain name?
  - Did you notice that it took ~72 hours for your name to become accessible?
  - This delay is called DNS Propagation



- Why would this process fail for a new DNS name?

Caching vs. Freshness

- DNS Propagation delay is caused by caching



- Zone files may be cached for 1-72 hours

DNS Resource Records

- DNS queries have two fields: name and type
- Resource record is the response to a query
  - Four fields: (name, value, type, TTL)
  - There may be multiple records returned for one query
- What do the name and value mean?
  - Depends on the type of query and response

DNS Types

- Type = A / AAAA
  - Name = domain name
  - Value = IP address
  - A is IPv4, AAAA is IPv6

Query	
	Name: www.ccs.neu.edu
	Type: A

Resp.	
	Name: www.ccs.neu.edu
	Value: 129.10.116.81

- Type = NS
  - Name = partial domain
  - Value = name of DNS server for this domain
  - "Go send your query to this other server"

Query	
	Name: ccs.neu.edu
	Type: NS

Resp.	
	Name: ccs.neu.edu
	Value: 129.10.116.51

DNS Types, Continued

- Type = CNAME
  - Name = hostname
  - Value = canonical hostname
  - Useful for aliasing
  - CDNs use this

Query	
	Name: foo.mysite.com
	Type: CNAME

Resp.	
	Name: foo.mysite.com
	Value: bar.mysite.com

- Type = MX
  - Name = domain in email address
  - Value = canonical name of mail server

Query	
	Name: ccs.neu.edu
	Type: MX

Resp.	
	Name: ccs.neu.edu
	Value: amber.ccs.neu.edu

Reverse Lookups

- What about the IP→name mapping?
- Separate server hierarchy stores reverse mappings
  - Rooted at in-addr.arpa and ip6.arpa
- Additional DNS record type: PTR
  - Name = IP address
  - Value = domain name
- Not guaranteed to exist for all IPs

Query
Name: 129.10.116.51
Type: PTR

Resp.
Name: 129.10.116.51
Value: ccs.neu.edu

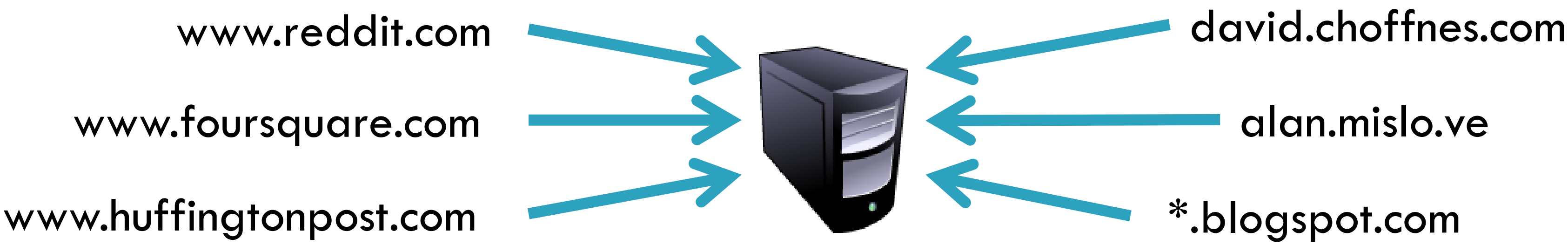
DNS as Indirection Service

- DNS gives us very powerful capabilities
  - Not only easier for humans to reference machines!

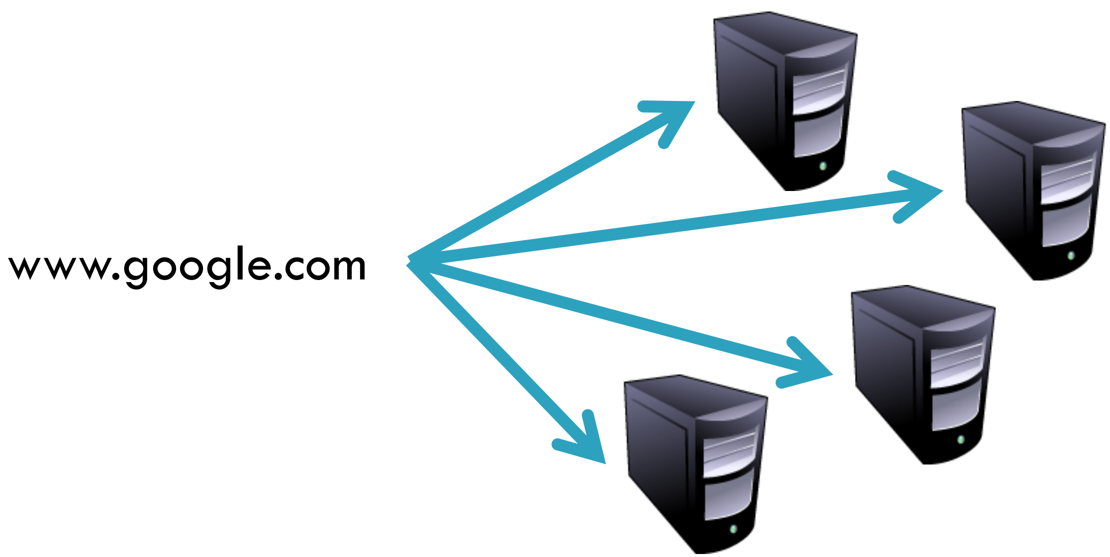
- Changing the IPs of machines becomes trivial
  - e.g. you want to move your web server to a new host
  - Just change the DNS record!

Aliasing and Load Balancing

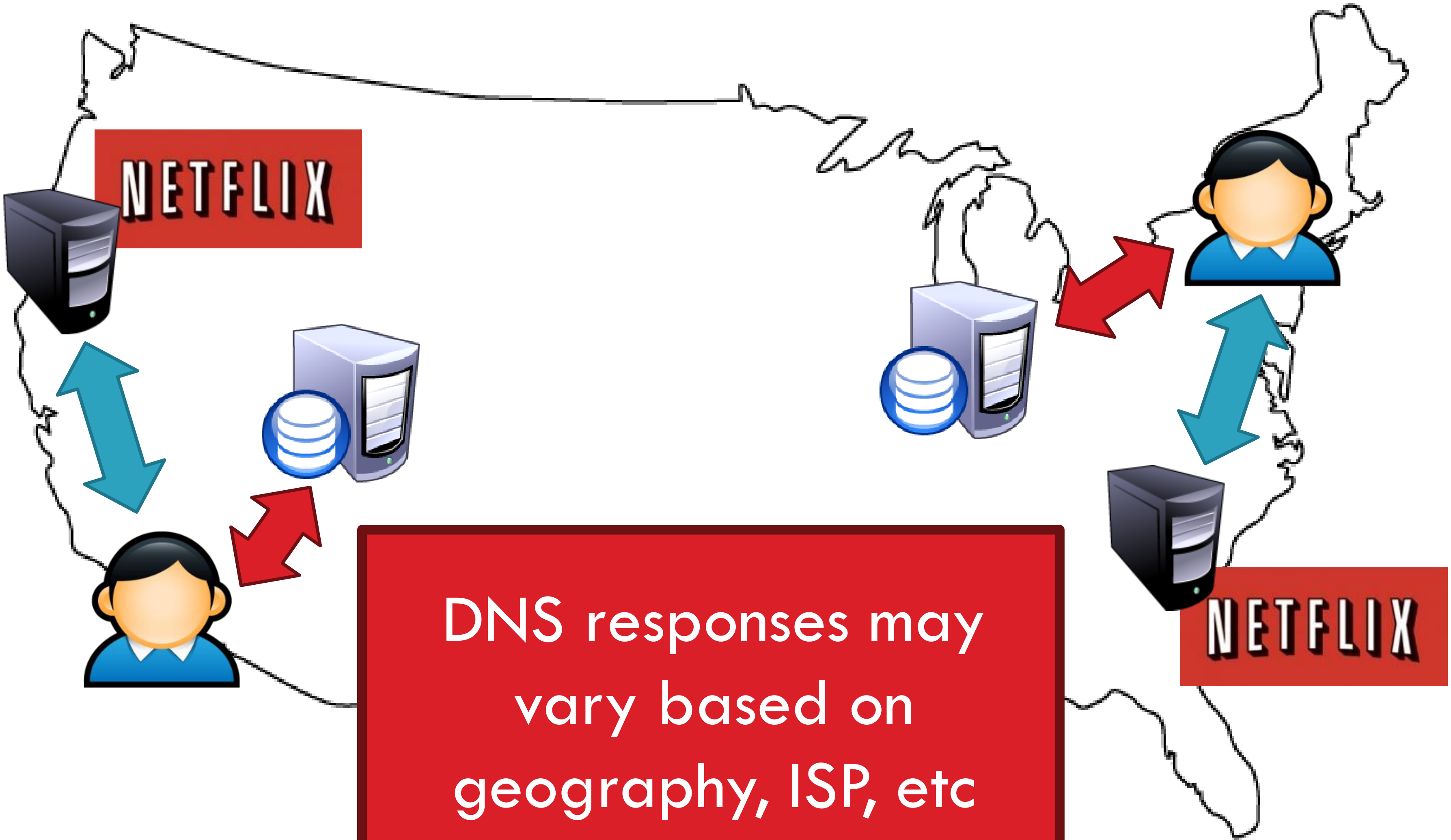
- One machine can have many aliases



- One domain can map to multiple machines



Content Delivery Networks



DNS responses may vary based on geography, ISP, etc

Outline

- HTTP Connection Basics
- HTTP Protocol
- Cookies, keeping state + tracking

Web and HTTP

First, a review…

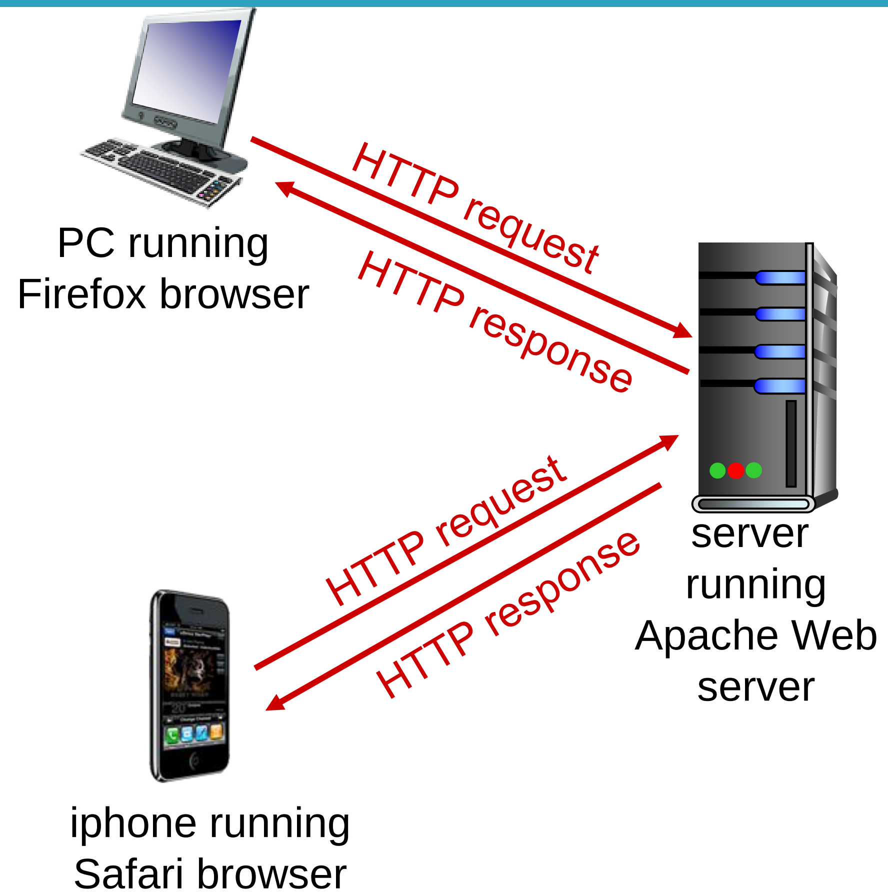
- *web page* consists of *objects*
- object can be HTML file, JPEG image, Java applet, audio file,…
- web page consists of *base HTML-file* which includes *several referenced objects*
- each object is addressable by a *URL,* e.g.,

$$\underbrace{\texttt{www.someschool.edu}}_{\text{host name}}\underbrace{\texttt{/someDept/pic.gif}}_{\text{path name}}$$

HTTP overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - *server:* Web server sends (using HTTP protocol) objects in response to requests



HTTP overview (continued)

uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless" (in theory…)

- server maintains no information about past client requests

aside

protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

HTTP connections

non-persistent HTTP

- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects required multiple connections

persistent HTTP

- multiple objects can be sent over single TCP connection between client, server

Example Web Page

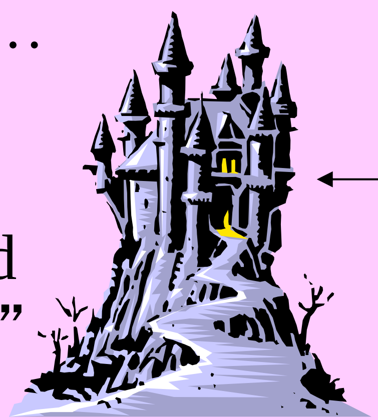
page.html

Harry Potter Movies

As you all know,
the new HP book
will be out in June
and then there will
be a new movie
shortly after that…

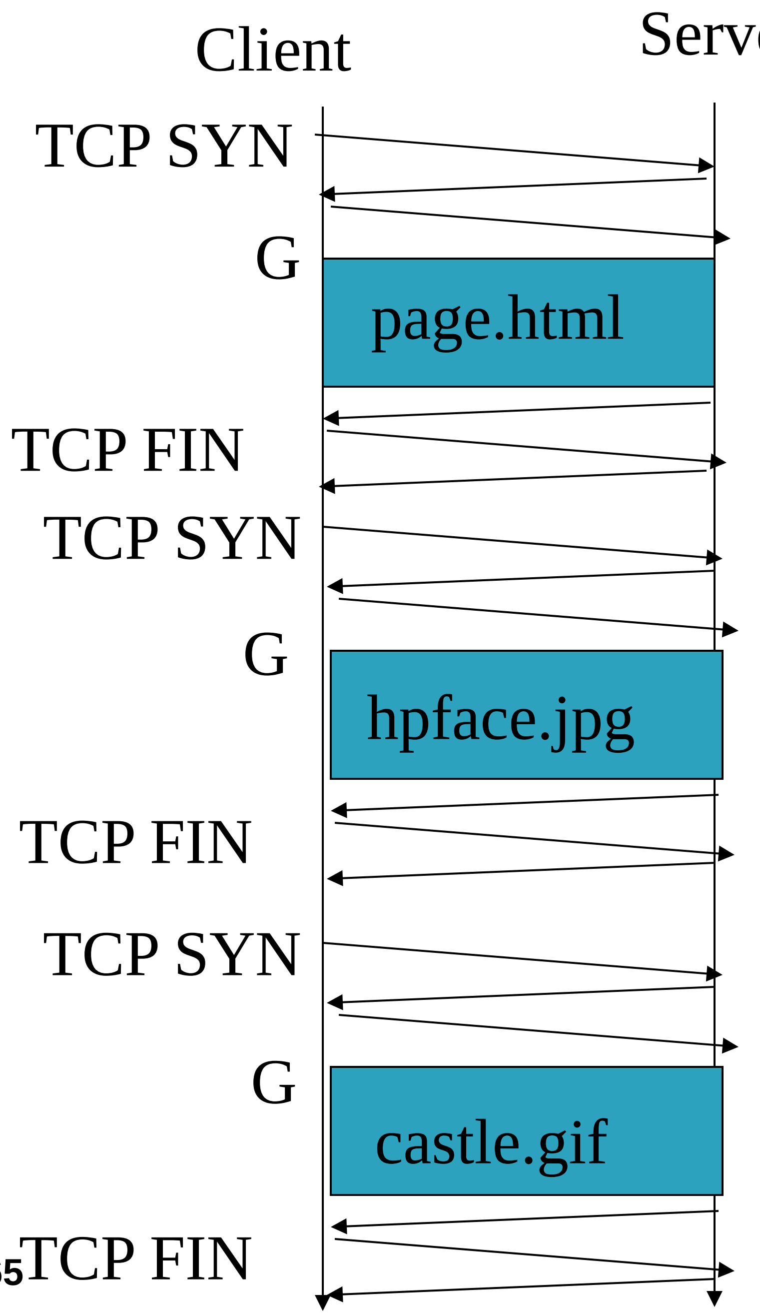
hpface.jpg

“Harry Potter and
the Bathtub Ring”



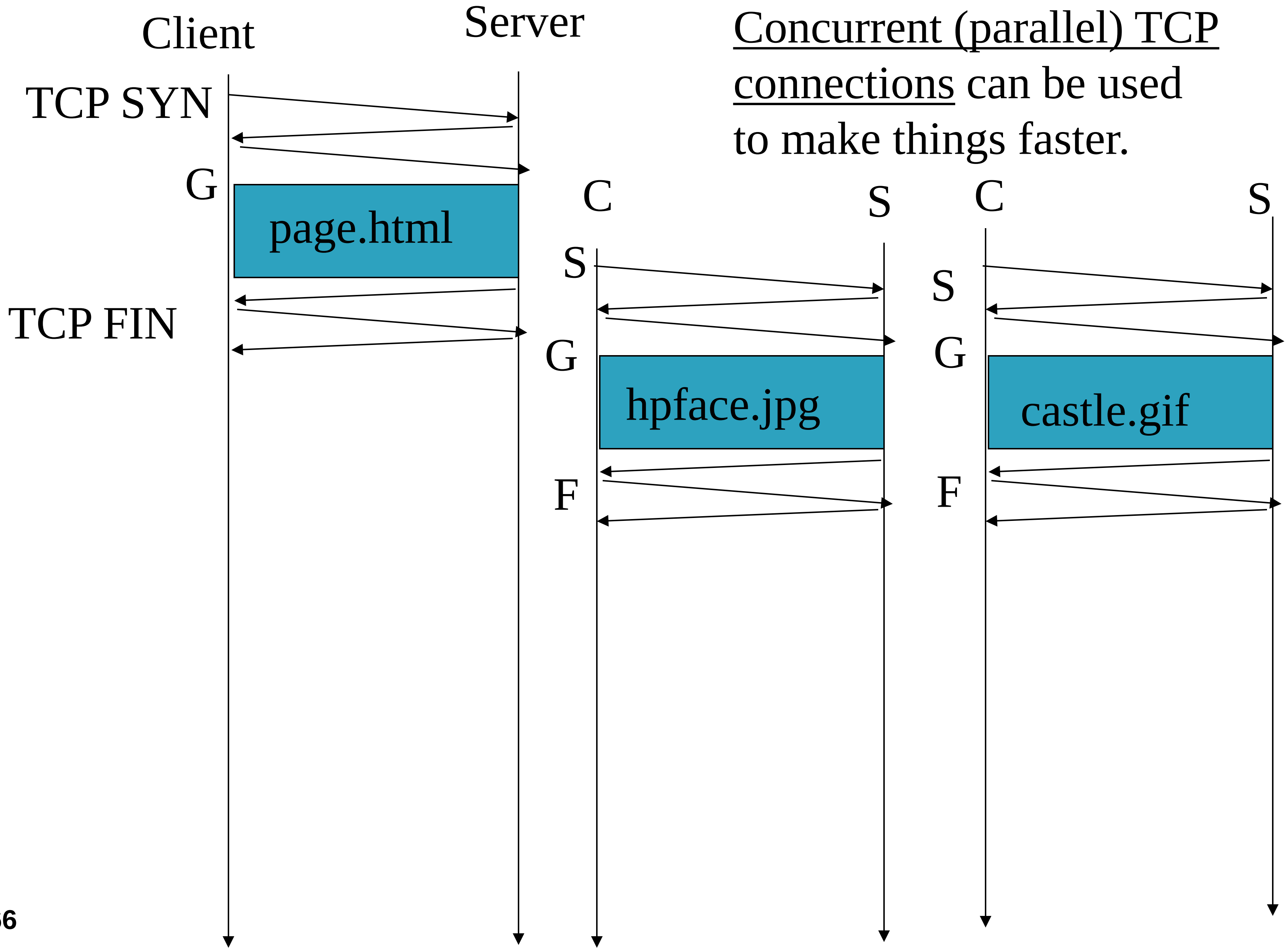
castle.gif

Non-Persistent HTTP



The “classic” approach in HTTP/1.0 is to use one HTTP request per TCP connection, serially.

<u>Concurrent (parallel) TCP connections</u> can be used to make things faster.



Persistent HTTP

non-persistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- 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 sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects

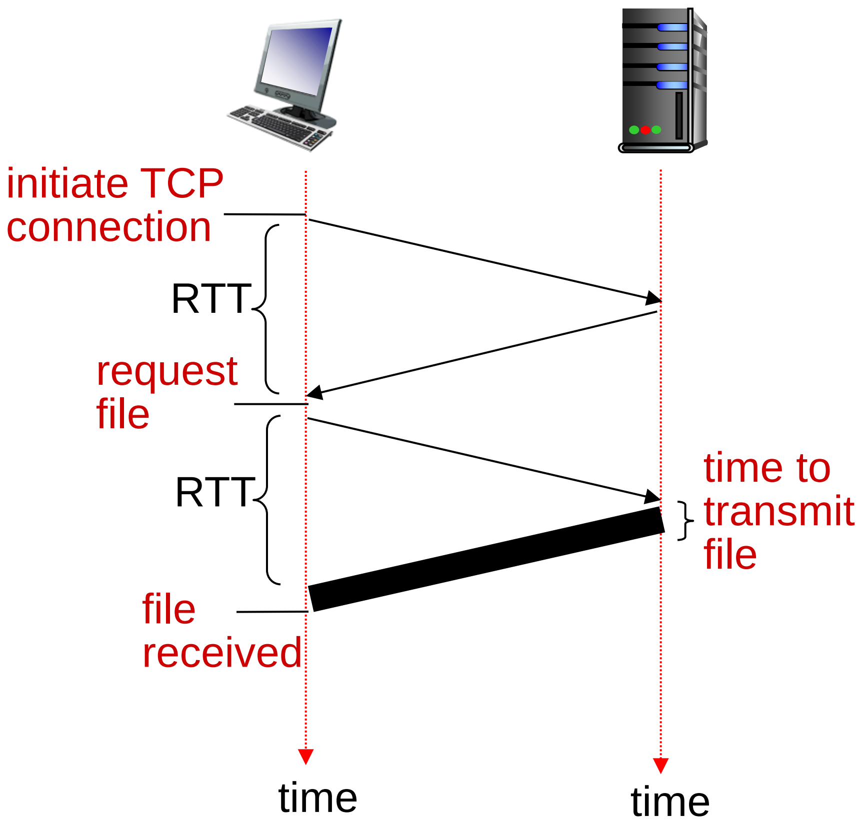
Non-persistent HTTP: response time

RTT: time for a packet to travel from client to server and back

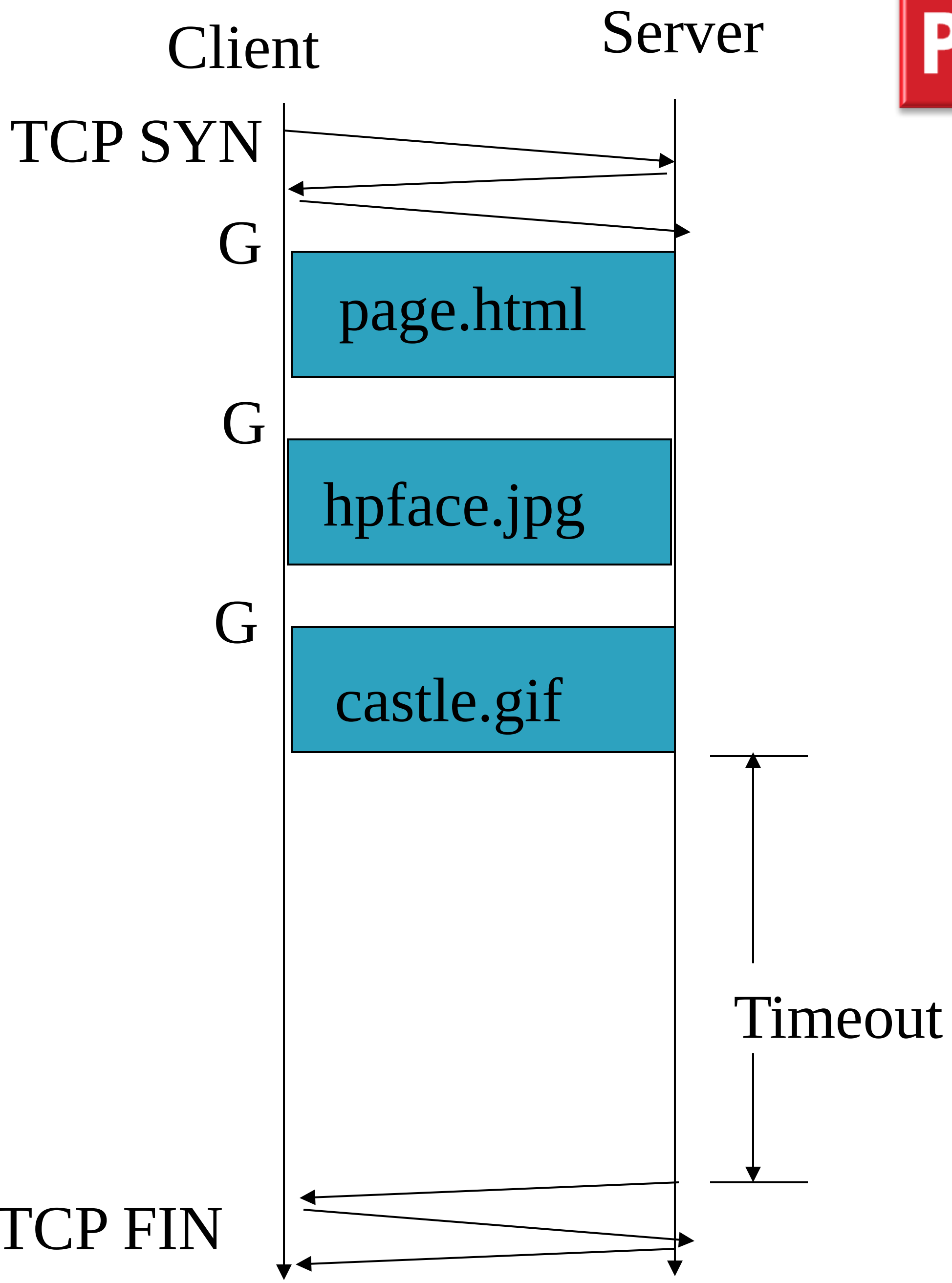
HTTP response time:

- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
  - This assumes HTTP GET piggy backed on the ACK
- file transmission time
- non-persistent HTTP response time =

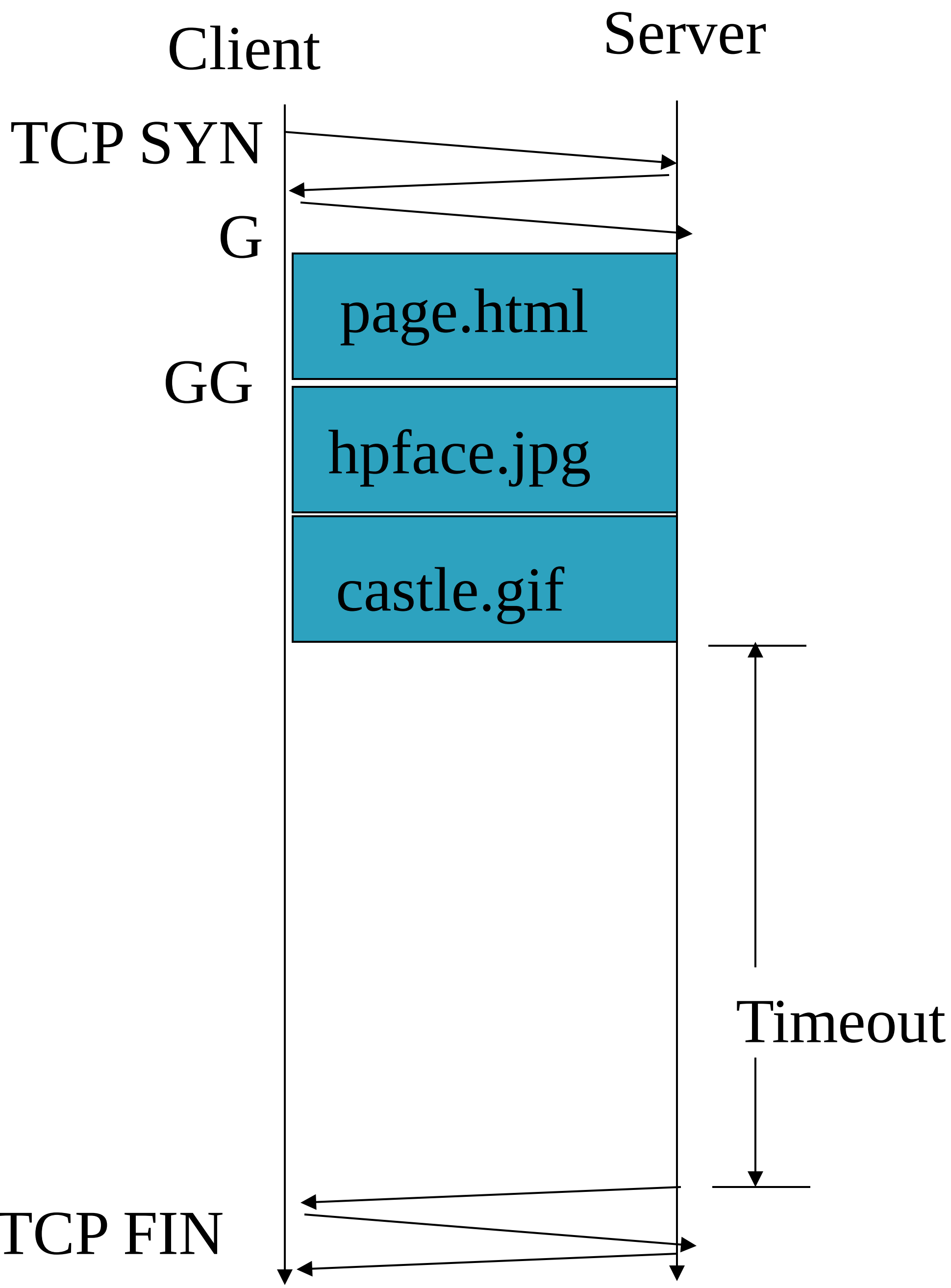
  2RTT+ file transmission time



Persistent HTTP



The "persistent HTTP" approach can re-use the same TCP connection for Multiple HTTP transfers, one after another, serially. Amortizes TCP overhead, but maintains TCP state longer at server.



The "pipelining" feature in HTTP/1.1 allows requests to be issued asynchronously on a persistent connection. Requests must be processed in proper order. Can do clever packaging.

Outline

- HTTP Connection Basics
- HTTP Protocol
- Cookies, keeping state + tracking

HTTP request message

- two types of HTTP messages: *request*, *response*
- HTTP request message:
  - ASCII (human-readable format)

carriage return character

line-feed character

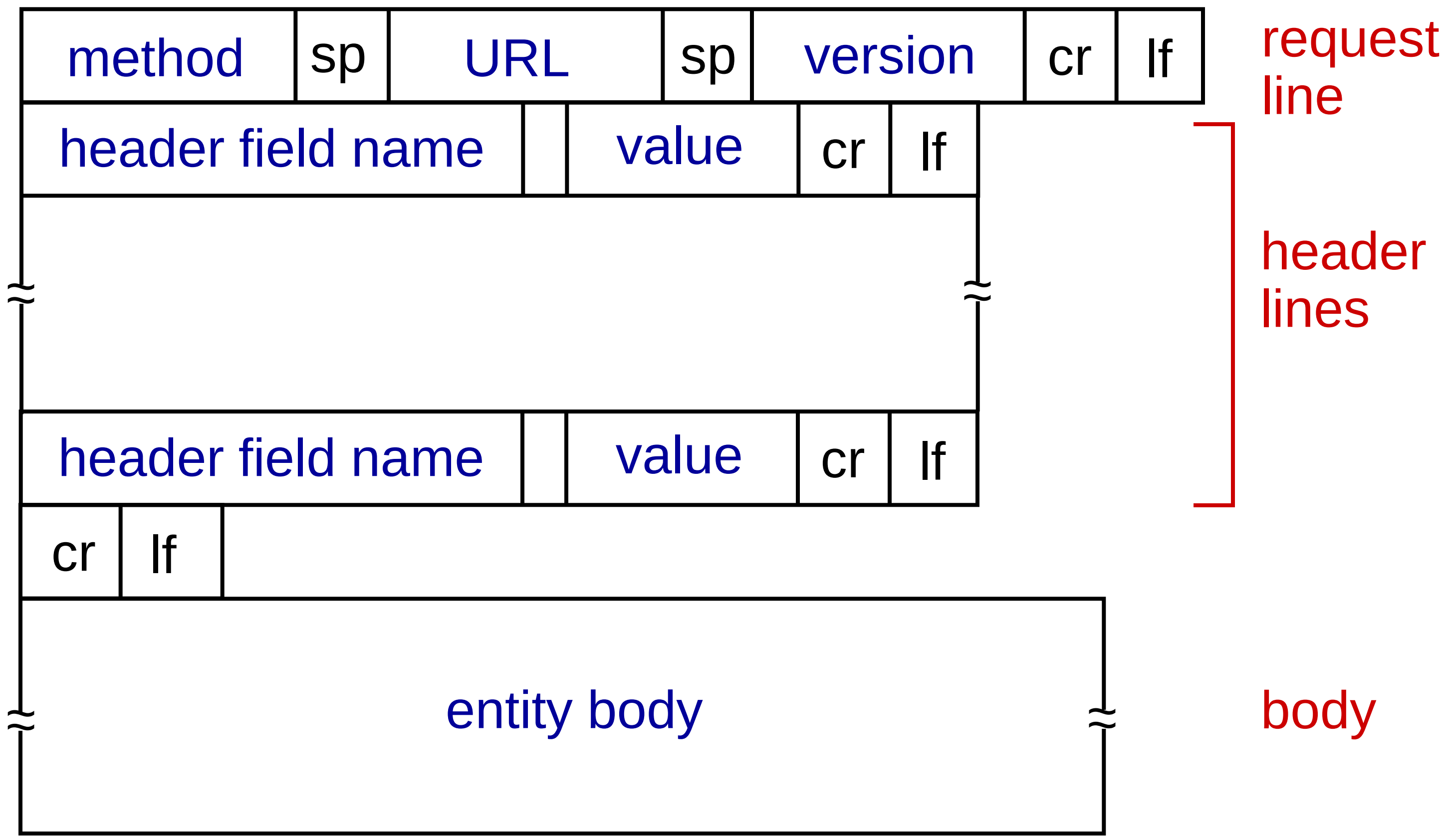
request line
(GET, POST,
HEAD commands)

header
lines

carriage return,
line feed at start
of line indicates
end of header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

HTTP request message: general format



Uploading form input

POST method:

- web page often includes form input
- input is uploaded to server in entity body

URL method:

- uses GET method
- input is uploaded in URL field of request line:

```
www.somesite.com/animalsearch?monkeys&banana
```

Method types

HTTP/1.0:

- GET
- POST
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1:

- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field

HTTP response message

status line (protocol status code status phrase)

header lines

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
   GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-
   1\r\n
\r\n
data data data data data ...
```

data, e.g., requested HTML file

HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:

`200 OK`

- request succeeded, requested object later in this msg

`301 Moved Permanently`

- requested object moved, new location specified later in this msg (Location:)

`400 Bad Request`

- request msg not understood by server

`404 Not Found`

- requested document not found on this server

`505 HTTP Version Not Supported`

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet cis.poly.edu 80
```

opens TCP connection to port 80 (default HTTP server port) at cis.poly.edu. anything typed in sent to port 80 at cis.poly.edu

2. type in a GET HTTP request:

```
GET /~ross/ HTTP/1.1
Host: cis.poly.edu
```

by typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

3. look at response message sent by HTTP server!

(or use Wireshark to look at captured HTTP request/response)

Outline

- HTTP Connection Basics
- HTTP Protocol
- Cookies, keeping state + tracking

User-server state: cookies

many Web sites use cookies

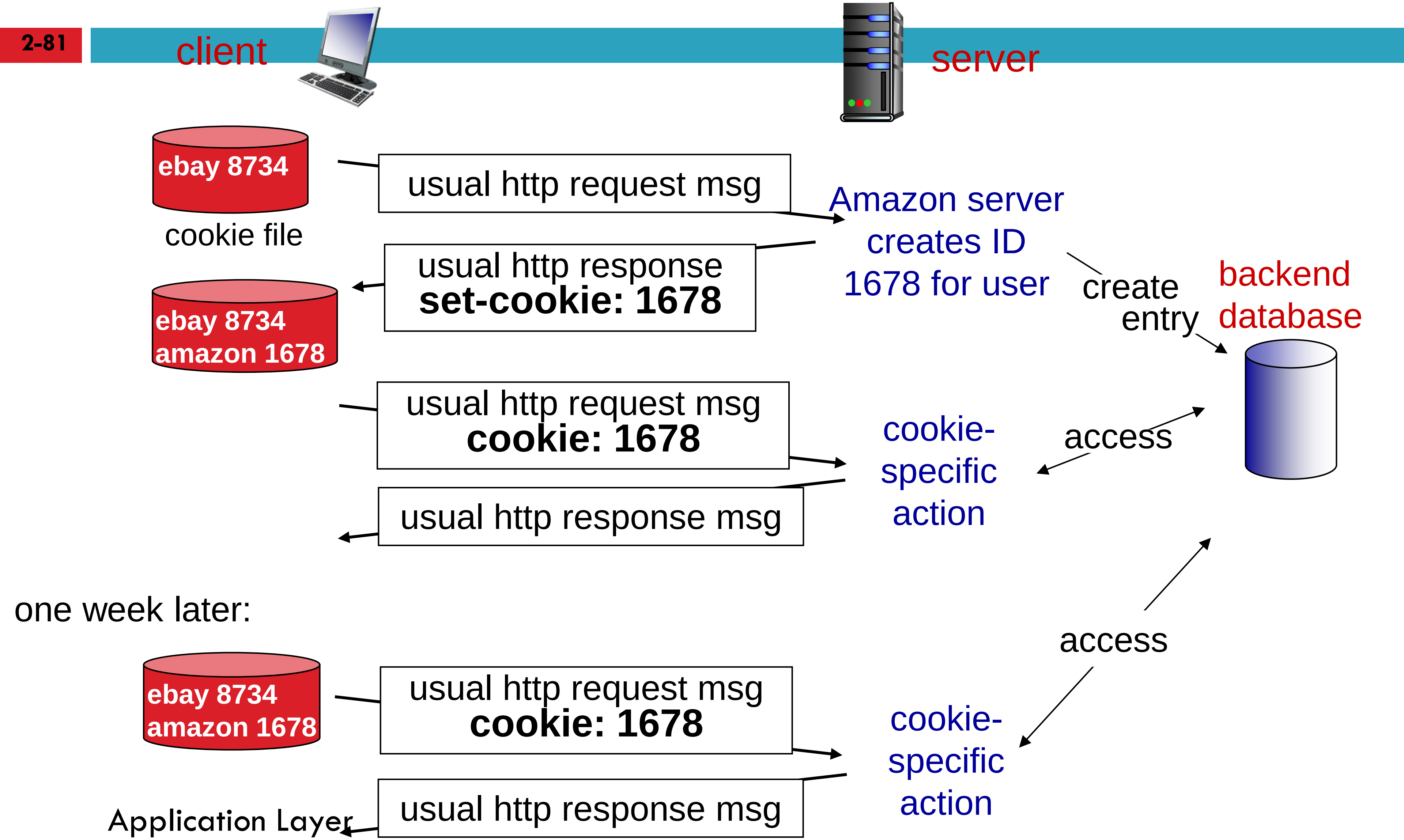
four components:

1) cookie header line of HTTP *response* message
2) cookie header line in next HTTP *request* message
3) cookie file kept on user's host, managed by user's browser
4) back-end database at Web site

example:

- Susan always access Internet from PC
- visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID

Cookies: keeping "state" (cont.)



Cookies (continued)

what cookies can be used for:

- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

aside

cookies and privacy:

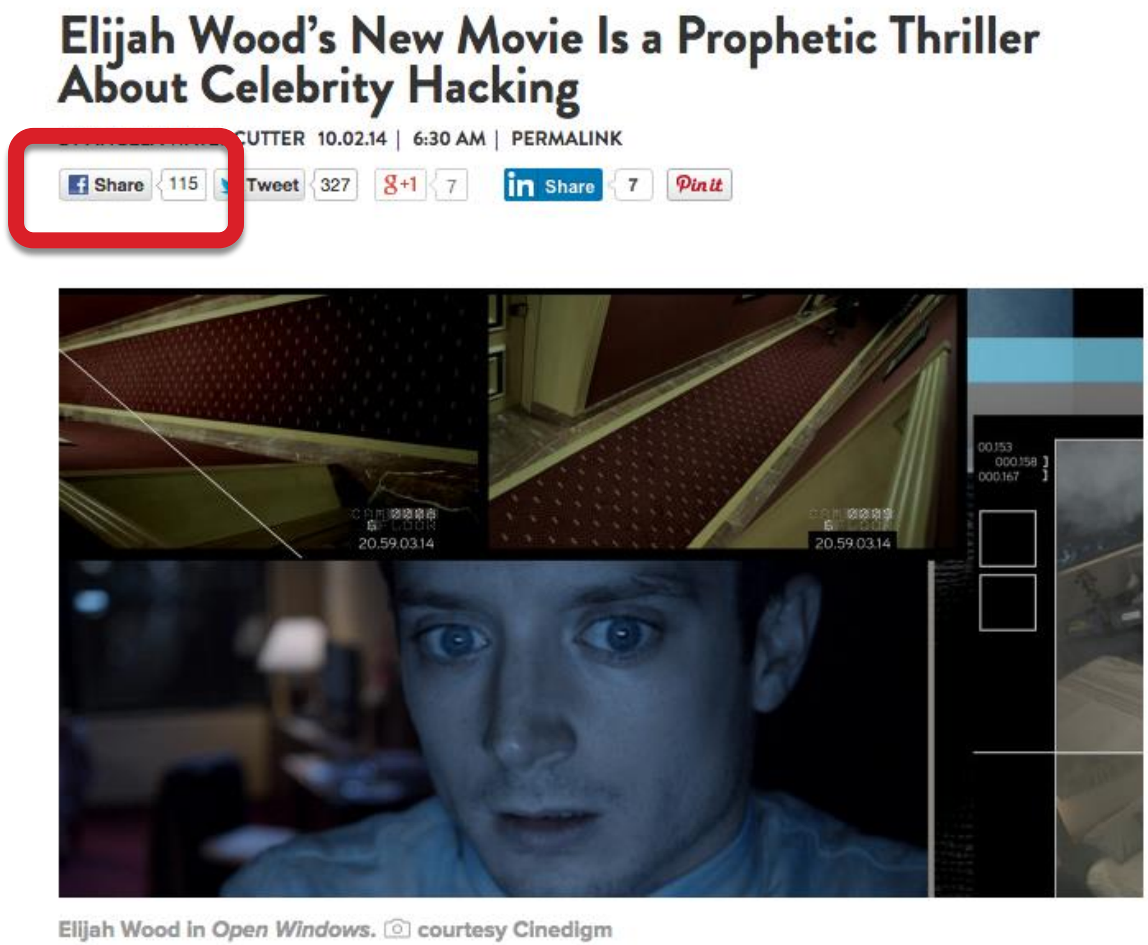
- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites

how to keep "state":

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

Cookies + Third Parties

- Example page (from Wired.com)



How it works

And it's not just Facebook!

Facebook now knows you visited this Wired article.
Works for all pages where 'like'/'share' button is embedded!