

Application Layer and Socket Programming

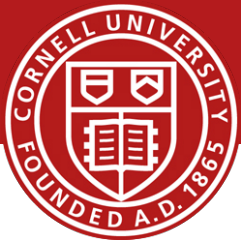
Hakim Weatherspoon

Associate Professor, Dept of Computer Science

CS 5413: High Performance Systems and Networking

February 3, 2017

Administrivia



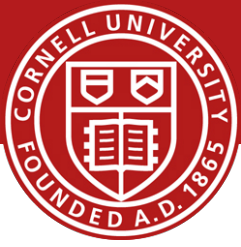
- Project Teams: due by Monday
- Lab1 due today

- Next week
- Lab2 done in class next Friday (Feb 10)
- HW1: Single threaded TCP proxy handed out next week
 - Done in groups
- Project Teams due Monday (Feb 6),
Project Ideas due following Monday (Feb 13)

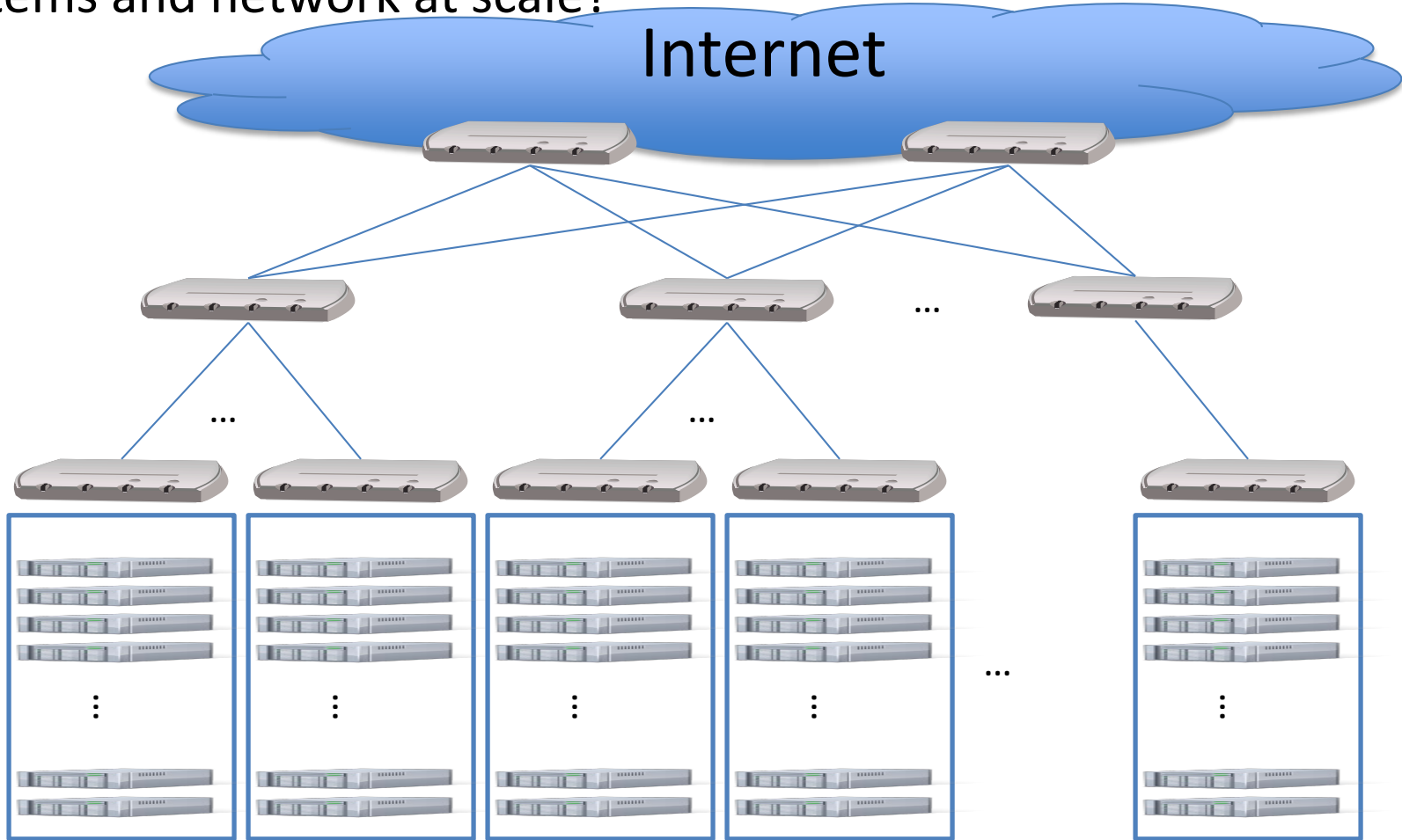
- Review chapter 3 from the book, Transport Layer
 - We will also briefly discuss data center topologies

- Check website for updated schedule

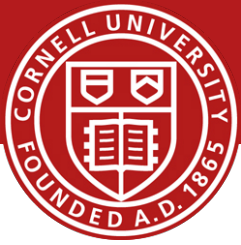
Big Picture



How do we understand and optimize the (datacenter) systems and network at scale?

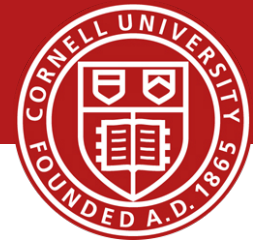


Goals for Today



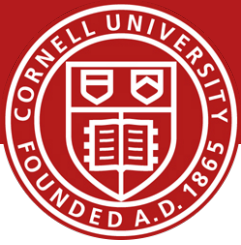
- Application Layer
 - Example network applications
 - conceptual, implementation aspects of network application protocols
 - client-server paradigm
 - transport-layer service models
- Socket Programming
 - Client-Server Example
- Backup Slides
 - Web Caching
 - DNS (Domain Name System)

Some network apps



- e-mail
- web
- text messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- voice over IP (e.g., Skype)
- real-time video conferencing
- social networking
- search
- ...
- ...

Creating a network app

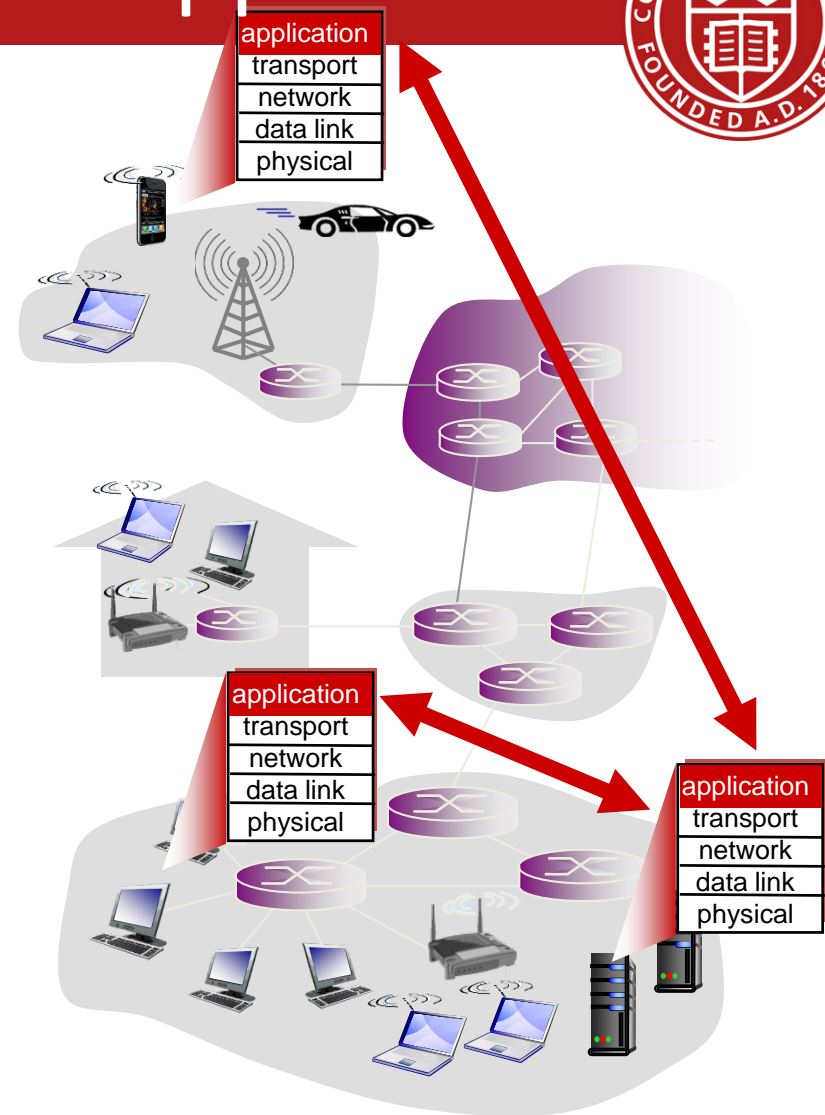


write programs that:

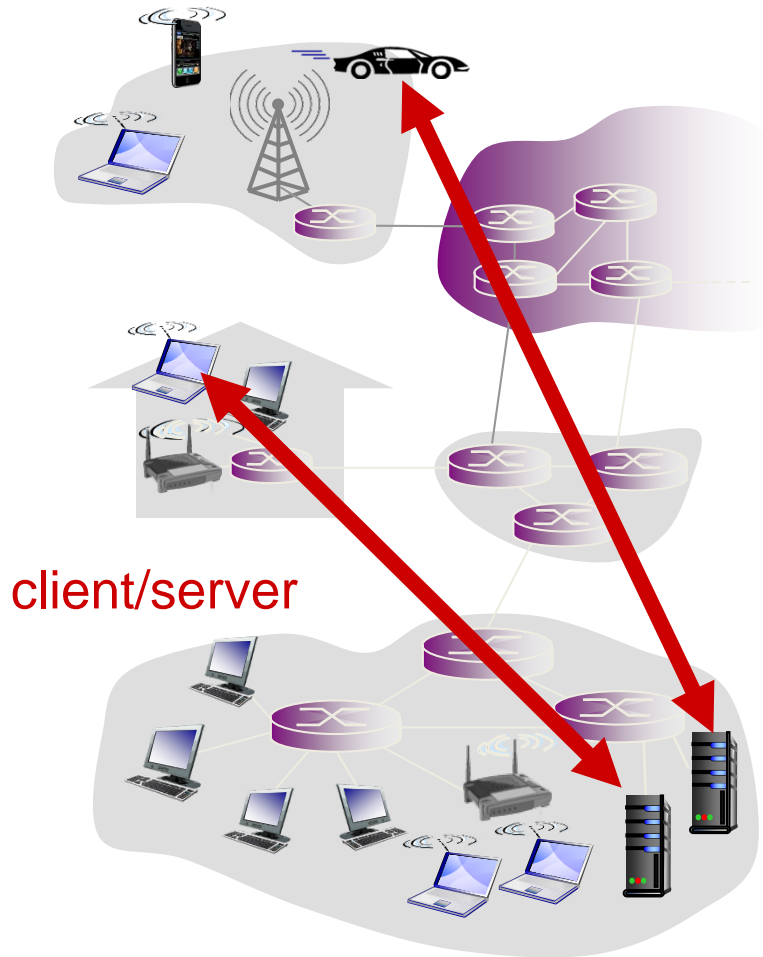
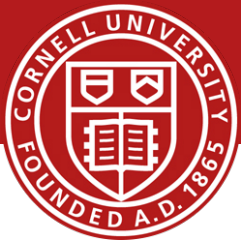
- run on (different) *end systems*
- communicate over network
- e.g., web server software communicates with browser software

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



Client-Server Architecture



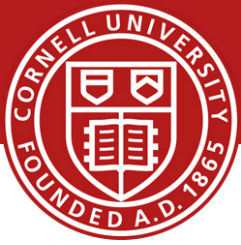
server:

- always-on host
- permanent IP address
- *data centers for scaling*

clients:

- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

Network Applications



Communicating Processes

process: program running within a host

- within same host, two processes communicate using **inter-process communication** (defined by OS)
- processes in different hosts communicate by exchanging **messages**

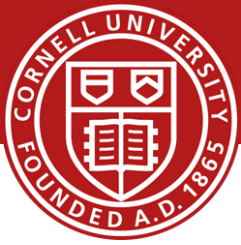
clients, servers

client process: process that initiates communication

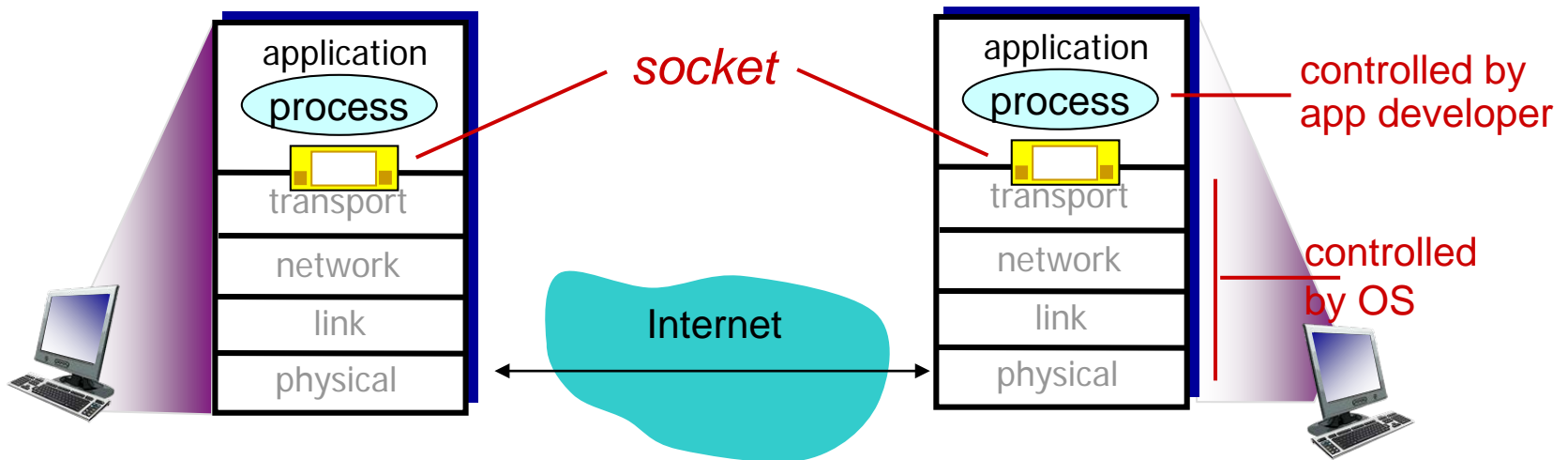
server process: process that waits to be contacted

- ❖ aside: applications with P2P architectures have client processes & server processes

Network Applications



- 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 to deliver message to socket at receiving process



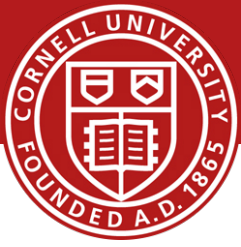
Network Applications



How to identify network applications?

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- Q: does IP address of host on which process runs suffice for identifying the process?
- A: no, *many* processes can be running on same host
- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
 - HTTP server: 80
 - mail server: 25
- to send HTTP message to `www.cs.cornell.edu` web server:
 - IP address: 132.236.207.20p
 - port number: 80

Network Applications



App-Layer protocols define:

- types of messages exchanged,
 - e.g., request, response
- message syntax:
 - what fields in messages & how fields are delineated
- message semantics
 - meaning of information in fields
- rules for when and how processes send & respond to messages

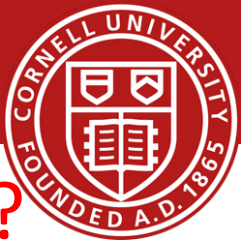
open protocols:

- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype

Network Applications



What transport layer services does an app need?

data integrity

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”

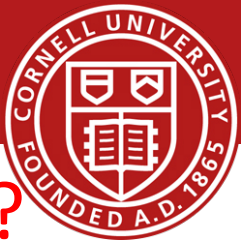
throughput

- ❖ some apps (e.g., multimedia) require minimum amount of throughput to be “effective”
- ❖ other apps (“elastic apps”) make use of whatever throughput they get

security

- ❖ encryption, data integrity,
...

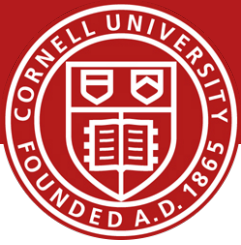
Network Applications



What transport layer services does an app need?

application	data loss	throughput	time sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video: 10kbps-5Mbps	yes, 100' s msec
stored audio/video	loss-tolerant	same as above	
interactive games	loss-tolerant	few kbps up	yes, few secs
text messaging	no loss	elastic	yes, 100' s msec yes and no

Network Applications



Transport Protocol Services

TCP service:

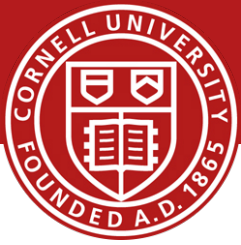
- *reliable transport* between sending and receiving process
- *flow control*: sender won't overwhelm receiver
- *congestion control*: throttle sender when network overloaded
- *does not provide*: timing, minimum throughput guarantee, security
- *connection-oriented*: setup required between client and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

UDP service:

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

Q: why bother? Why is there a UDP?

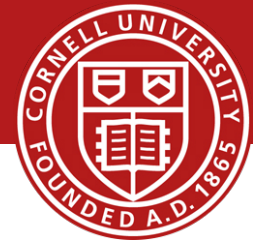
Network Applications



Transport Protocol Services

application	application layer protocol	underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	HTTP (e.g., YouTube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP

Network Applications: Securing TCP



TCP & UDP

- no encryption
- cleartext passwds sent into socket traverse Internet in cleartext

SSL

- provides encrypted TCP connection
- data integrity
- end-point authentication

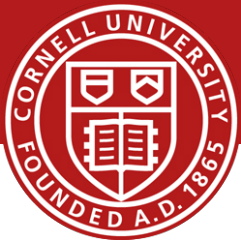
SSL is at app layer

- Apps use SSL libraries, which “talk” to TCP

SSL socket API

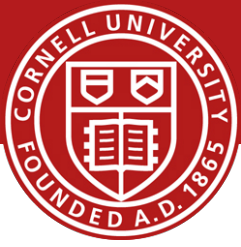
- ❖ cleartext passwds sent into socket traverse Internet encrypted
- ❖ See Chapter 7

Goals for Today



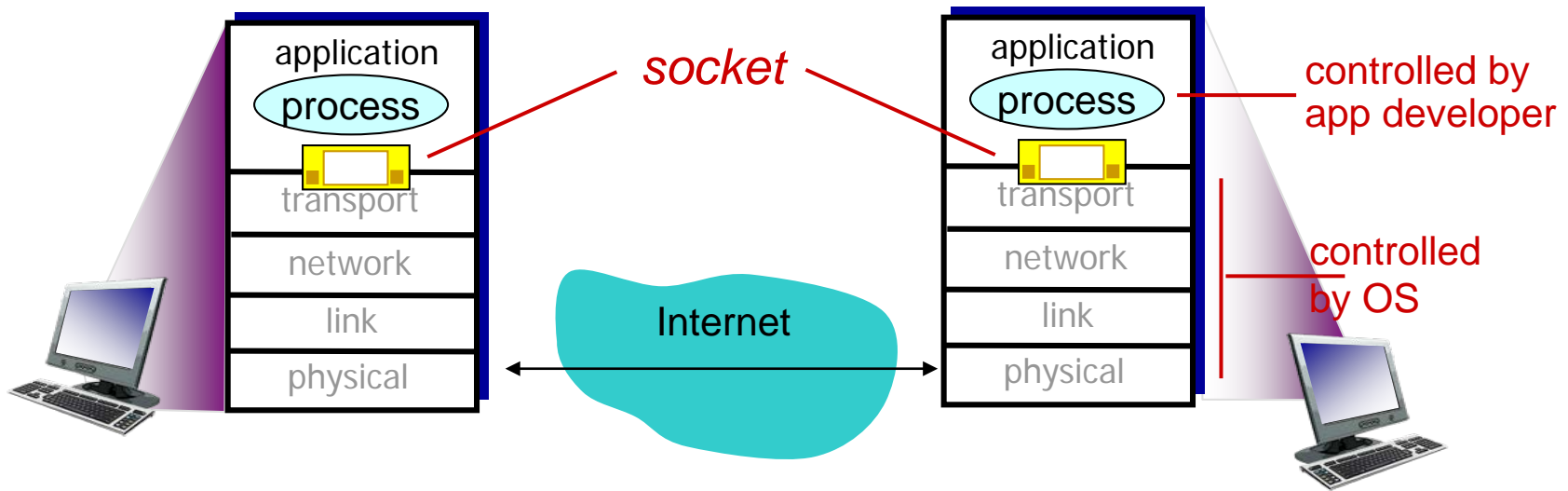
- Application Layer
 - Example network applications
 - conceptual, implementation aspects of network application protocols
 - client-server paradigm
 - transport-layer service models
- Socket Programming
 - Client-Server Example
- Backup Slides
 - Web Caching
 - DNS (Domain Name System)

Socket Programming

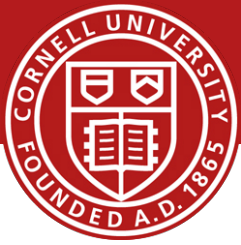


goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Socket Programming



Two socket types for two transport services:

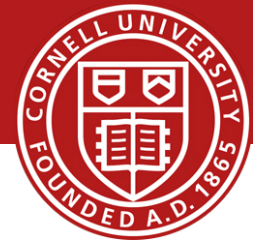
- **UDP**: unreliable datagram
- **TCP**: reliable, byte stream-oriented

Application Example:

1. Client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

More socket examples at <https://docs.python.org/2/library/socket.html>

Socket Programming w/ UDP



UDP: no “connection” between client & server

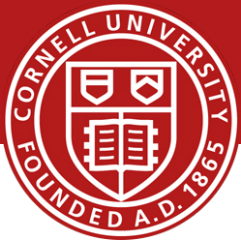
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- rcvr extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes (“datagrams”) between client and server

Socket Programming w/ UDP



server (running on *serverIP*)

create socket, port= x:
`serverSocket =
socket(AF_INET,SOCK_DGRAM)`

↓
read datagram from
`serverSocket`

↓
write reply to
`serverSocket`
specifying
client address,
port number

client

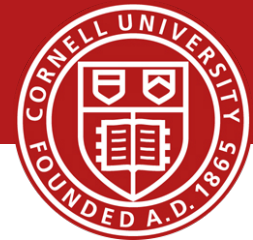
create socket:
`clientSocket =
socket(AF_INET,SOCK_DGRAM)`

↓
Create datagram with server IP and
port=x; send datagram via
`clientSocket`

↓
read datagram from
`clientSocket`

↓
close
`clientSocket`

Socket Programming w/ UDP



Python UDPClient

include Python's socket library

```
from socket import *  
serverName = 'localhost'  
serverPort = 12000
```

create UDP socket for server

```
clientSocket = socket(AF_INET, SOCK_DGRAM)
```

get user keyboard input

```
message = raw_input('Input lowercase sentence:')
```

Attach server name, port to message; send into socket

```
clientSocket.sendto(message, (serverName, serverPort))
```

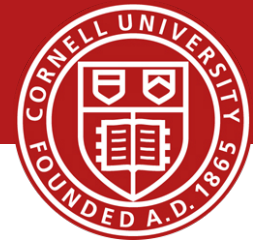
read reply characters from socket into string

```
modifiedMessage, serverAddress =  
clientSocket.recvfrom(2048)
```

print out received string and close socket

```
print modifiedMessage  
clientSocket.close()
```

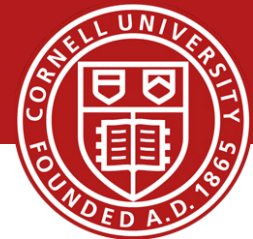
Socket Programming w/ UDP



Python UDP Server

```
from socket import *
serverPort = 12000
create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)
bind socket to local port → serverSocket.bind(('', serverPort))
number 12000
print 'The server is ready to receive'
loop forever → while 1:
Read from UDP socket into → message, clientAddress = serverSocket.recvfrom(2048)
message, getting client's → modifiedMessage = message.upper()
address (client IP and port)
send upper case string → serverSocket.sendto(modifiedMessage, clientAddress)
back to this client
```

Socket Programming w/ TCP



client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

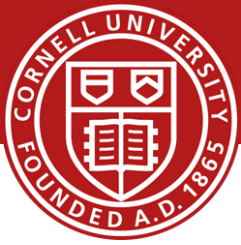
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
 - allows server to talk with multiple clients
 - source port numbers used to distinguish clients (more in Chap 3)

application viewpoint:

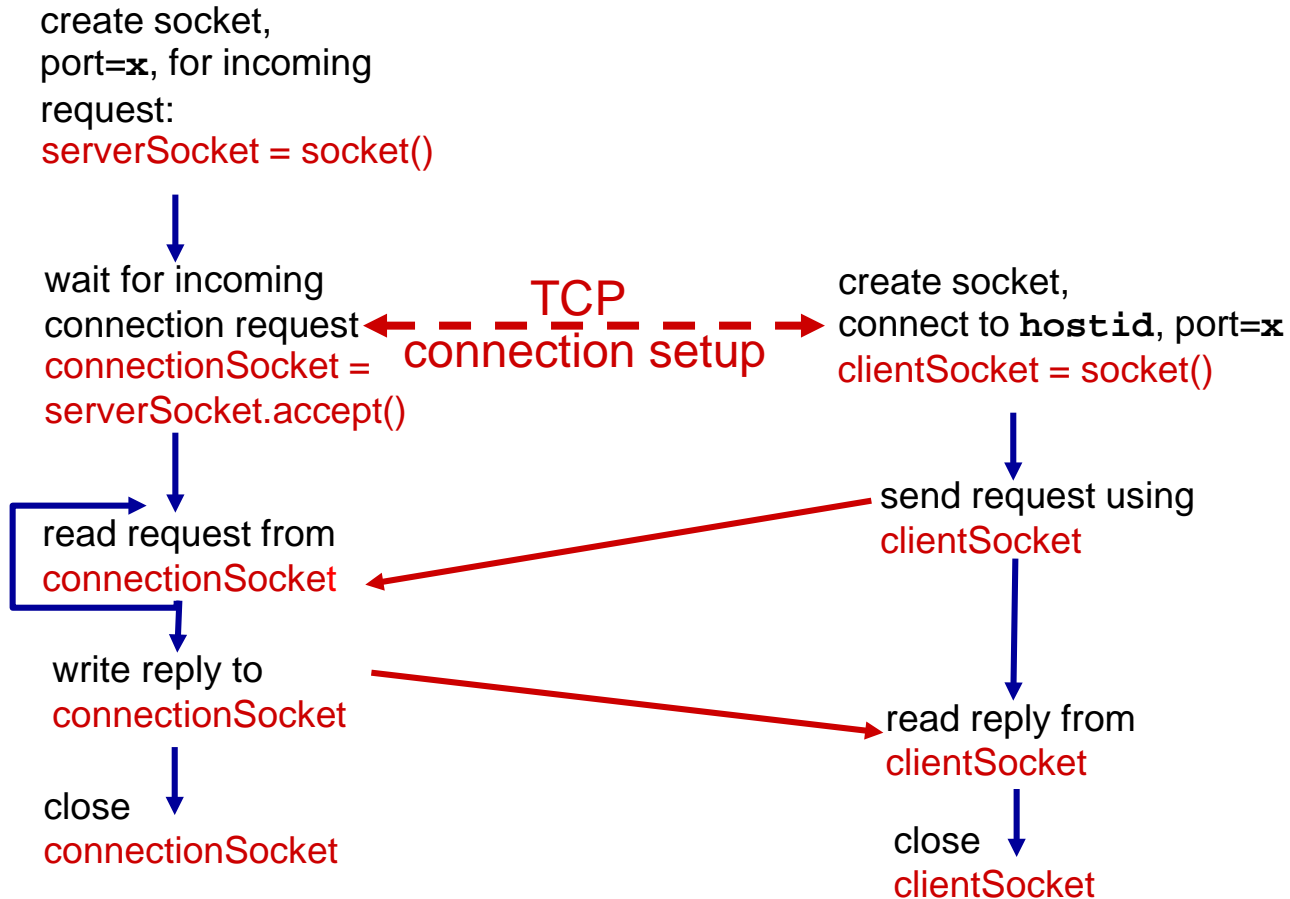
TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server

Socket Programming w/ TCP

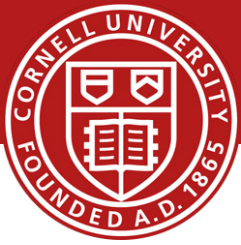


server (running on `hostid`)

client



Socket Programming w/ TCP



Python TCPClient

```
from socket import *
serverName = 'localhost'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

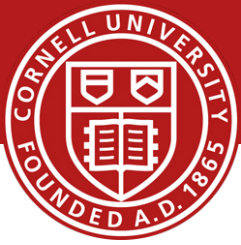
create TCP socket for
server, remote port 12000



No need to attach server
name, port



Socket Programming w/ TCP



Python TCP Server

create TCP welcoming
socket



server begins listening for
incoming TCP requests



loop forever



server waits on accept()
for incoming requests, new
socket created on return



read bytes from socket (but
not address as in UDP)

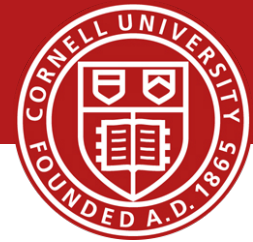


close connection to this
client (but *not* welcoming
socket)



```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while 1:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

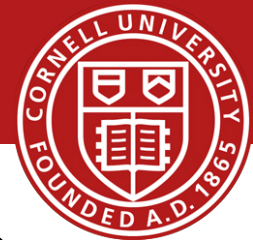
Perspective



- application architectures
 - client-server
 - P2P
- application service requirements:
 - reliability, bandwidth, delay
- Internet transport service model
 - connection-oriented, reliable: TCP
 - unreliable, datagrams: UDP
- ❖ specific protocols:
 - HTTP
 - FTP
 - SMTP, POP, IMAP
 - DNS
 - P2P: BitTorrent, DHT
- ❖ socket programming: TCP, UDP sockets

Application Layer is the same in a data center!

Before Next time

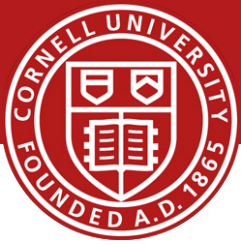


- Project Group: Make sure that you are part of one
- Finish Lab0

- No required reading and review due
- But, review chapter 3 from the book, Transport Layer
 - We will also briefly discuss
 - *Data center TCP (DCTCP)*, Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. ACM SIGCOMM Computer Communications Review, Volume 40, Issue 4 (August 2010), pages 63-74.

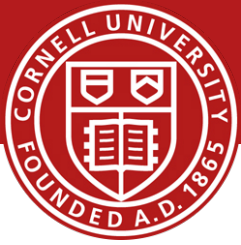
- Check website for updated schedule

Goals for Today



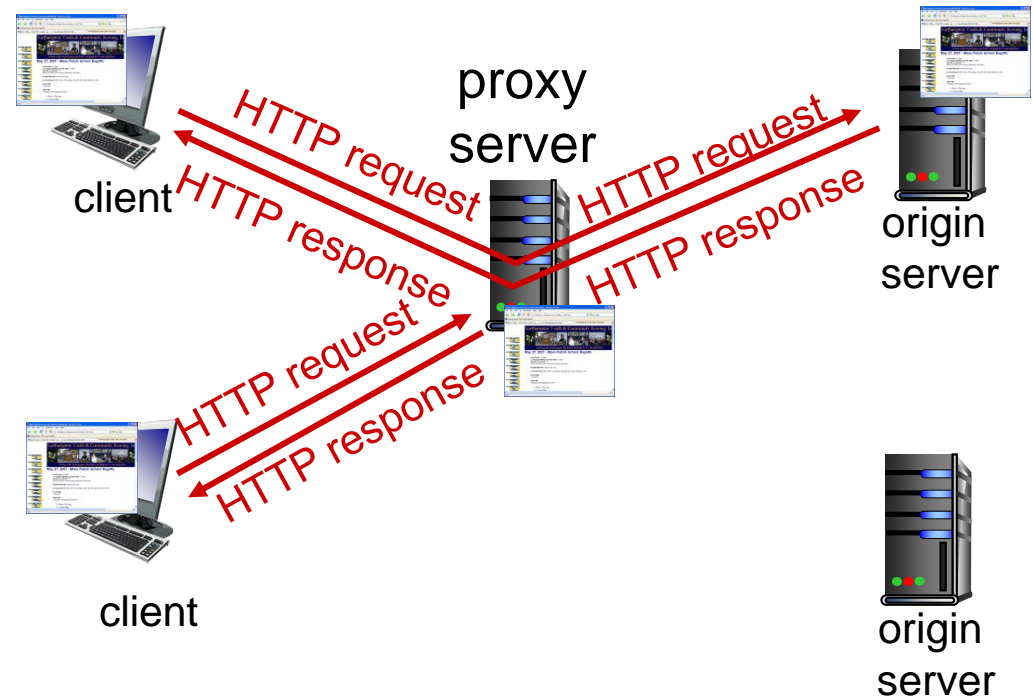
- Application Layer
 - Example network applications
 - conceptual, implementation aspects of network application protocols
 - client-server paradigm
 - transport-layer service models
- Socket Programming
 - Client-Server Example
- Backup Slides
 - Web Caching
 - DNS (Domain Name System)

Web Caches (proxies)

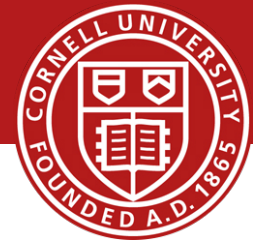


goal: satisfy client request without involving origin server

- user sets browser: Web accesses via cache
- browser sends all HTTP requests to cache
 - object in cache: cache returns object
 - else cache requests object from origin server, then returns object to client



Web Caches (proxies)

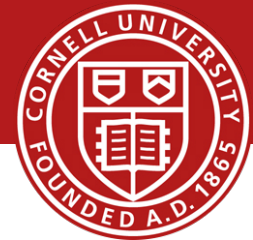


- cache acts as both client and server
 - server for original requesting client
 - client to origin server
- typically cache is installed by ISP (university, company, residential ISP)

why Web caching?

- reduce response time for client request
- reduce traffic on an institution's access link
- Internet dense with caches: enables “poor” content providers to effectively deliver content (so too does P2P file sharing)

Web Caching Example

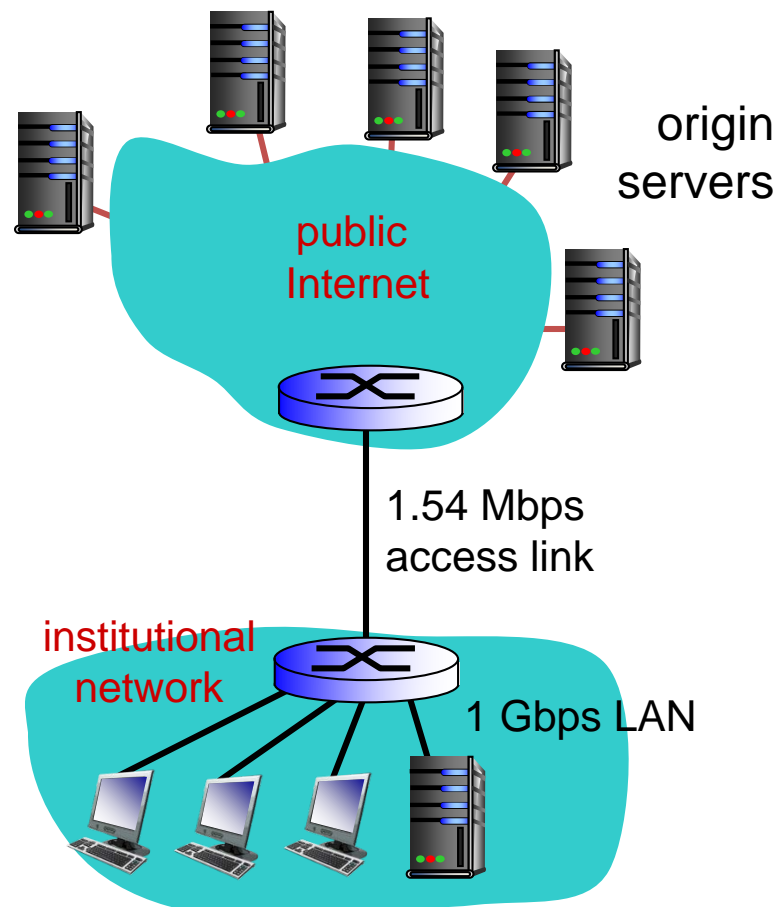


assumptions:

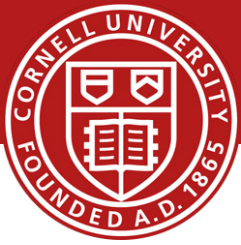
- ❖ avg object size: 100K bits
- ❖ avg request rate from browsers to origin servers: 15/sec
- ❖ avg data rate to browsers: 1.50 Mbps
- ❖ RTT from institutional router to any origin server: 2 sec
- ❖ access link rate: 1.54 Mbps

consequences:

- ❖ LAN utilization: 15%
- ❖ access link utilization = 99% *problem!*
- ❖ total delay = Internet delay + access delay + LAN delay
= 2 sec + minutes + usecs



Web Caching Example: Fatter access Link



assumptions:

- ❖ avg object size: 100K bits
- ❖ avg request rate from browsers to origin servers: 15/sec
- ❖ avg data rate to browsers: 1.50 Mbps
- ❖ RTT from institutional router to any origin server: 2 sec
- ❖ access link rate: 1.54 Mbps

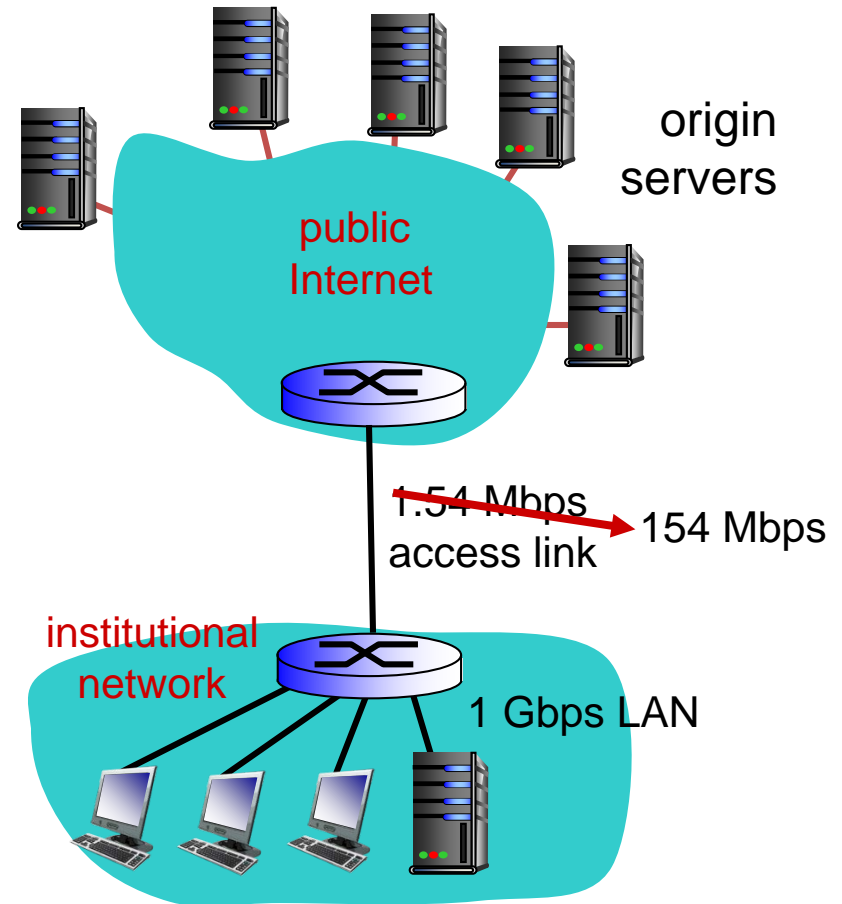
154 Mbps

consequences:

- ❖ LAN utilization: 15%
- ❖ access link utilization = 99%
- ❖ total delay = Internet delay + access delay + LAN delay
= 2 sec + minutes + usecs
= msec

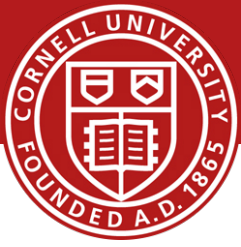
9.9%

msecs



Cost: increased access link speed (not cheap!)

Web Caching Example: Install Local Cache



assumptions:

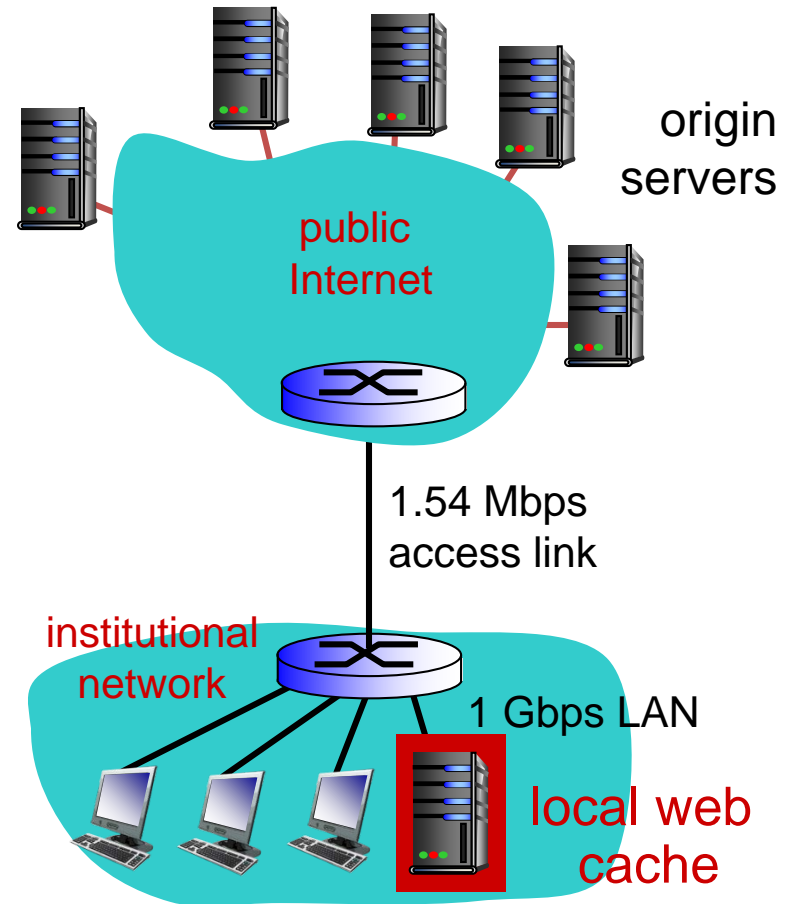
- ❖ avg object size: 100K bits
- ❖ avg request rate from browsers to origin servers: 15/sec
- ❖ avg data rate to browsers: 1.50 Mbps
- ❖ RTT from institutional router to any origin server: 2 sec
- ❖ access link rate: 1.54 Mbps

consequences:

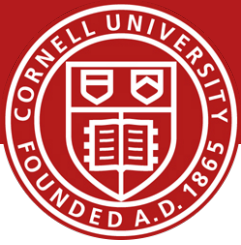
- ❖ LAN utilization: 15%
- ❖ access link utilization = 100%
- ❖ total delay = Internet delay + access delay + LAN delay ?
= 2 sec + min ?

How to compute link utilization, delay?

Cost: web cache (cheap!)

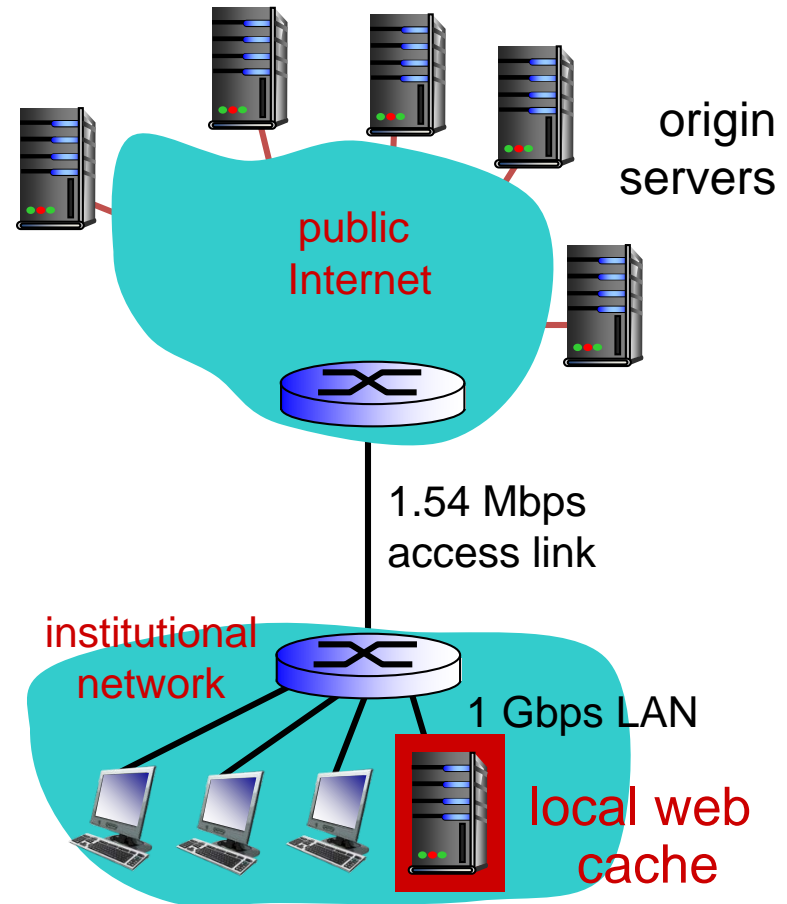


Web Caching Example: Install Local Cache



Calculating access link utilization, delay with cache:

- suppose cache hit rate is 0.4
 - 40% requests satisfied at cache, 60% requests satisfied at origin
- ❖ access link utilization:
 - 60% of requests use access link
- ❖ data rate to browsers over access link = $0.6 * 1.50 \text{ Mbps} = .9 \text{ Mbps}$
 - utilization = $0.9 / 1.54 = .58$
- ❖ total delay
 - = $0.6 * (\text{delay from origin servers}) + 0.4 * (\text{delay when satisfied at cache})$
 - = $0.6 (2.01) + 0.4 (\sim \text{msecs})$
 - = $\sim 1.2 \text{ secs}$
 - less than with 154 Mbps link (and cheaper too!)



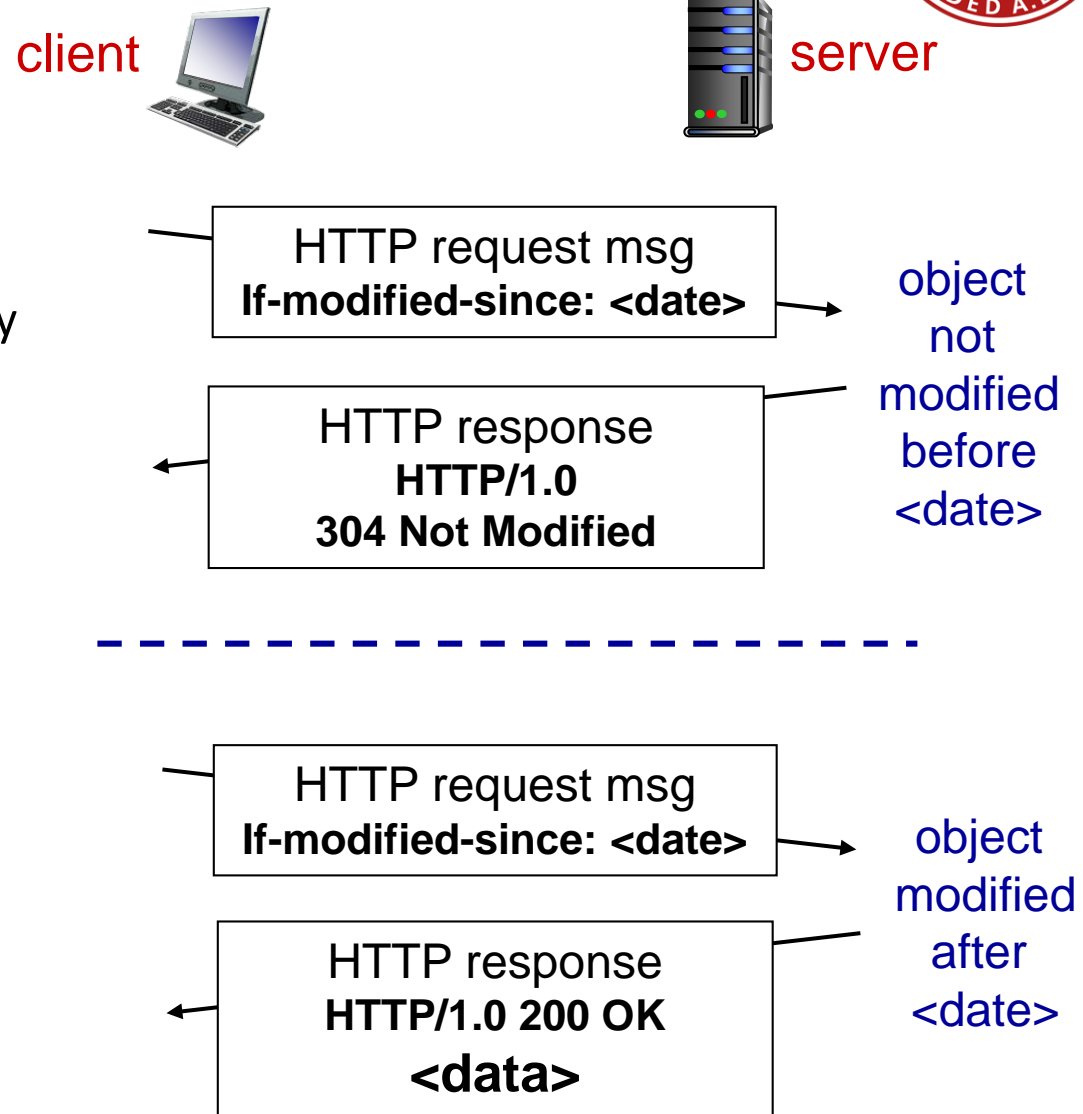
Web Caching Example: Conditional GET



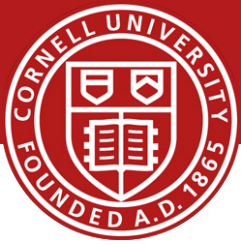
- **Goal:** don't send object if cache has up-to-date cached version
 - no object transmission delay
 - lower link utilization
- **cache:** specify date of cached copy in HTTP request

If-modified-since:
<date>

- **server:** response contains no object if cached copy is up-to-date:
HTTP/1.0 304 Not Modified

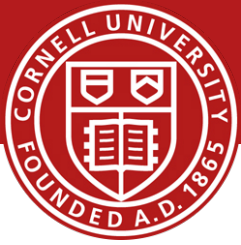


Goals for Today



- Application Layer
 - Example network applications
 - conceptual, implementation aspects of network application protocols
 - client-server paradigm
 - transport-layer service models
- Socket Programming
 - Client-Server Example
- Backup Slides
 - Web Caching
 - DNS (Domain Name System)

DNS (Domain Name System)



people: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

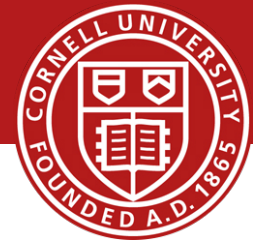
- IP address (32 bit) - used for addressing datagrams
- “name”, e.g., www.yahoo.com - used by humans

Q: how to map between IP address and name, and vice versa ?

Domain Name System:

- *distributed database*
implemented in hierarchy of many *name servers*
- *application-layer protocol:*
hosts, name servers communicate to *resolve* names (address/name translation)
 - note: core Internet function, implemented as application-layer protocol
 - complexity at network’s “edge”

DNS Structure



DNS services

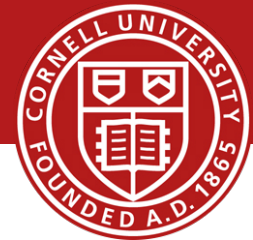
- hostname to IP address translation
- host aliasing
 - canonical, alias names
- mail server aliasing
- load distribution
 - replicated Web servers: many IP addresses correspond to one name

why not centralize DNS?

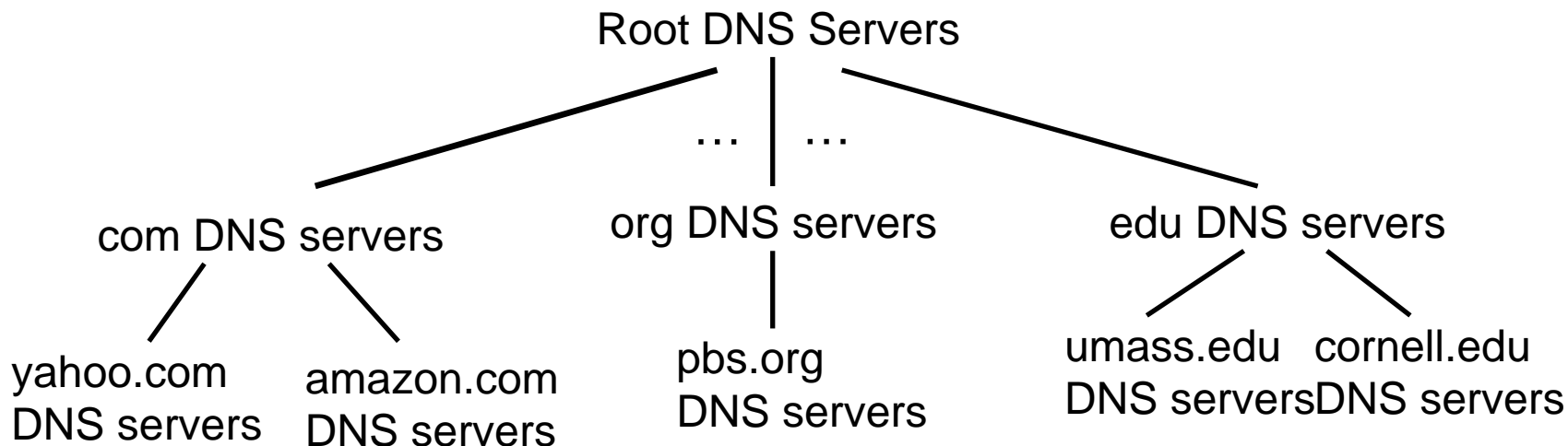
- single point of failure
- traffic volume
- distant centralized database
- maintenance

A: doesn't scale!

DNS Structure



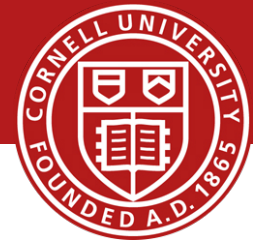
A distributed hierarchical database



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

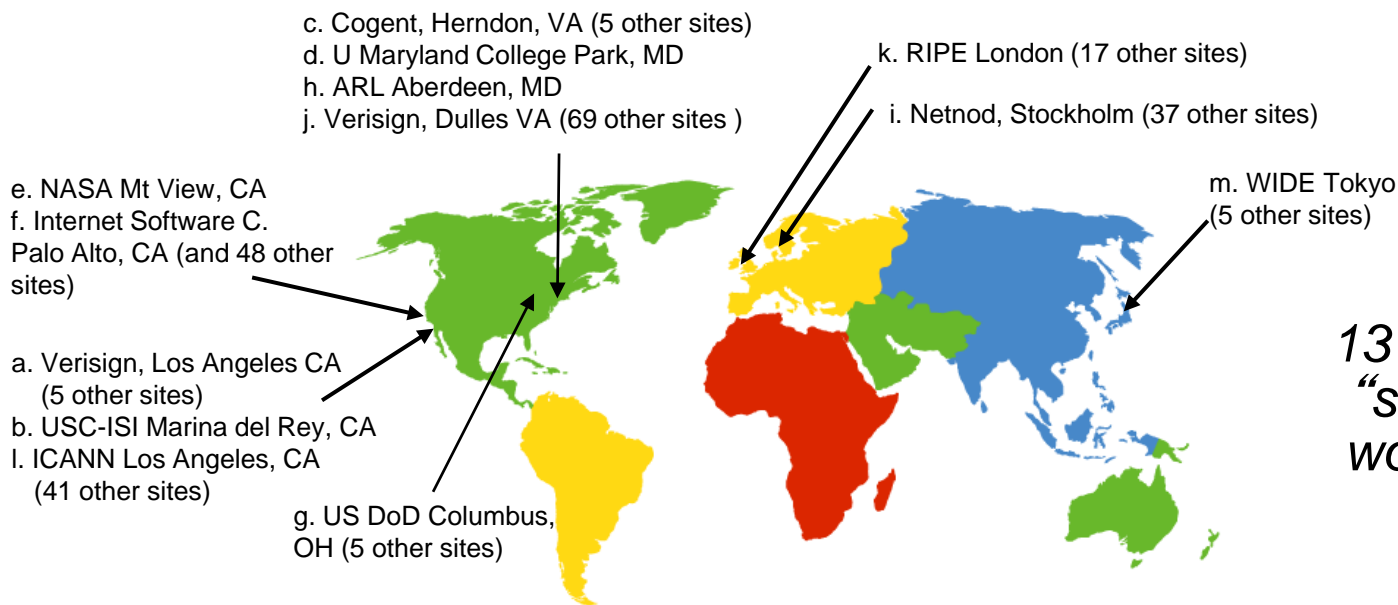
- client queries root server to find com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com

DNS Structure



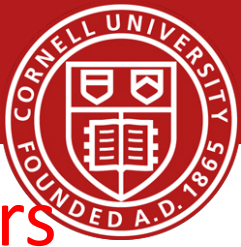
Root name servers

- contacted by local name server that can not resolve name
- root name server:
 - contacts authoritative name server if name mapping not known
 - gets mapping
 - returns mapping to local name server



*13 root name
“servers”
worldwide*

DNS Structure



Top-Level Domain (TLD) and Authoritative Servers

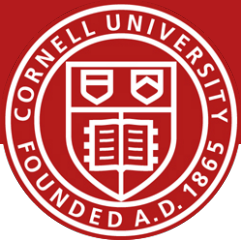
top-level domain (TLD) servers:

- responsible for com, org, net, edu, aero, jobs, museums, and all top-level country domains, e.g.: uk, fr, ca, jp
- Network Solutions maintains servers for .com TLD
- Educause for .edu TLD

authoritative DNS servers:

- organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts
- can be maintained by organization or service provider

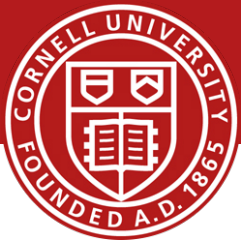
DNS Structure



Local DNS Name Servers

- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
 - also called “default name server”
- when host makes DNS query, query is sent to its local DNS server
 - has local cache of recent name-to-address translation pairs (but may be out of date!)
 - acts as proxy, forwards query into hierarchy

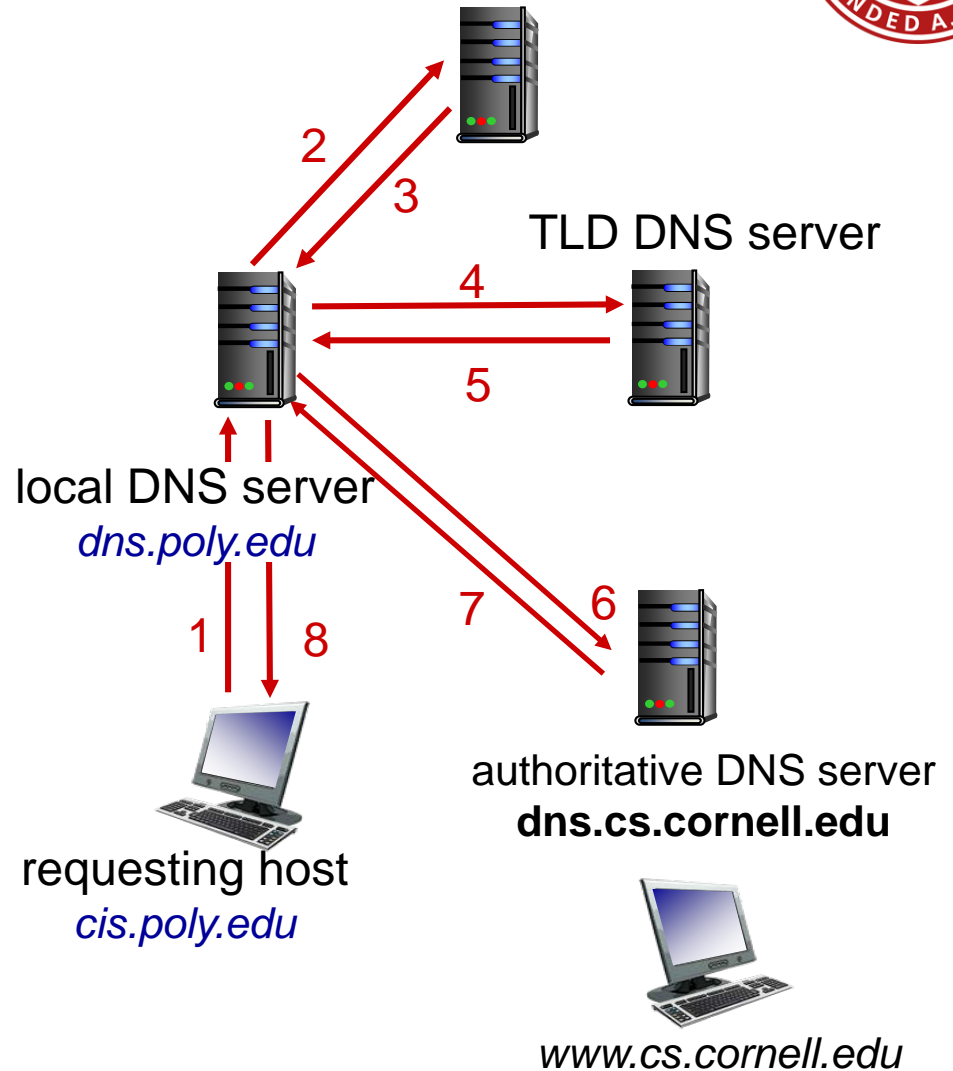
DNS Structure: Resolution example



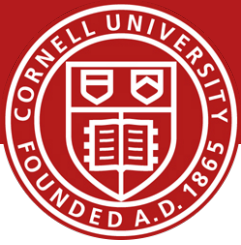
- host at cis.poly.edu wants IP address for www.cs.cornell.edu

iterated query:

- ❖ contacted server replies with name of server to contact
- ❖ “I don’t know this name, but ask this server”

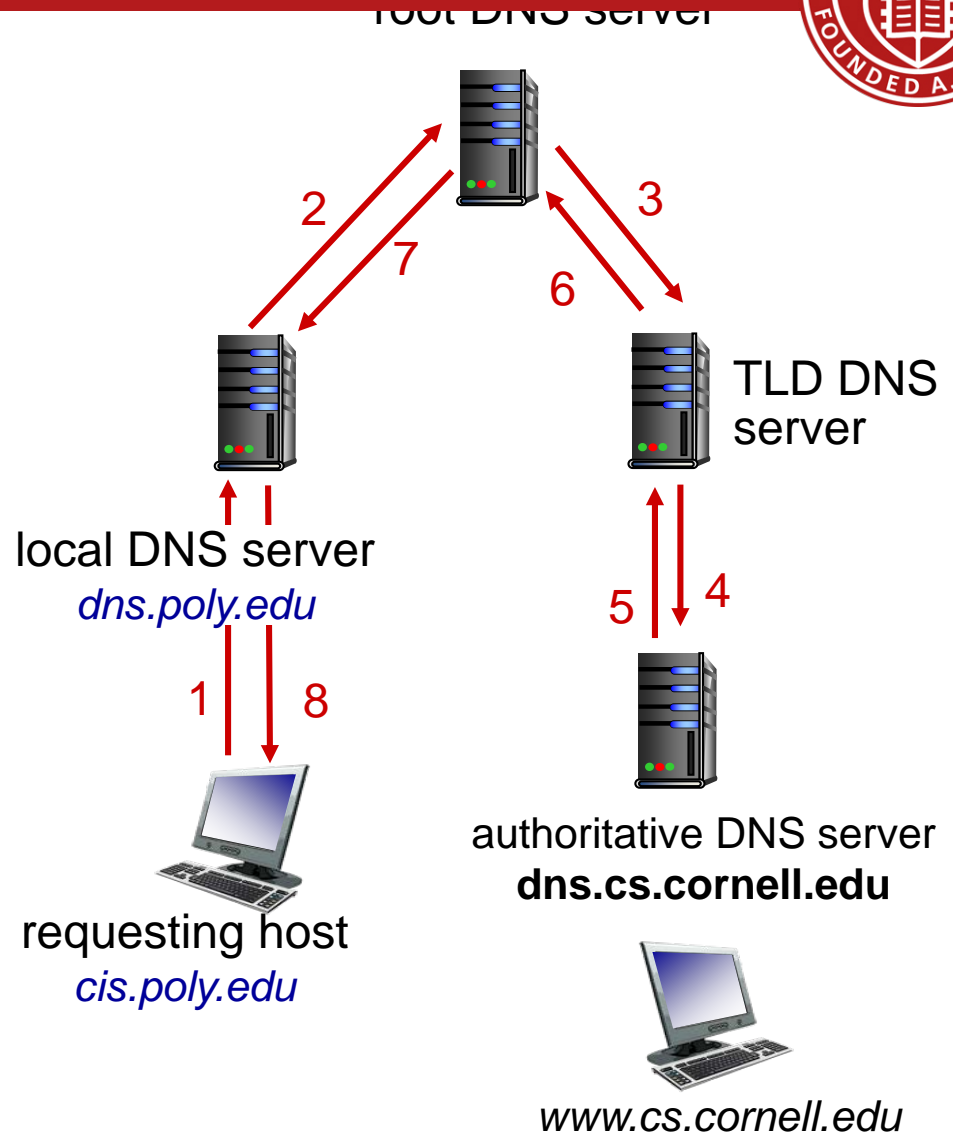


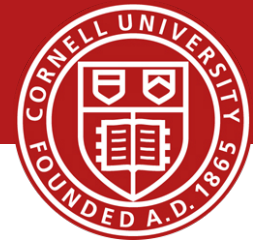
DNS Structure: Resolution example



recursive query:

- ❖ puts burden of name resolution on contacted name server
- ❖ heavy load at upper levels of hierarchy?

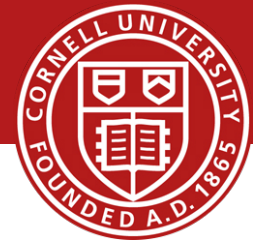




Caching and Updating Records

- once (any) name server learns mapping, it *caches* mapping
 - cache entries timeout (disappear) after some time (TTL)
 - TLD servers typically cached in local name servers
 - thus root name servers not often visited
- cached entries may be *out-of-date* (best effort name-to-address translation!)
 - if name host changes IP address, may not be known Internet-wide until all TTLs expire
- update/notify mechanisms proposed IETF standard
 - RFC 2136

DNS Structure



DNS Records

DNS: distributed db storing resource records (RR)

RR format: (name, value, type, ttl)

type=A

- **name** is hostname
- **value** is IP address

type=NS

- **name** is domain (e.g., foo.com)
- **value** is hostname of authoritative name server for this domain

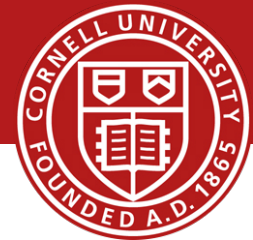
type=CNAME

- **name** is alias name for some “canonical” (the real) name
- **www.ibm.com** is really **servereast.backup2.ibm.com**
- **value** is canonical name

type=MX

- **value** is name of mailserver associated with **name**

DNS Structure

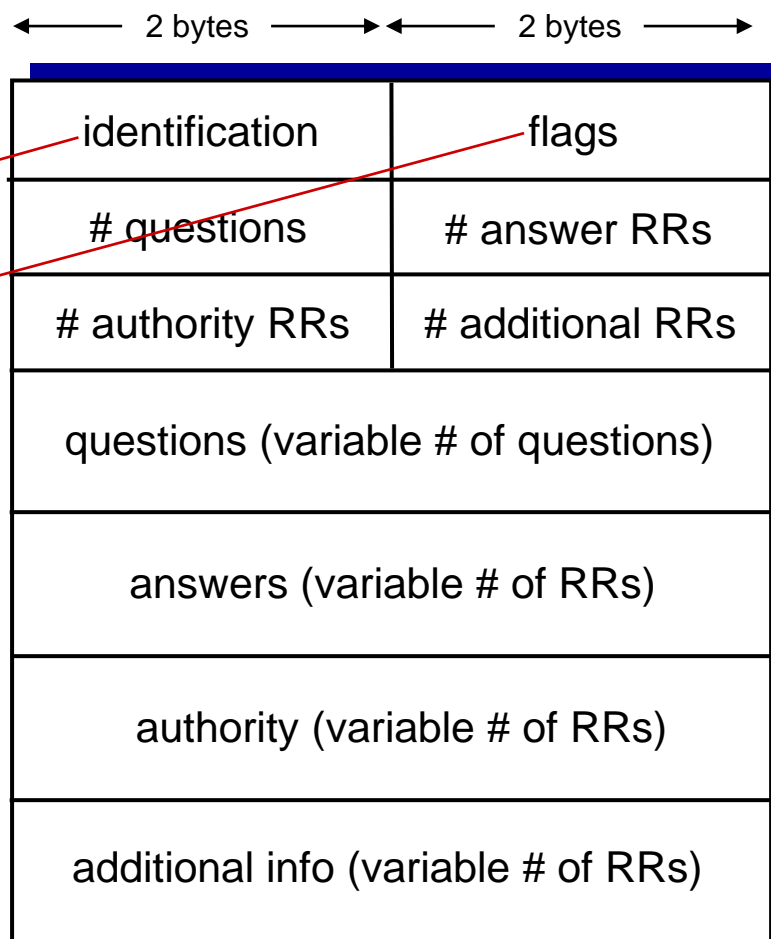


DNS Protocol and Messages

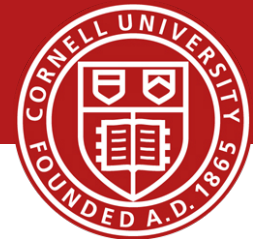
- *query* and *reply* messages, both with same *message format*

msg header

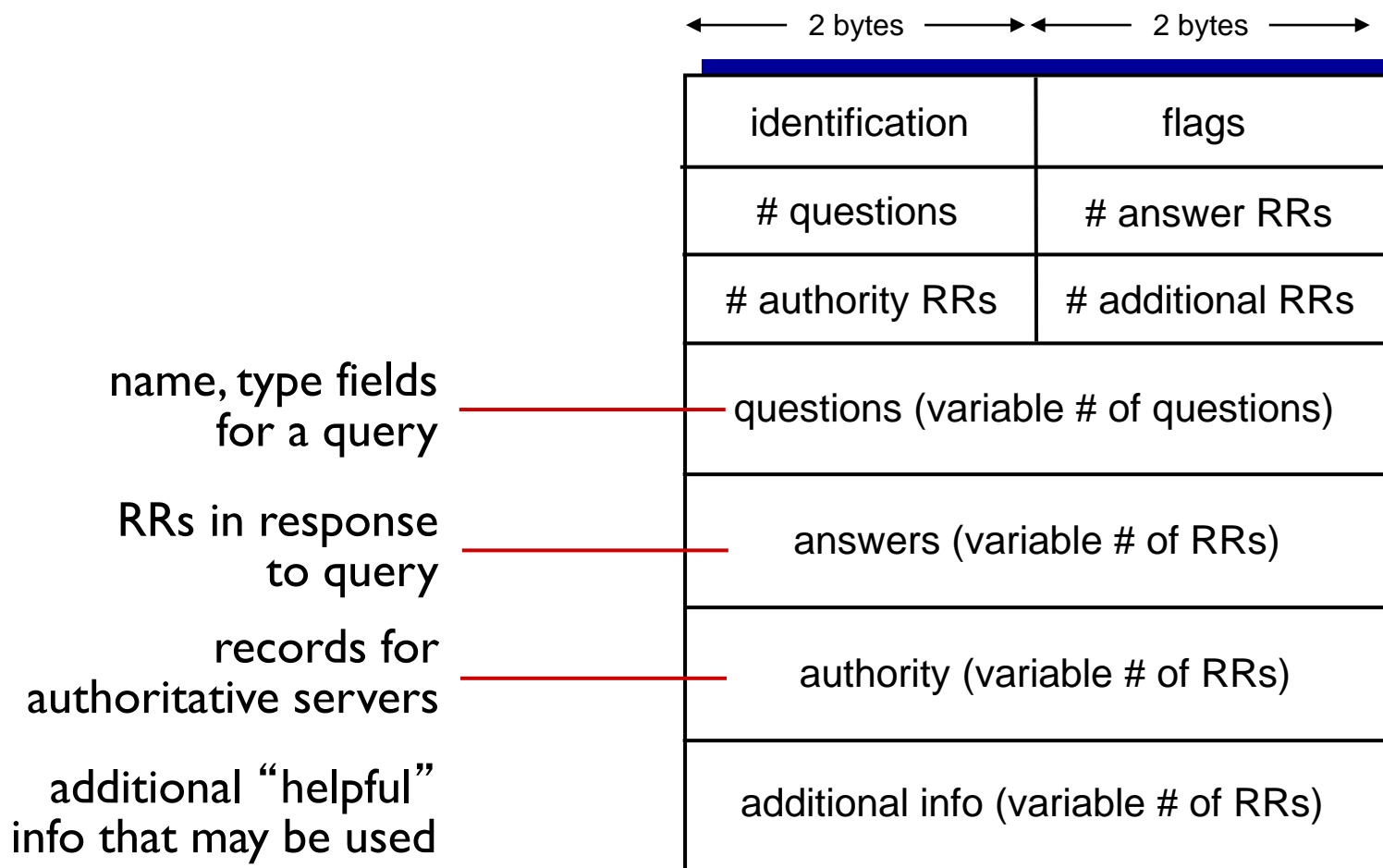
- ❖ **identification**: 16 bit # for query, reply to query uses same #
- ❖ **flags**:
 - query or reply
 - recursion desired
 - recursion available
 - reply is authoritative



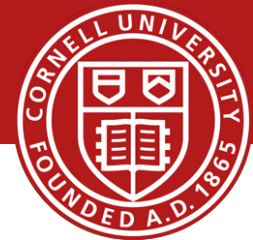
DNS Structure



DNS Protocol and Messages



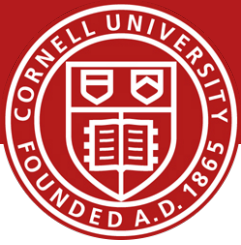
DNS Structure



Inserting Records into DNS

- example: new startup “Network Utopia”
- register name networkutopia.com at *DNS registrar* (e.g., Network Solutions)
 - provide names, IP addresses of authoritative name server (primary and secondary)
 - registrar inserts two RRs into .com TLD server:
(networkutopia.com, dns1.networkutopia.com, NS)
(dns1.networkutopia.com, 212.212.212.1, A)
- create authoritative server type A record for www.networkutopia.com; type MX record for networkutopia.com

Attacking DNS



DDoS attacks

- Bombard root servers with traffic
 - Not successful to date
 - Traffic Filtering
 - Local DNS servers cache IPs of TLD servers, allowing root server bypass
- Bombard TLD servers
 - Potentially more dangerous

Redirect attacks

- Man-in-middle
 - Intercept queries
- DNS poisoning
 - Send bogus replies to DNS server, which caches

Exploit DNS for DDoS

- Send queries with spoofed source address: target IP
- Requires amplification