Computer Networks and Applications

COMP 3331/COMP 9331

Week 3

Application Layer (Email, DNS, CDN, Socket Programming)

Reading Guide: Chapter 2, Sections 2.3, 2.4, 2.6, 2.7

Application Layer: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 electronic mail
- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks (CDNs)

2.7 socket programming with UDP and TCP

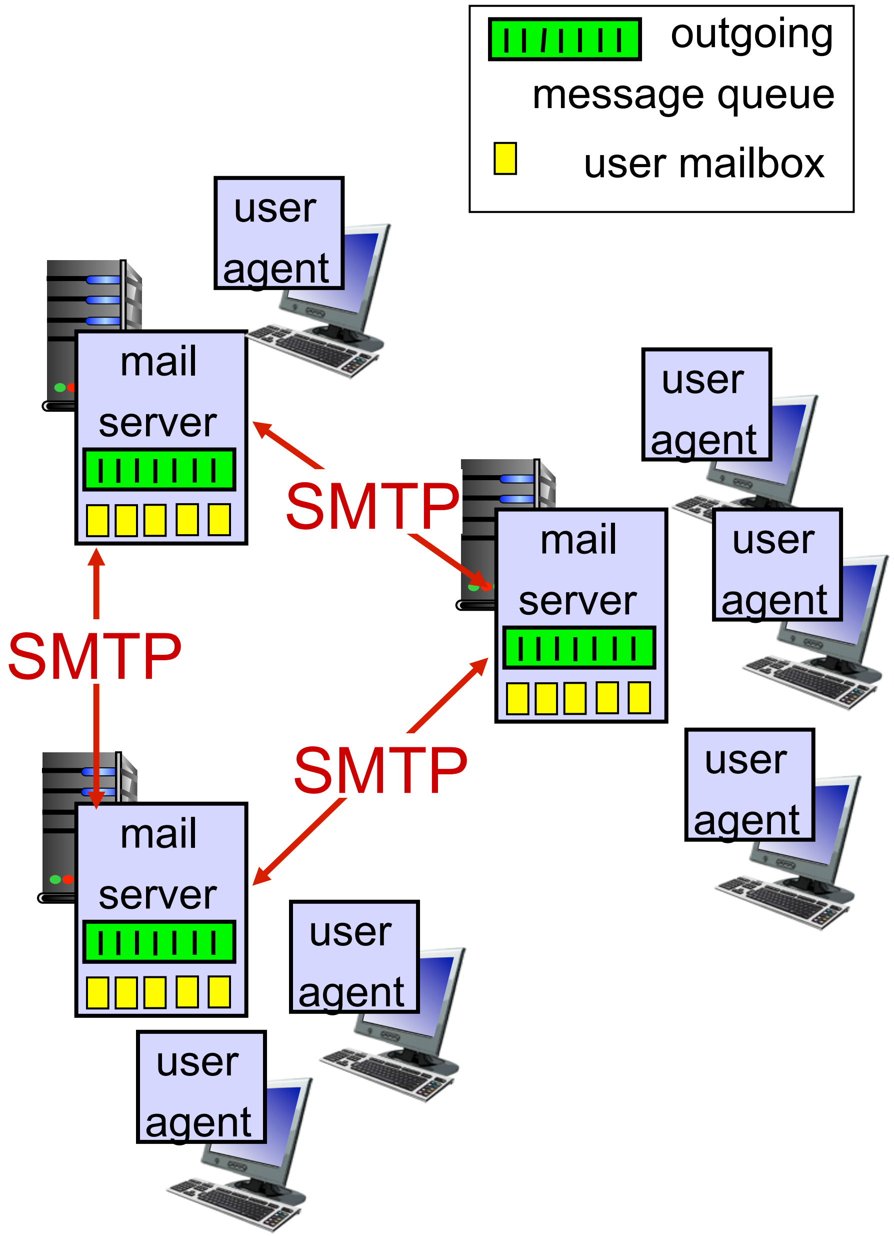
Electronic mail

Three major components:

1. user agents
2. mail servers
3. simple mail transfer protocol: SMTP

User Agent

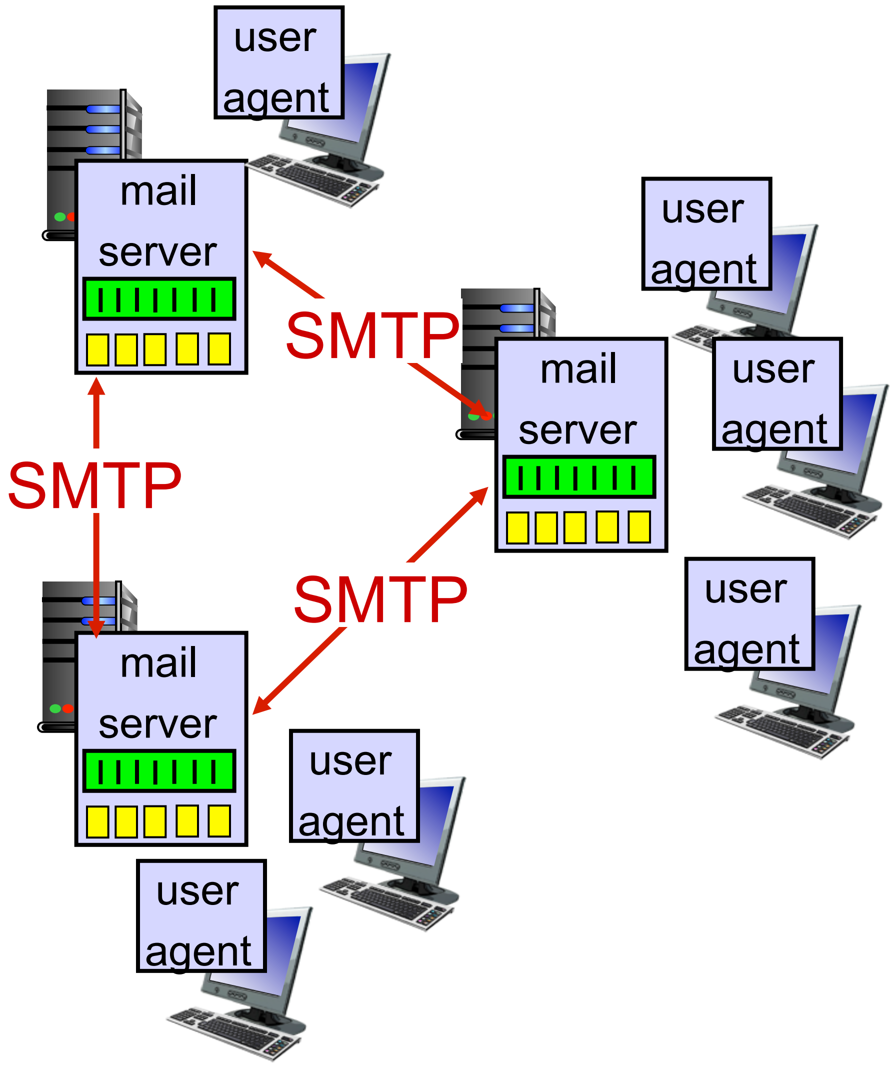
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Outlook, gmail, iPhone mail client
- outgoing, incoming messages stored on server



Electronic mail: mail servers

mail servers:

- *mailbox* contains incoming messages for user
- *message queue* of outgoing (to be sent) mail messages
- *SMTP protocol* between mail servers to send email messages
  - client: sending mail server
  - "server": receiving mail server

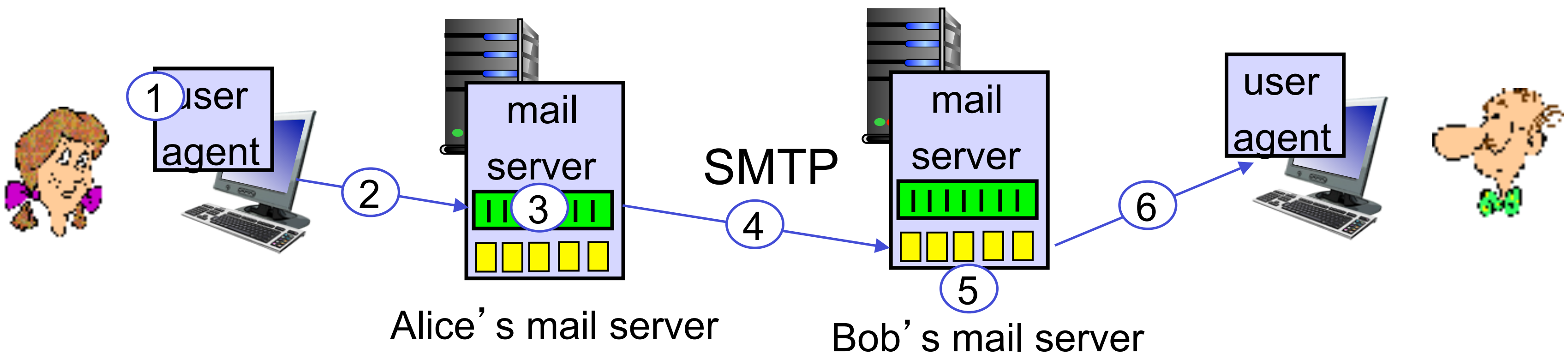


Electronic Mail: SMTP [RFC 2821]

- uses TCP to reliably transfer email message from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction (like HTTP, FTP)
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCII

Scenario: Alice sends message to Bob

1) Alice uses UA (user agent) to compose message "to" `bob@someschool.edu`
2) Alice's UA sends message to her mail server; message placed in message queue
3) client side of SMTP opens TCP connection with Bob's mail server
4) SMTP client sends Alice's message over the TCP connection
5) Bob's mail server places the message in Bob's mailbox
6) Bob invokes his user agent to read message



Sample SMTP interaction

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250  Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

SMTP: final words

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses `CRLF.CRLF` to determine end of message

comparison with HTTP:

- HTTP: pull
- SMTP: push

- both have ASCII command/response interaction, status codes

- HTTP: each object encapsulated in its own response msg
- SMTP: multiple objects sent in multipart msg

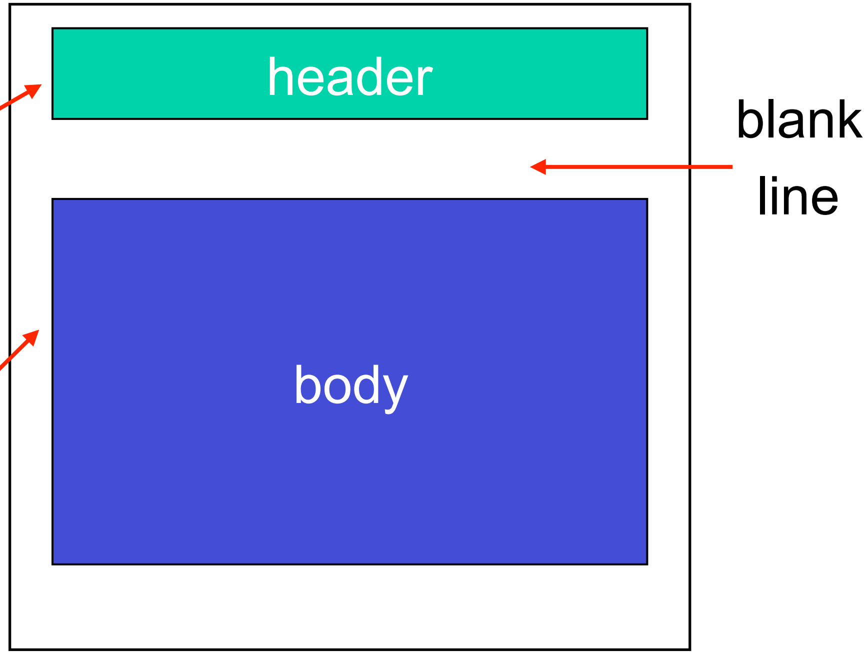
Mail message format

SMTP: protocol for exchanging email msgs

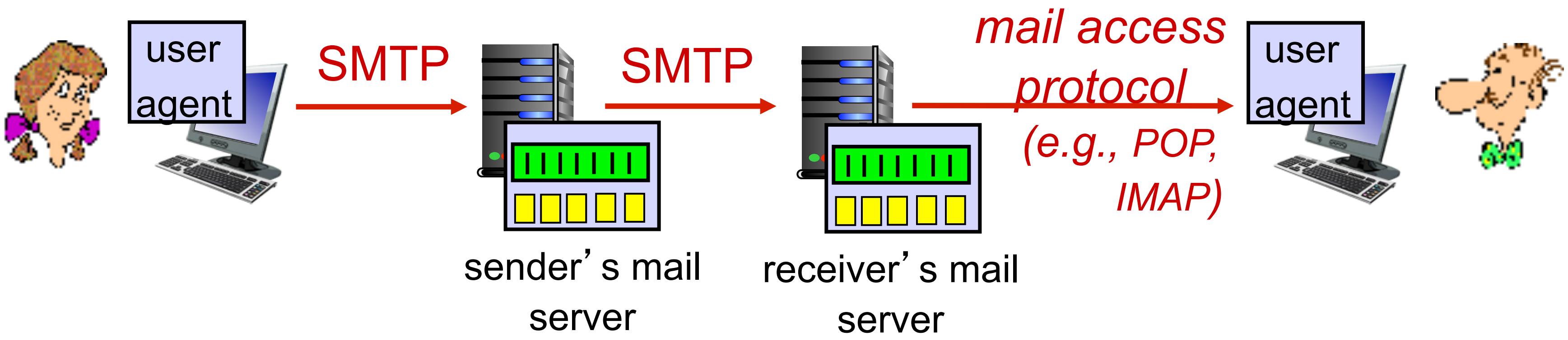
RFC ~~2822~~ (5322): standard for text message format:

- header lines, e.g.,
  - To:
  - From:
  - Subject:

  different from SMTP MAIL FROM, RCPT TO: commands!
- Body: the "message"
  - ASCII characters only

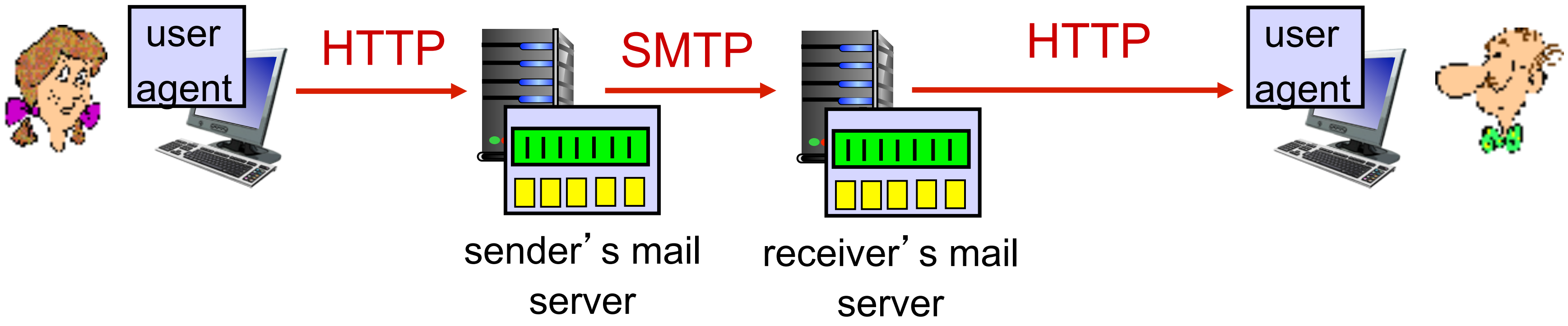


Mail access protocols POP vs. IMAP



- SMTP: delivery/storage to receiver's server
- mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]: authorization, download (once downloaded, its gone from the server)
  - IMAP: Internet Mail Access Protocol [RFC 1730]: more features, including manipulation of stored msgs on server (good for nomadic users accessing mails from different machines)

Web-based Email



- Use web browser to access emails
- Examples: Gmail, Yahoo, …

2. Application Layer: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 electronic mail
- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks (CDNs)

2.7 socket programming with UDP and TCP

A nice overview: https://webhostinggeeks.com/guides/dns/

DNS: domain name system

people: many identifiers:
- TFN, 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: History

- Initially all host-address mappings were in a hosts.txt file (in /etc/hosts):
  - Maintained by the Stanford Research Institute (SRI)
  - Changes were submitted to SRI by email
  - New versions of hosts.txt periodically FTP'd from SRI
  - An administrator could pick names at their discretion

Jon Postel

- As the Internet grew this system broke down:
  - SRI couldn't handle the load; names were not unique; hosts had inaccurate copies of hosts.txt

- The Domain Name System (DNS) was invented to fix this

http://www.wired.com/2012/10/joe-postel/

DNS: services, 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
  - Content Distribution Networks: use IP address of requesting host to find best suitable server
    - Example: closest, least-loaded, etc

why not centralize DNS?

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

A: *doesn't scale!*

Goals

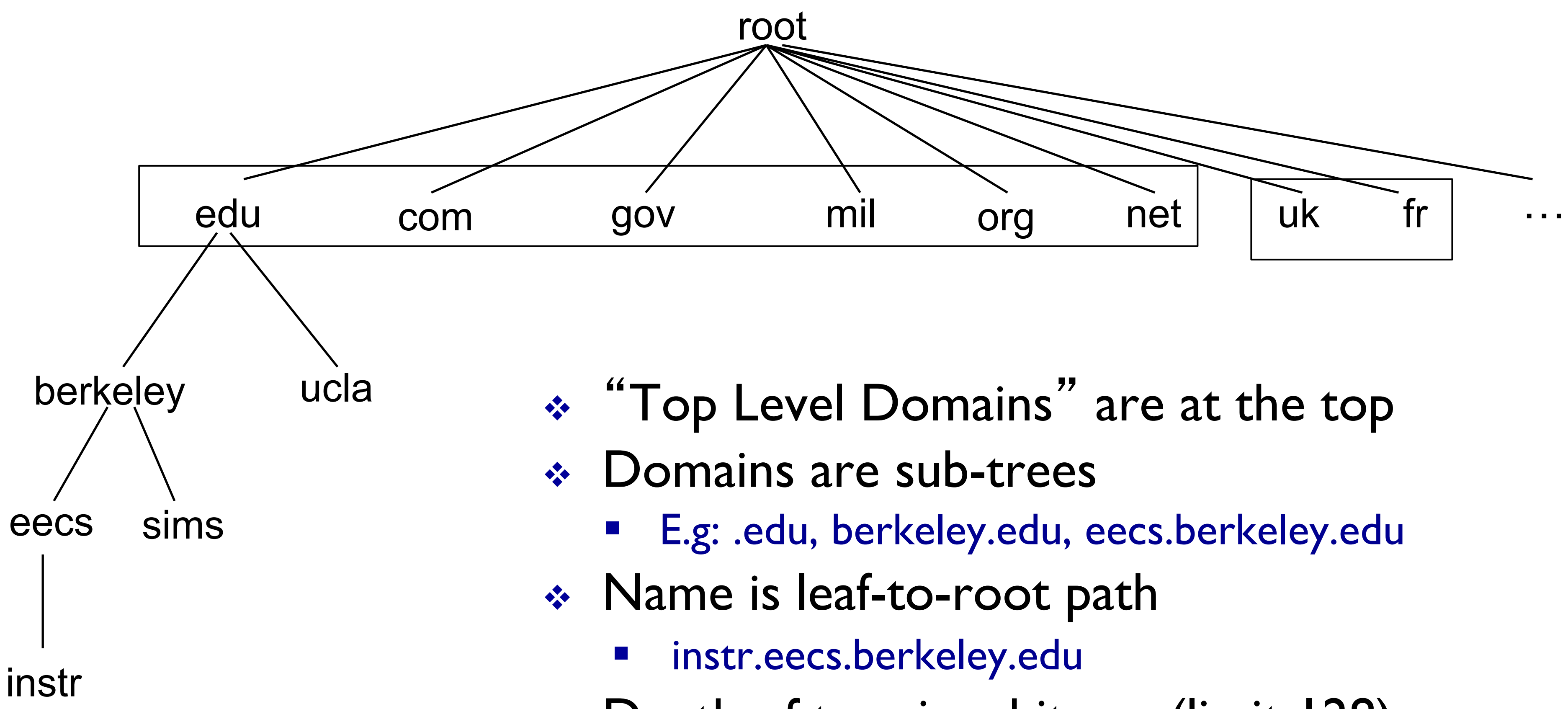
- No naming conflicts (uniqueness)
- Scalable
  - many names
  - (secondary) frequent updates
- Distributed, autonomous administration
  - Ability to update my own (machines') names
  - Don't have to track everybody's updates
- Highly available
- Lookups should be fast

Key idea: Hierarchy

Three intertwined hierarchies

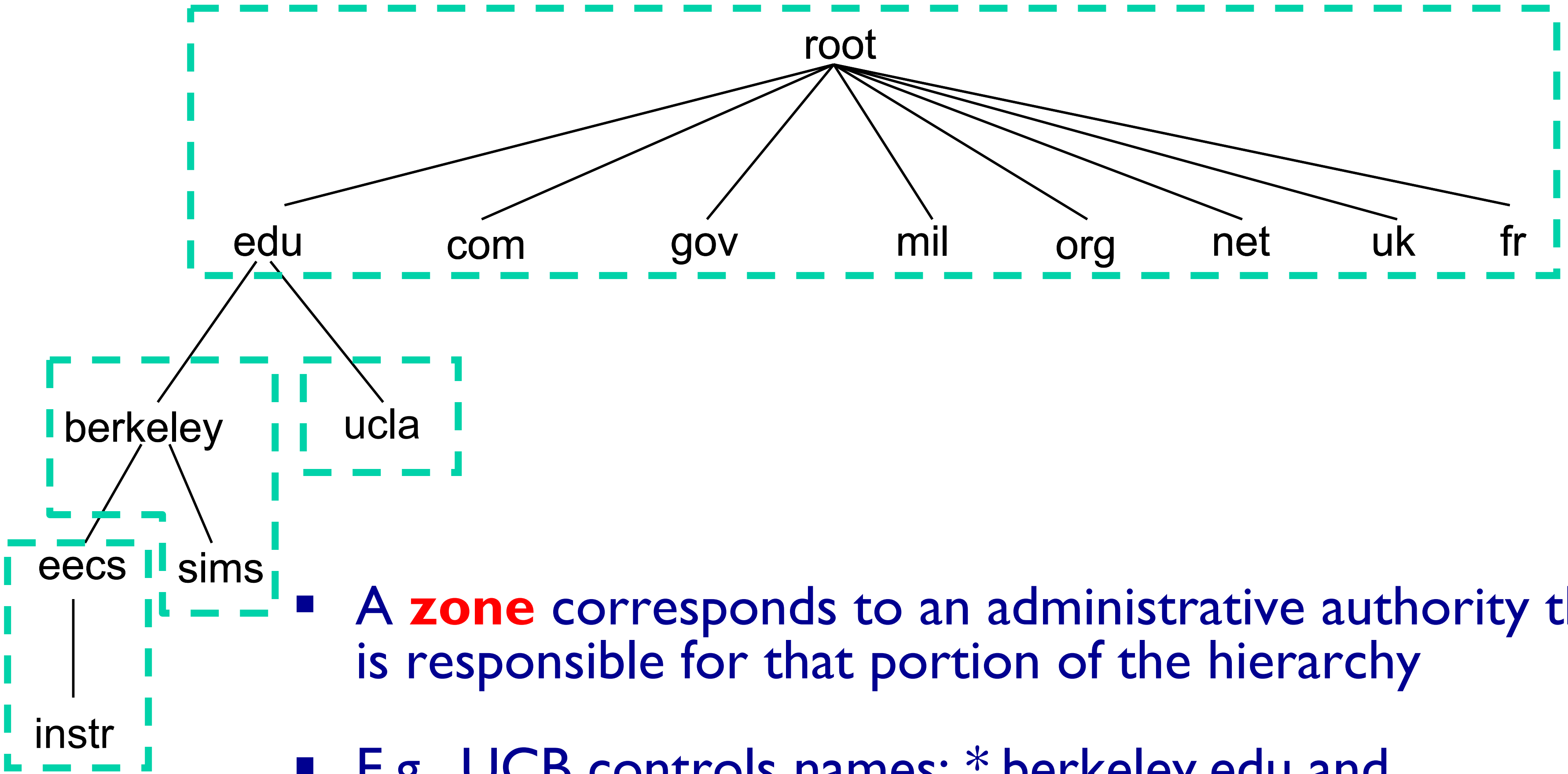
- Hierarchical namespace
  - As opposed to original flat namespace
- Hierarchically administered
  - As opposed to centralised
- (Distributed) hierarchy of servers
  - As opposed to centralised storage

Hierarchical Namespace



- "Top Level Domains" are at the top
- Domains are sub-trees
  - E.g: .edu, berkeley.edu, eecs.berkeley.edu
- Name is leaf-to-root path
  - instr.eecs.berkeley.edu
- Depth of tree is arbitrary (limit 128)
- Name collisions trivially avoided
  - each domain is responsible

Hierarchical Administration



- A **zone** corresponds to an administrative authority that is responsible for that portion of the hierarchy
- E.g., UCB controls names: *.berkeley.edu and *.sims.berkeley.edu
- E.g., EECS controls names: *.eecs.berkeley.edu

Server Hierarchy

- Top of hierarchy: Root servers
  - Location hardwired into other servers
- Next Level: Top-level domain (TLD) servers
  - .com, .edu, etc.
  - Managed professionally
- Bottom Level: Authoritative DNS servers
  - Actually store the name-to-address mapping
  - Maintained by the corresponding administrative authority

Server Hierarchy

- Each server stores a (small!) subset of the total DNS database
- An authoritative DNS server stores "resource records" for all DNS names in the domain that it has authority for
- Each server needs to know other servers that are responsible for the other portions of the hierarchy
  - Every server knows the root
  - Root server knows about all top-level domains

DNS Root Servers

- 400+ root servers scattered around the world
- USA, RUSSIA, UK, INDIA, AUSTRALIA, …
- 13 organizations maintain these 400+ root severs
- Provide IP addresses of TLD servers

TLD, 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
- Professional companies maintain these servers
  - Verisign maintains servers for .com TLD
  - Educause for .edu TLD
- Provides IP addresss for Authoritative servers

authoritative DNS servers (stores name-to-IP mapping):

- 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

Local DNS name server

- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called "default name server"
- Hosts configured with local DNS server address (e.g., /etc/resolv.conf) or learn server via a host configuration protocol (e.g., DHCP)
- Client application
  - Obtain DNS name (e.g., from URL)
  - Do **gethostbyname()** to trigger DNS request to its local DNS 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

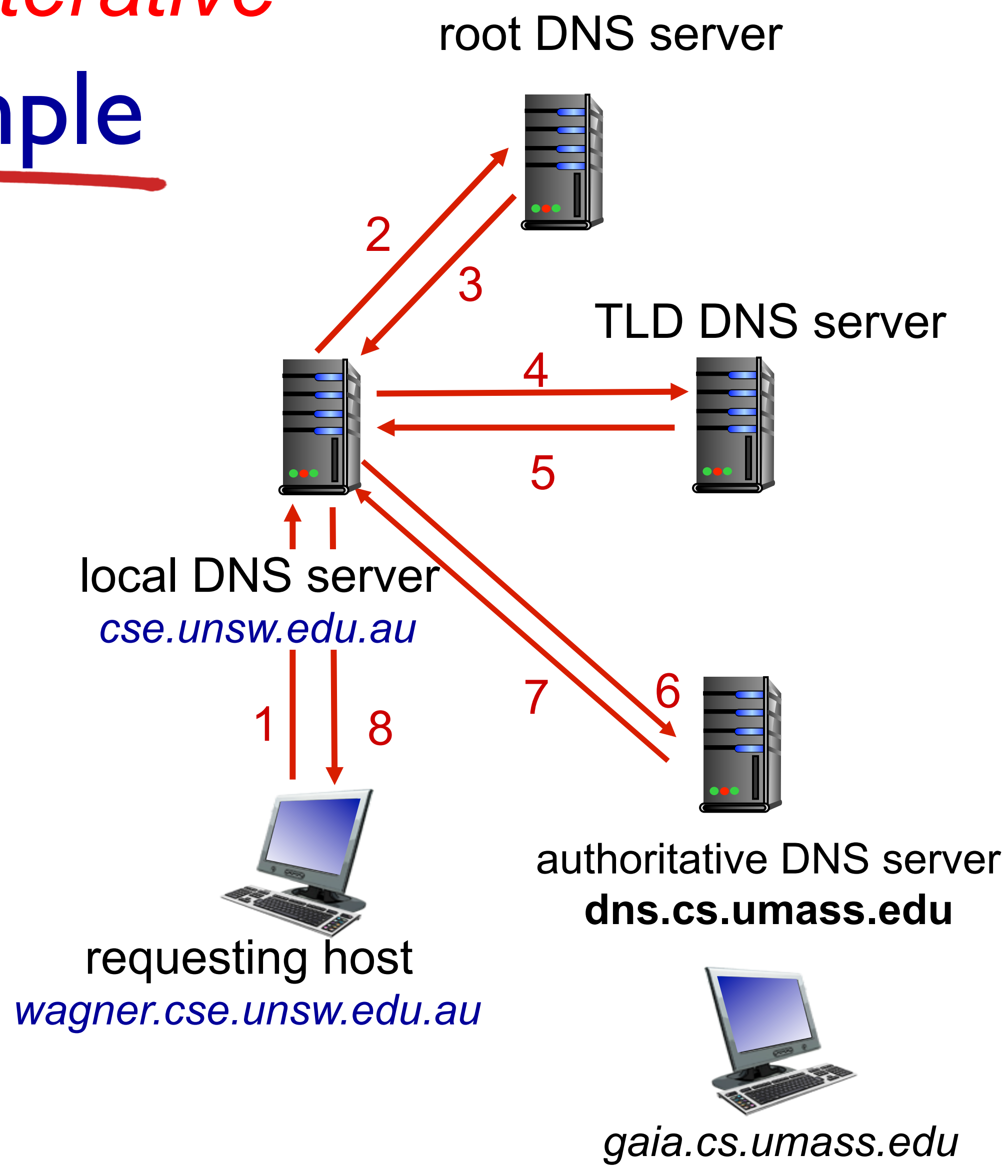
DNS name resolution example

iterative

- host at wagner.cse.unsw.edu.au wants IP address for gaia.cs.umass.edu

iterated query:

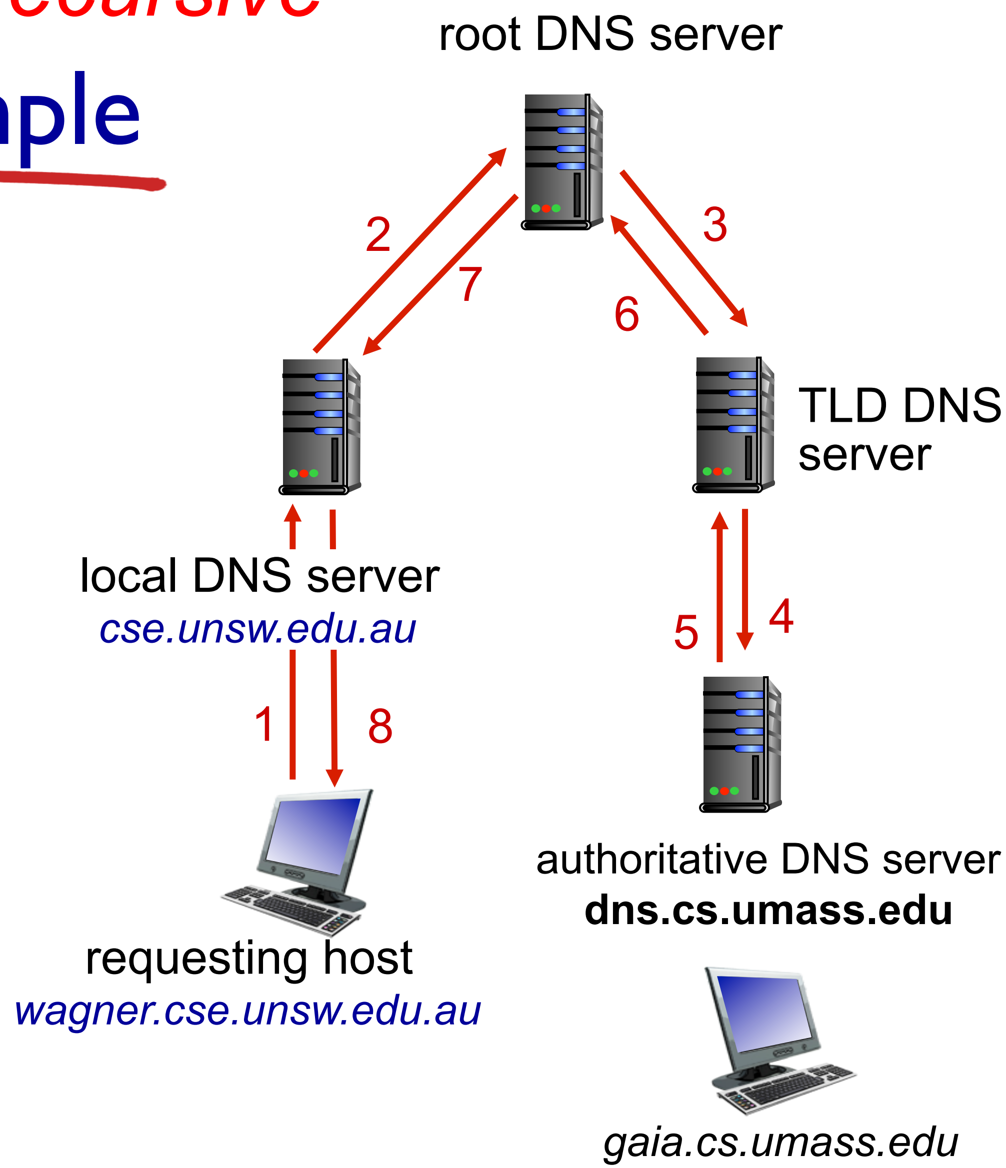
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS name resolution example *recursive*

recursive query:

- puts burden of name resolution on contacted name server



DNS: caching, 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
- Subsequent requests need not burden DNS
- 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

DNS resource records (RRs)

Each DNS reply carries one or more RRs

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`**

The meaning of *name* and *value* depends on *type*

Examples

- Type A: (relay1.bar.foo.com, 145.37.93.126, A)
- Type NS: (foo.com, dns.foo.com, NS)
- Type CNAME: (foo.com, relay1.bar.foo.com, CNAME)
- Type MX: (foo.com, mail.bar.foo.com, MX)

- A company can name both its web server and a mail server as foo.com → a dns client must use MX query to find out the mail server and CNAME to locate the web server

DNS protocol, messages

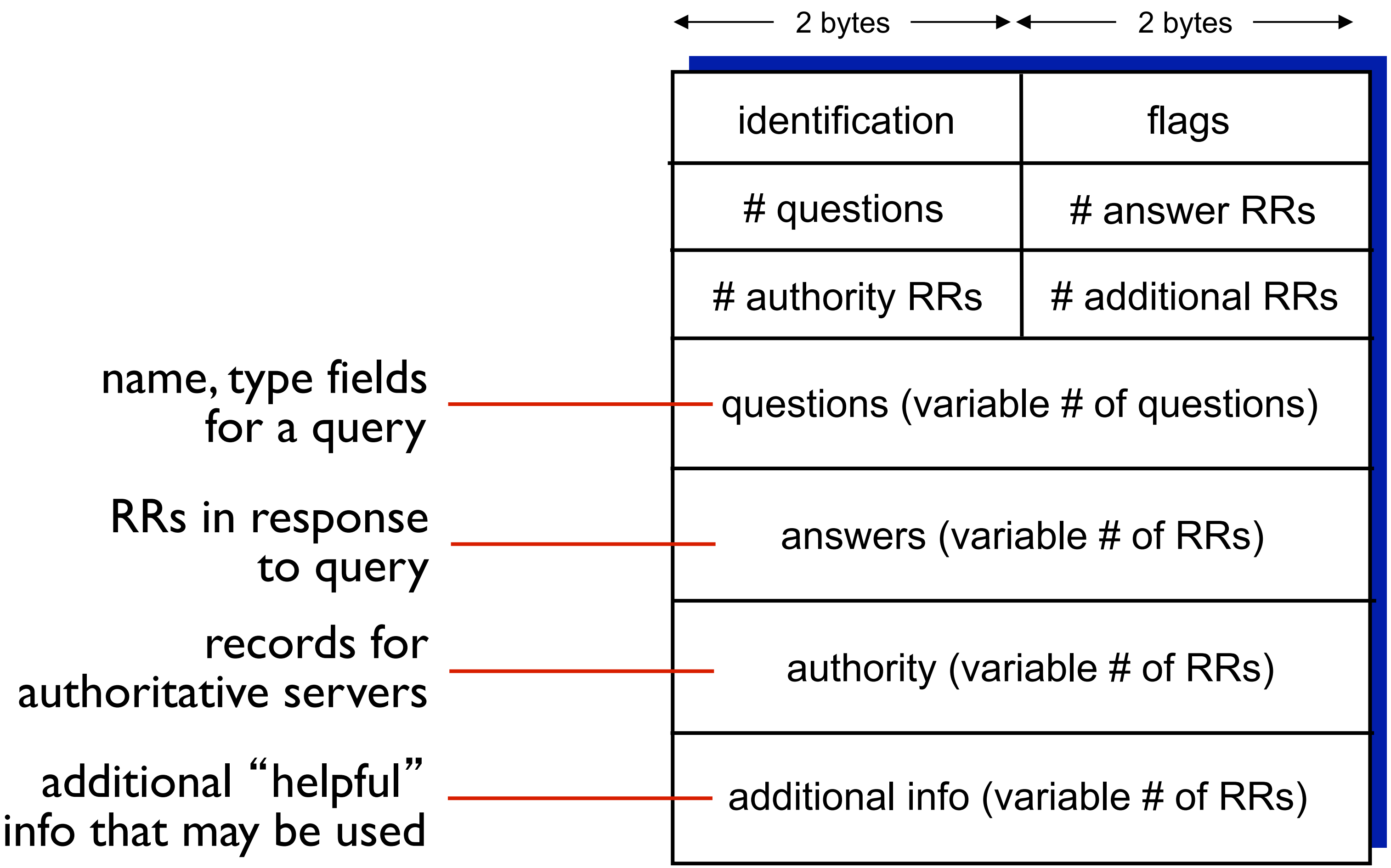
- *query* and *reply* messages, both with same *message format*

msg header

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

← 2 bytes →	← 2 bytes →
identification	flags
# questions	# answer RRs
# authority RRs	# additional RRs
questions (variable # of questions)	
answers (variable # of RRs)	
authority (variable # of RRs)	
additional info (variable # of RRs)	

DNS protocol, messages



An Example

Try this out yourself.

```
bash-3.2$ dig www.oxford.ac.uk

; <<>> DiG 9.8.3-P1 <<>> www.oxford.ac.uk
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 35102
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 4, ADDITIONAL: 5

;; QUESTION SECTION:
;www.oxford.ac.uk.              IN      A

;; ANSWER SECTION:
www.oxford.ac.uk.       300     IN      A       129.67.242.154
www.oxford.ac.uk.       300     IN      A       129.67.242.155

;; AUTHORITY SECTION:
oxford.ac.uk.           86399   IN      NS      dns2.ox.ac.uk.
oxford.ac.uk.           86399   IN      NS      dns1.ox.ac.uk.
oxford.ac.uk.           86399   IN      NS      ns2.ja.net.
oxford.ac.uk.           86399   IN      NS      dns0.ox.ac.uk.

;; ADDITIONAL SECTION:
ns2.ja.net.             33560   IN      A       193.63.105.17
ns2.ja.net.             33560   IN      AAAA    2001:630:0:45::11
dns0.ox.ac.uk.          48090   IN      A       129.67.1.190
dns1.ox.ac.uk.          86399   IN      A       129.67.1.191
dns2.ox.ac.uk.          54339   IN      A       163.1.2.190

;; Query time: 589 msec
;; SERVER: 129.94.172.11#53(129.94.172.11)
;; WHEN: Thu Mar  9 17:53:52 2017
;; MSG SIZE  rcvd: 242
```

Intermission

- So far, we have seen how to retrieve records from DNS servers

- Now let's see how we can put records in DNS servers in the first place

Inserting records into DNS

- example: new startup "Network Utopia"
- register name networkuptopia.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.networkuptopia.com; type MX record for networkutopia.com
- Q: Where do you insert these type A and type MX records?

  A: authoritative DNS server of network Utopia

Visiting webpage of Network Utopia

1. Alice in Sydney wants to visit web page of Network Utopia
2. Alice's browser sends DNS query to local DNS server
3. Local DNS server contacts TLD server for com (assuming TLD server address is cached in local server to avoid going to root)
4. TLD server sends reply to local server with two RRs
   1. **(networkutopia.com, dns1.networkutopia.com, NS)**
   2. **(dns1.networkutopia.com, 212.212.212.1, A)**
5. Local DNS server sends DNS query to **212.212.212.1 asking for Type A record for** www.networkutopia.com
6. **The response provides RR that contains say 212.212.71.4**
7. **Alice's browser now initiates HTTP request to 212.212.71.4 to get the web pages for Network Utopia**

Reliability

- DNS servers are replicated (primary/secondary)
  - Name service available if at least one replica is up
  - Queries can be load-balanced between replicas
- Usually, UDP used for queries
  - Need reliability: must implement this on top of UDP
  - Spec supports TCP too, but not always implemented
- Try alternate servers on timeout
  - Exponential backoff when retrying same server
- Same identifier for all queries
  - Don't care which server responds

DNS provides Indirection

- Addresses can change underneath
  - Move www.cnn.com to 4.125.91.21
  - Humans/Apps should be unaffected
- Name could map to multiple IP addresses
  - Enables
    - Load-balancing
    - Reducing latency by picking nearby servers
- Multiple names for the same address
  - E.g., many services (mail, www, ftp) on same machine
  - E.g., aliases like www.cnn.com and cnn.com
- But, this flexibility applies only within domain!

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 to be bypassed
- 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

Want to dig deeper?

http://www.networkworld.com/article/2886283/security0/top-10-dns-attacks-likely-to-infiltrate-your-network.html

Dig deeper?

DNS Cache Poisoning Test
https://www.grc.com/dns/dns.htm

DNSSEC: DNS Security Extensions,
http://www.dnssec.net

Application Layer: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks (CDNs)

2.7 socket programming with UDP and TCP

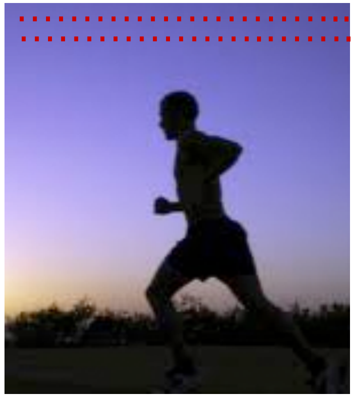
Video Streaming and CDNs: context

- video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube: 37%, 16% of downstream residential ISP traffic
  - ~1B YouTube users, ~75M Netflix users
- challenge: scale - how to reach ~1B users?
  - single mega-video server won't work (why?)
- challenge: heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- *solution:* distributed, application-level infrastructure

Multimedia: video

- video: sequence of images displayed at constant rate
  - e.g., 24 images/sec
- digital image: array of pixels
  - each pixel represented by bits
- coding: use redundancy *within* and *between* images to decrease # bits used to encode image
  - spatial (within image)
  - temporal (from one image to next)

spatial coding example: instead of sending *N* values of same color (all purple), send only two values: color value (*purple*) *and number of repeated values* (N)



frame *i*

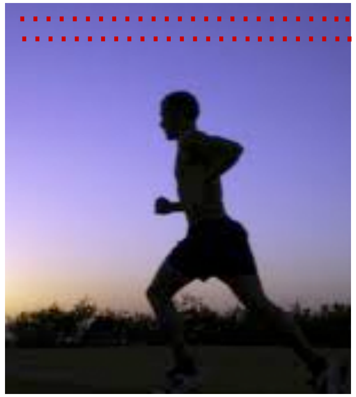
temporal coding example: instead of sending complete frame at i+1, send only differences from frame i

frame *i+1*

Multimedia: video

- CBR: (constant bit rate): video encoding rate fixed
- VBR: (variable bit rate): video encoding rate changes as amount of spatial, temporal coding changes
- examples:
  - MPEG 1 (CD-ROM) 1.5 Mbps
  - MPEG2 (DVD) 3-6 Mbps
  - MPEG4 (often used in Internet, < 1 Mbps)

spatial coding example: instead of sending *N* values of same color (all purple), send only two values: color value (*purple*) *and number of repeated values* (N)



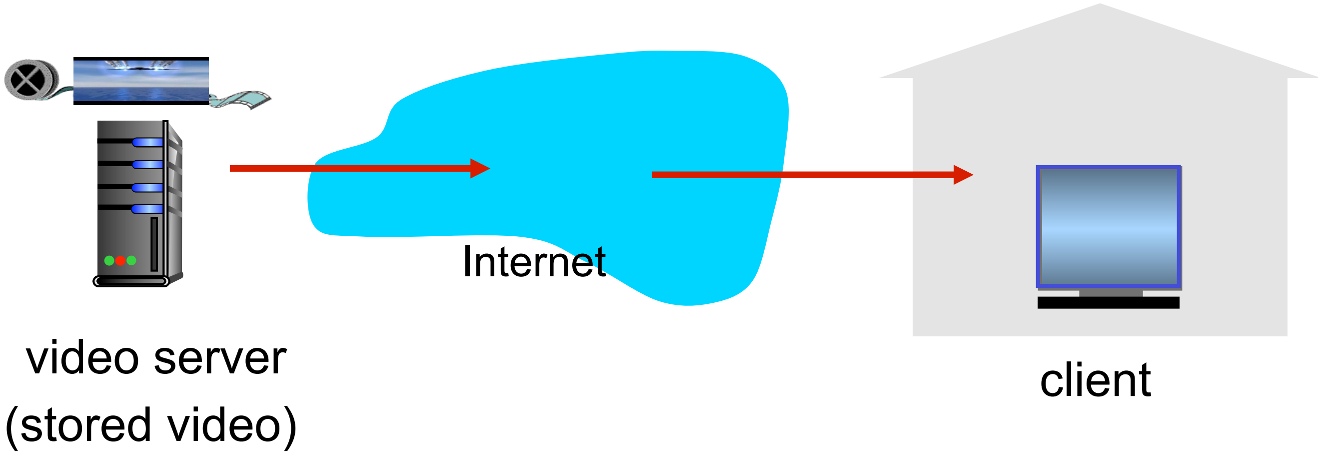
frame *i*

temporal coding example: instead of sending complete frame at i+1, send only differences from frame i

frame *i+1*

Streaming stored video:

simple scenario:

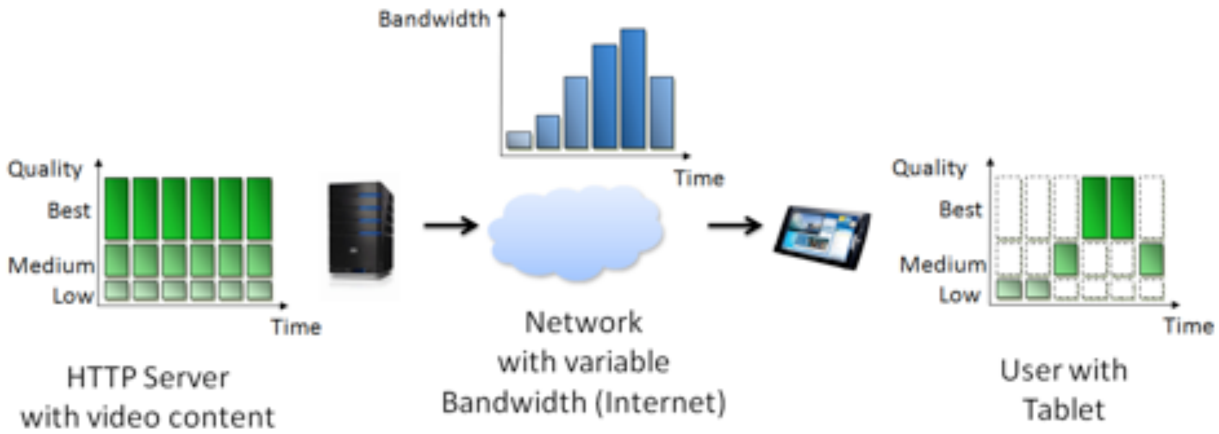


Streaming multimedia: DASH

- *DASH: D*ynamic, *A*daptive *S*treaming over *H*TTP
- *server:*
  - divides video file into multiple chunks
  - each chunk stored, encoded at different rates
  - *manifest file:* provides URLs for different chunks
- *client:*
  - periodically measures server-to-client bandwidth
  - consulting manifest, requests one chunk at a time
    - chooses maximum coding rate sustainable given current bandwidth
    - can choose different coding rates at different points in time (depending on available bandwidth at time)

Streaming multimedia: DASH

- *DASH: Dynamic, Adaptive Streaming over HTTP*
- *"intelligence"* at client: client determines
  - *when* to request chunk (so that buffer starvation, or overflow does not occur)
  - *what encoding rate* to request (higher quality when more bandwidth available)
  - *where* to request chunk (can request from URL server that is "close" to client or has high available bandwidth)



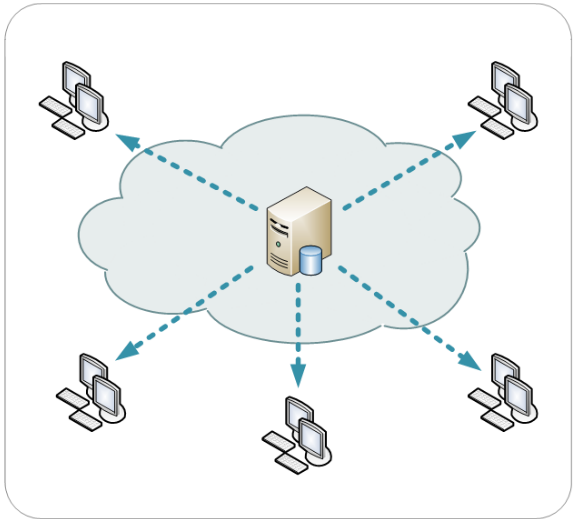
DASH implementations

- Android through ExoPlayer
- Samsung, LG, Sony, Philips Smart TVs
- Youtube
- Netflix
- JavaScript Implementaton for HTML5
- ….

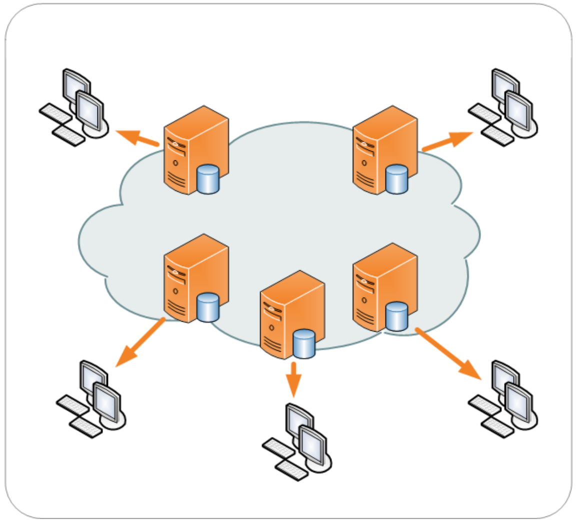
Content distribution networks

- Caching and replication as a service (amortise cost of infrastructure)
- Goal: bring content close to the user
- Large-scale distributed storage infrastructure (usually) administered by one entity
  - *e.g.*, Akamai has servers in 20,000+ locations
- Combination of (pull) caching and (push) replication
  - **Pull:** Direct result of clients' requests
  - **Push:** Expectation of high access rate

Single server



CDN



An example

```
bash-3.2$ dig www.mit.edu

; <<>> DiG 9.8.3-P1 <<>> www.mit.edu
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 27387
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 9, ADDITIONAL: 9

;; QUESTION SECTION:
;www.mit.edu.                   IN      A

;; ANSWER SECTION:
www.mit.edu.            1800    IN      CNAME   www.mit.edu.edgekey.net.
www.mit.edu.edgekey.net. 60     IN      CNAME   e9566.dscb.akamaiedge.net.
e9566.dscb.akamaiedge.net. 20   IN      A       23.77.150.125

;; AUTHORITY SECTION:
dscb.akamaiedge.net.    681     IN      NS      n4dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n5dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      a0dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n6dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n1dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n3dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n0dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n7dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n2dscb.akamaiedge.net.

;; ADDITIONAL SECTION:
a0dscb.akamaiedge.net.  7144    IN      AAAA    2600:1480:e800::c0
n0dscb.akamaiedge.net.  3048    IN      A       88.221.81.193
n1dscb.akamaiedge.net.  2752    IN      A       88.221.81.194
n2dscb.akamaiedge.net.  1380    IN      A       104.72.70.167
n3dscb.akamaiedge.net.  3048    IN      A       88.221.81.195
n4dscb.akamaiedge.net.  2810    IN      A       104.71.131.100
n5dscb.akamaiedge.net.  1326    IN      A       104.72.70.166
n6dscb.akamaiedge.net.  49      IN      A       104.72.70.174
n7dscb.akamaiedge.net.  2554    IN      A       104.72.70.175

;; Query time: 246 msec
;; SERVER: 129.94.172.11#53(129.94.172.11)
;; WHEN: Thu Mar  9 18:04:37 2017
;; MSG SIZE  rcvd: 463
```

Many well-known sites are hosted by CDNs. A simple way to check is using dig as shown here.

Can you find more examples?

Content distribution networks

- *challenge:* how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?

- *option 1:* single, large "mega-server"
  - single point of failure
  - point of network congestion
  - long path to distant clients
  - multiple copies of video sent over outgoing link

....quite simply: this solution *doesn't scale*

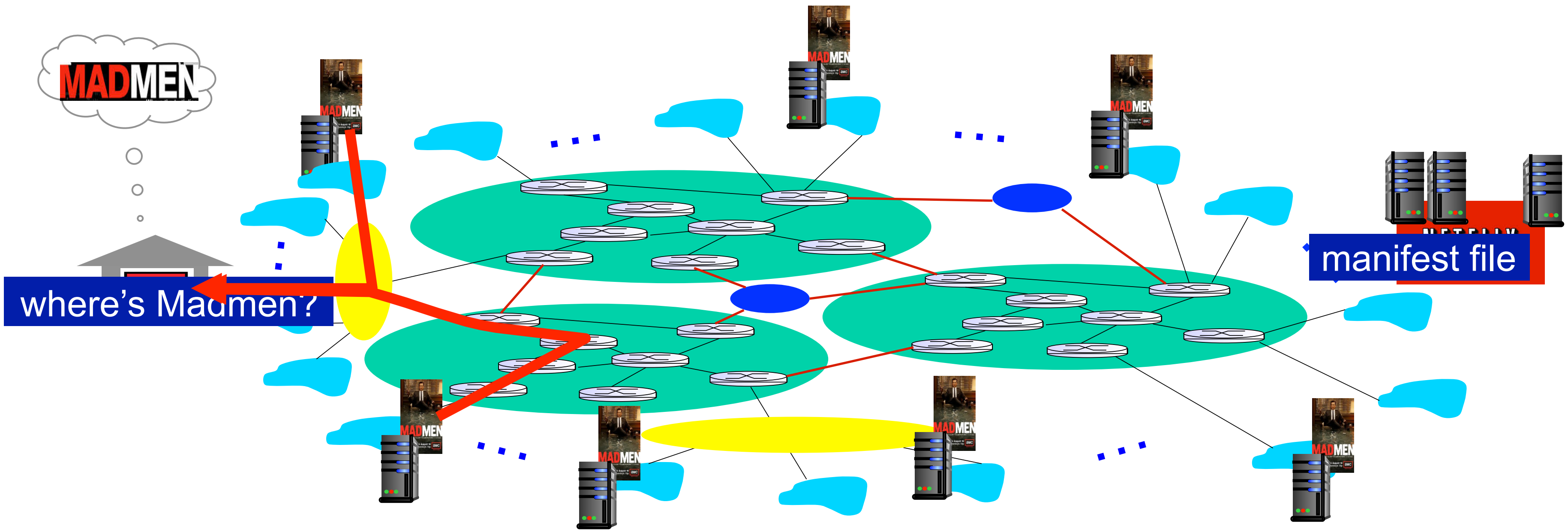
Content distribution networks

- *challenge:* how to stream content (selected from millions of videos) to hundreds of thousands of simultaneous users?

- *option 2:* store/serve multiple copies of videos at multiple geographically distributed sites *(CDN)*
  - *enter deep:* push CDN servers deep into many access networks
    - close to users
    - used by Akamai, thousands of locations
  - *bring home:* smaller number (10's) of larger clusters in POPs near (but not within) access networks
    - used by Limelight

Content Distribution Networks (CDNs)

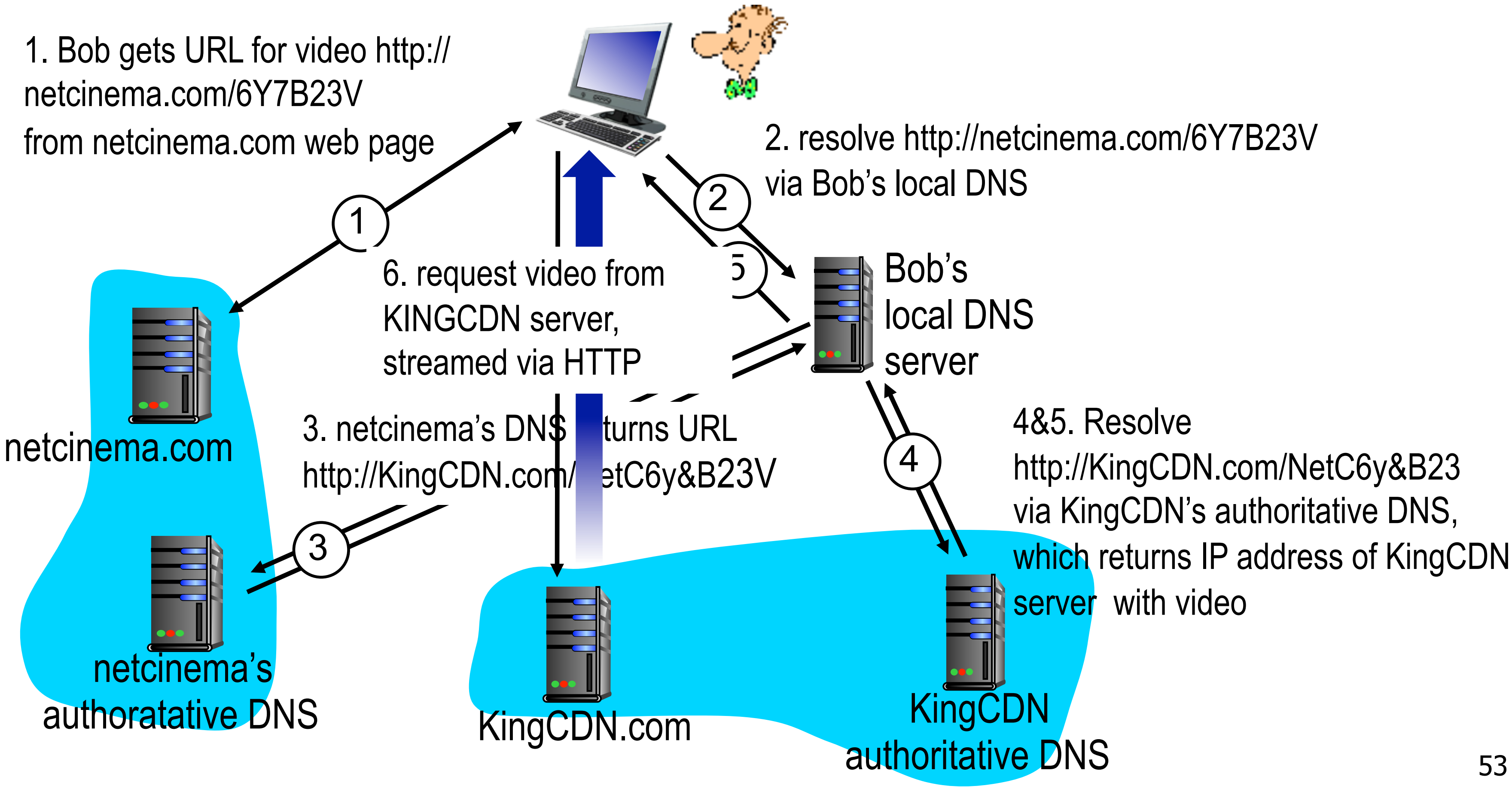
- CDN: stores copies of content at CDN nodes
  - e.g. Netflix stores copies of MadMen
- subscriber requests content from CDN
  - directed to nearby copy, retrieves content
  - may choose different copy if network path congested



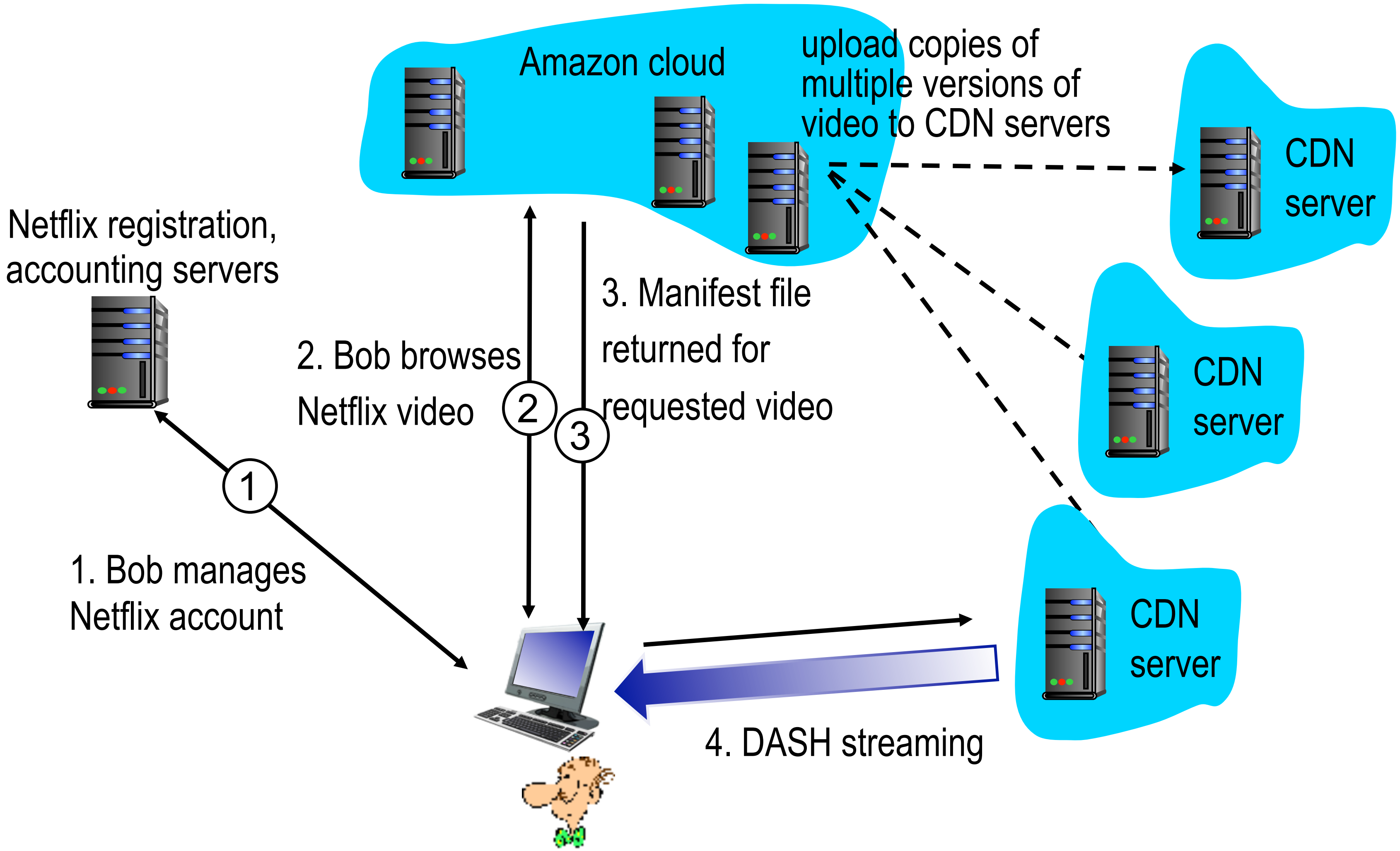
CDN content access: a closer look

Bob (client) requests video http://netcinema.com/6Y7B23V

- video stored in CDN at http://KingCDN.com/NetC6y&B23V



Case study: Netflix



2. Application Layer: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 electronic mail
- SMTP, POP3, IMAP

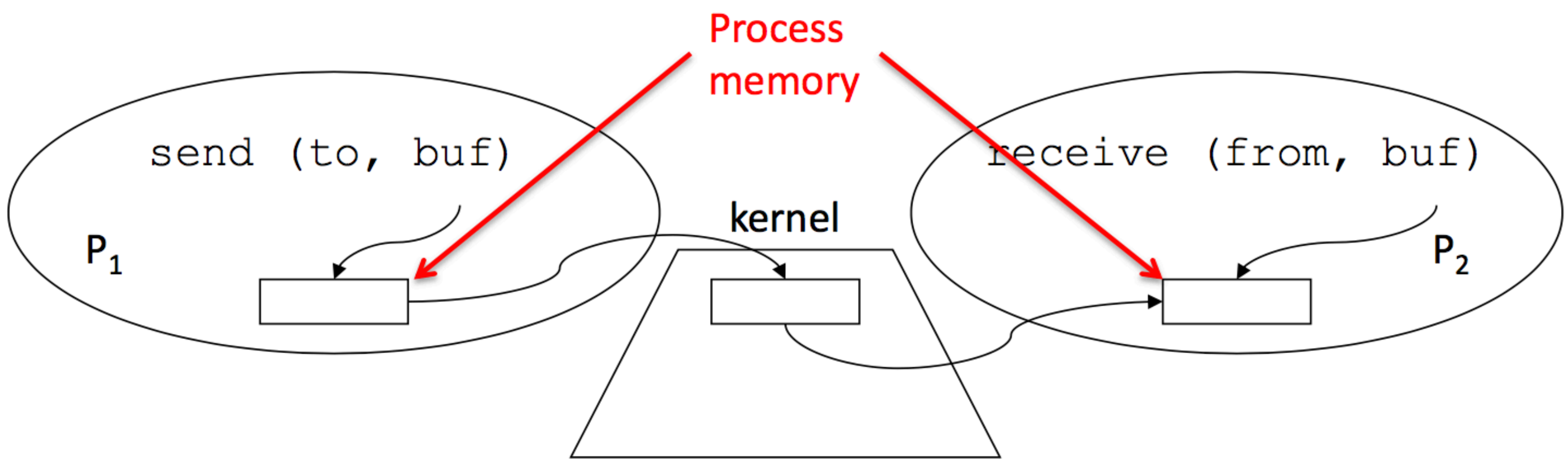
2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks (CDNs)

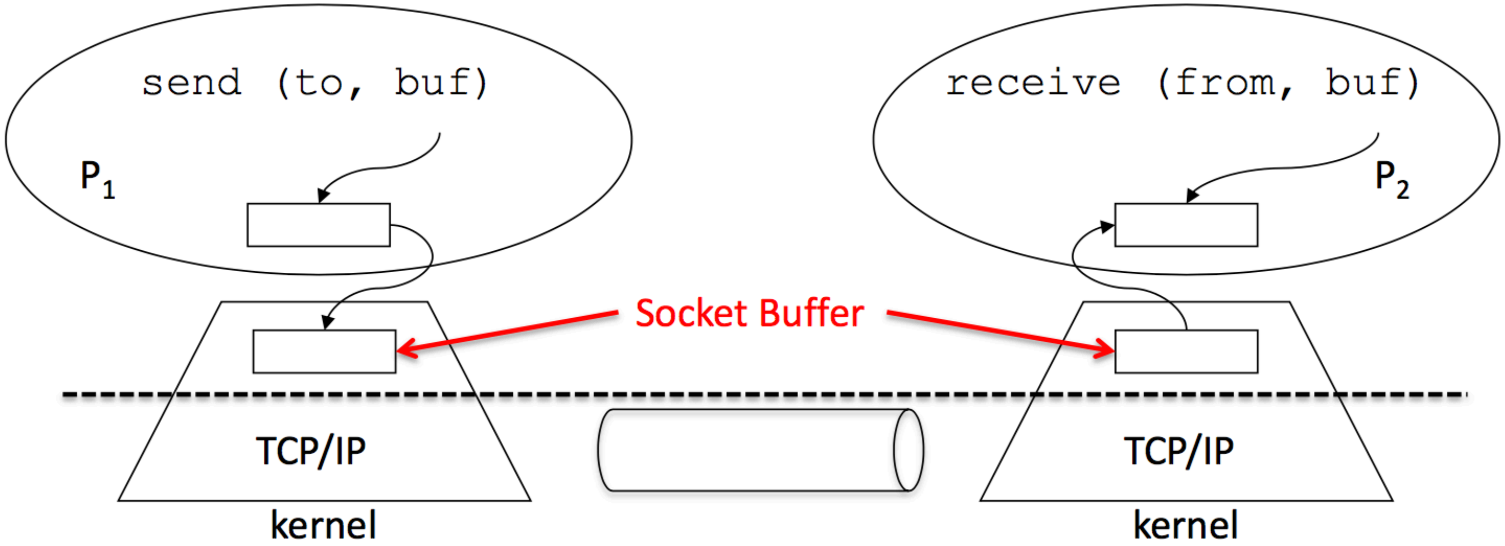
2.7 socket programming with UDP and TCP

Message Passing (local)



- Operating system mechanism for IPC
  - `send (destination, message_buffer)`
  - `receive (source, message_buffer)`
- Data transfer: in to and out of kernel message buffer

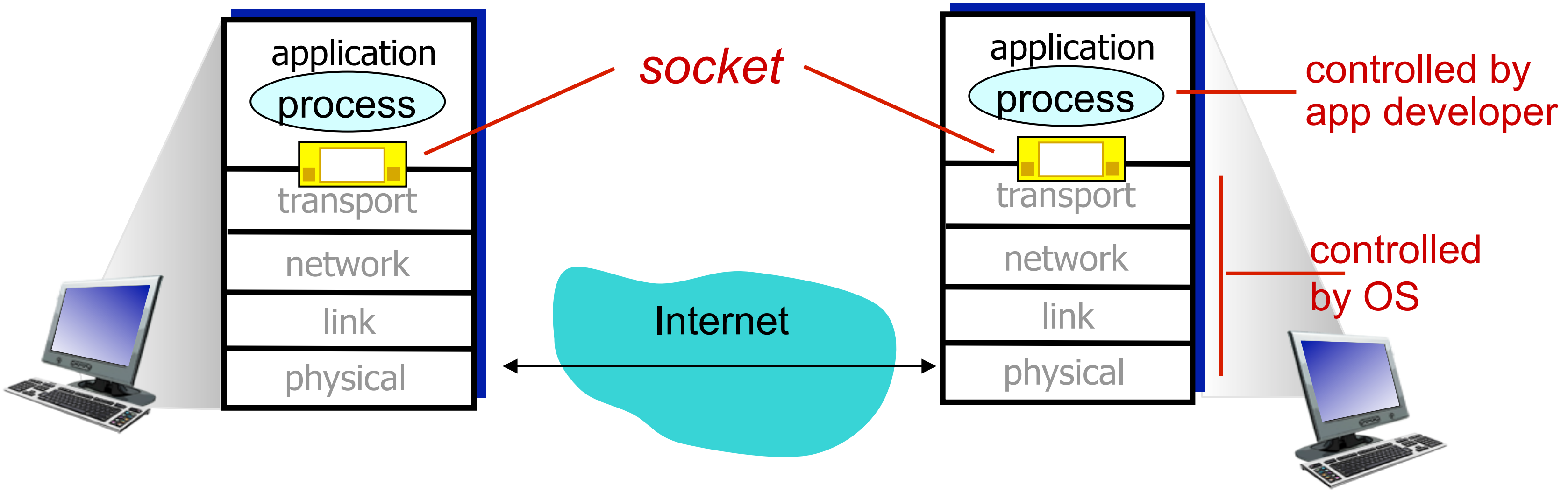
Message Passing (network)



- Data transfer
  - Copy to/from OS socket buffer
  - Extra step across network: hidden from applications

Socket programming

Socket: a door between application process and end-end-transport protocol (UDP or TCP)



Sockets Specify Transport Services

- Sockets define the interfaces between an application and the transport layer
- Applications choose the type of transport layer by choosing the type of socket
  - UDP Sockets – called DatagramSocket in Java, SOCK_DGRAM in C
  - TCP Sockets – called Socket/ServerSocket in Java, SOCK_STREAM in C
- Client and server agree on the type of socket, the server port number and the protocol

How do clients and servers communicate?

API: application programming interface

- defines interface between application and transport layer
- socket: Internet API
  - two processes communicate by sending data into socket, reading data out of socket

Q: how does a process "identify" the other process with which it wants to communicate?

- IP address of host running other process
- "port number" - allows receiving host to determine to which local process the message should be delivered

Languages and Platforms

Socket API is available for many languages on many platforms:

- C, Java, Perl, Python,…
- *nix, Windows,…

Socket Programs written in any language and running on any platform can communicate with each other!

Writing communicating programs in different languages is a good exercise

Lean how to build client/server application that communicate using sockets

Socket Programming is Easy

- Create socket much like you open a file
- Once open, you can read from it and write to it
- Operating System hides most of the details

System calls for a socket:

Socket()	Read()
Bind()	Write()
Connect()	Close()
Listen()	
Accept()	

Decisions

- Before you go to write socket code, decide
  - Do you want a TCP-style reliable, full duplex, connection oriented channel? Or do you want a UDP-style, unreliable, message oriented channel?
  - Will the code you are writing be the client or the server?
    - Client: you assume that there is a process already running on another machines that you need to connect to.
    - Server: you will just start up and wait to be contacted

OVERVIEW: TCP vs UDP

TCP service:

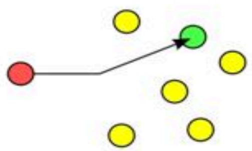
- *connection-oriented:* setup required between client, server
- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing or minimum bandwidth guarantees

UDP service:

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

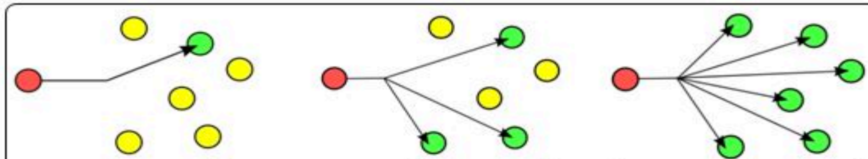
OVERVIEW: TCP vs UDP

- **Slower but reliable transfers**
- **Typical applications:**
  - Email
  - Web browsing



unicast

- **Fast but non-guaranteed transfers ("best effort")**
- **Typical applications:**
  - VoIP
  - Music streaming



unicast **multicast** **broadcast**

Pseudo code TCP server

Create socket (doorbellSocket)

Bind socket to a specific port where clients can contact you

Register with the kernel your willingness to listen that on socket for client to contact you

Loop

Listen to doorbell Socket for an incoming connection, get a connectSocket

Read and Write Data Into connectSocket to Communicate with client

Close connectSocket

End Loop

Close doorbellSocket

Pseudo code TCP client

Create socket, connectSocket

Do an active connect specifying the IP address and port number of server

Read and Write Data Into connectSocket to Communicate with server

Close connectSocket

Example: Java server (TCP)

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
   {
    String clientSentence;
    String capitalizedSentence;

    ServerSocket doorbellSocket = new ServerSocket(6789);

    while(true) {

        Socket connectSocket = doorbellSocket.accept();

       BufferedReader inFromClient =
         new BufferedReader(new
         InputStreamReader(connectSocket.getInputStream()));
```

Example: Java server (TCP), cont

Create output stream, attached to socket

Read in line from socket

Write out line to socket

End of while loop, loop back and wait for another client connection

```
DataOutputStream  outToClient =
  new DataOutputStream(connectSocket.getOutputStream());

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';

outToClient.writeBytes(capitalizedSentence);
   }
  }
 }
```

Example: Java client (TCP)

```
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

        Socket connectSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
          new DataOutputStream(connectSocket.getOutputStream());
```

Create input stream

Create client socket, connect to server

Create output stream attached to socket

Example: Java client (TCP), cont.

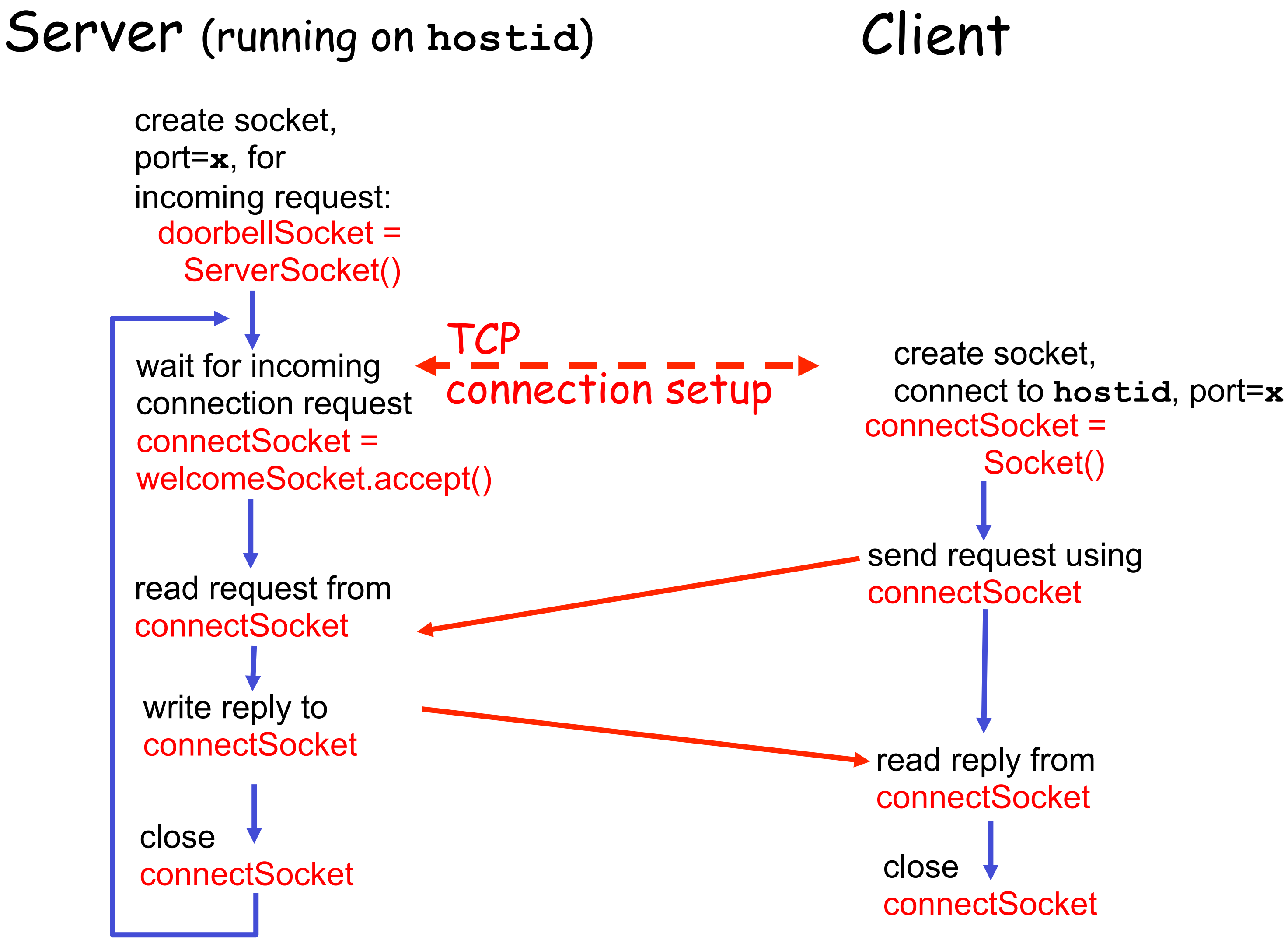
Create input stream attached to socket

Send line to server

Read line from server

```
BufferedReader inFromServer =
  new BufferedReader(new
  InputStreamReader(connectSocket.getInputStream()));

sentence = inFromUser.readLine();

outToServer.writeBytes(sentence + '\n');

modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

connectSocket.close();

  }
}
```

Client/server socket interaction: TCP (Java)



Concurrent/Multithreaded TCP Servers

- What good is the doorbell socket? Can't accept new connections until call accept again anyway?
- Benefit comes in ability to hand off processing to another process or thread
  - Parent process creates the "door bell" or "welcome" socket on well-known port and waits for clients to request connection
  - When a client does connect, fork off a child process or pass to another thread to handle that connection so that parent can return to waiting for connections as soon as possible

Pseudo code concurrent TCP server

```
Create socket doorbellSocket
Bind
Listen
Loop
   Accept the connection, connectSocket
   Fork
   If I am the child
         Read/Write connectSocket
         Close connectSocket
         exit
EndLoop
Close doorbellSocket
```

Socket programming with UDP

UDP: very different mindset than TCP

- no connection just independent messages sent
- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received datagram to know who to respond to

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

Pseudo code UDP client

Create socket

Loop

(Send Message To Well-known port of server)+

(Receive Message From Server)

Close Socket

Pseudo code UDP server

Create socket

Bind socket to a specific port where clients can contact you

Loop

    (Receive UDP Message from client x)+

    (Send UDP Reply to client x)*

Close Socket

Example: Java server (UDP)

Create datagram socket at port 9876

Create space for received datagram

Receive datagram

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
    {

      DatagramSocket serverSocket = new DatagramSocket(9876);

      byte[] receiveData = new byte[1024];
      byte[] sendData  = new byte[1024];

      while(true)
        {

          DatagramPacket receivePacket =
             new DatagramPacket(receiveData, receiveData.length);
          serverSocket.receive(receivePacket);
```

Example: Java server (UDP), cont

Get IP addr port #, of sender

Create datagram to send to client

Write out datagram to socket

```
String sentence = new String(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

        String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();

DatagramPacket sendPacket =
  new DatagramPacket(sendData, sendData.length, IPAddress,
                    port);

serverSocket.send(sendPacket);
      }
    }
  }
```

End of while loop, loop back and wait for another datagram

Example: Java client (UDP)

```
import java.io.*;
import java.net.*;

class UDPClient {
  public static void main(String args[]) throws Exception
  {

    BufferedReader inFromUser =
      new BufferedReader(new InputStreamReader(System.in));

    DatagramSocket clientSocket = new DatagramSocket();

    InetAddress IPAddress = InetAddress.getByName("hostname");

    byte[] sendData = new byte[1024];
    byte[] receiveData = new byte[1024];

    String sentence = inFromUser.readLine();

    sendData = sentence.getBytes();
```

Create input stream

Create client socket, DatagramSocket, but no port

Translate hostname to IP address using DNS

Example: Java client (UDP), cont.

Create datagram with data-to-send, length, IP addr, port

Send datagram to server

Read datagram from server

```
DatagramPacket sendPacket =
  new DatagramPacket(sendData, sendData.length, IPAddress, 9876);

clientSocket.send(sendPacket);

DatagramPacket receivePacket =
  new DatagramPacket(receiveData, receiveData.length);

clientSocket.receive(receivePacket);

String modifiedSentence =
   new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
}

}
```

Client/server socket interaction: UDP

Server (running on `hostid`)

create socket,
port=**x**, for
incoming request:
serverSocket =
DatagramSocket()

read request from
serverSocket

write reply to
serverSocket
specifying client
host address,
port umber

Client

create socket,
clientSocket =
DatagramSocket()

Create, address (**`hostid, port=x,`**
send datagram request
using clientSocket

read reply from
clientSocket

close
clientSocket

Socket programming

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.