Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

DNS: Domain Name System

people: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

- IP address (32 bit) - used for addressing datagrams
- "name", e.g., cs.umass.edu - used by humans

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

need translation, need a directory service?

Domain Name System (DNS):

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol:* hosts, DNS servers communicate to *resolve* names (address/name translation)

* runs over UDP at TL, port # 53

* UNIX servers with Berkeley Internet Name Domain (BIND) software

DNS: services

connection to www.ankara.edu.tr

- Writing the URL to the add. bar of browser
- Client side of DNS app is run
- DNS client sends a query with the hostname to a DNS server
- DNS client receives a reply with IP address
- TCP connection between browser and HTTP server process (at port # 80)

Any additional delay except for the packet delays?

Caching in nearby DNS servers?

DNS: services

- Service 1: translating host names to IP add. - employed by other AL protocols (e.g. HTTP & SMTP)
- Service 2: host aliasing

  * canonical, alias names

  a complicated host name: relay1.west-coast.enterprise.com

  one or more aliases: enterprise.com, www.enterprise.com

  if aliases are there then canonical host name

  DNS can be used by an app to get canonical for an alias hostname

  * mail server aliasing

  relay1.west-coast.yahoo.com (MX record)

  yahoo.com

  mail server and a Web server to have identical aliases

DNS: services

- load distribution among replicated servers
  - replicated Web servers: many IP addresses correspond to one name
  - When clients make a DNS query, entire set of IP addresses are sent (ordering of the addresses is rotated within each reply)

Thinking about the DNS

humongous distributed database:

- ~ billion records, each simple

handles many *trillions* of queries/day:

- *many* more reads than writes
- *performance matters:* almost every Internet transaction interacts with DNS - msecs count!

organizationally, physically decentralized:

- millions of different organizations responsible for their records

Thinking about the DNS

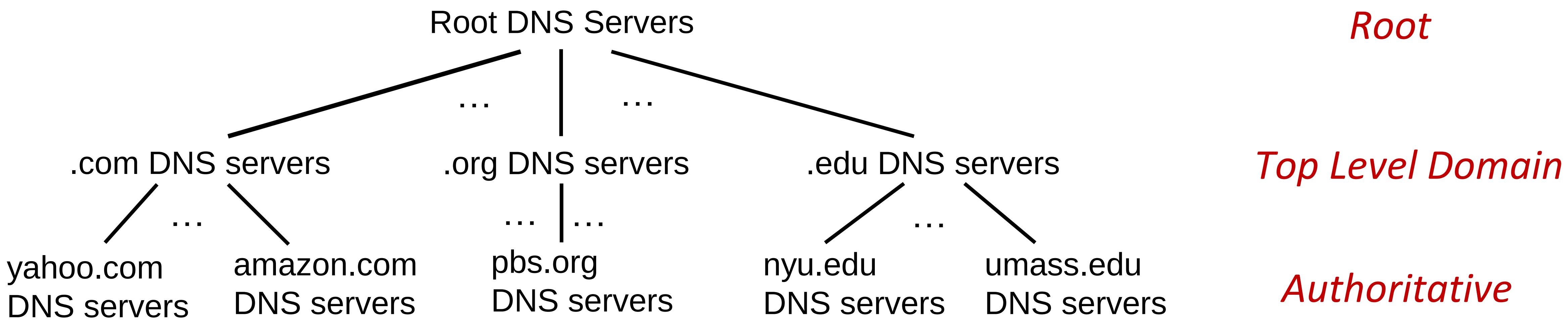
a simple design - centralized: only one DNS server containing all the mappings

- fits today's Internet?
- what happens if it crashes? (single point of failure)
- can it handle all queries?
- can be "close to" all querying clients? significant delays?
- what about maintenance? should it be updated for every new host?

centralized or distributed?

- lots of servers organized in a hierarchy, distributed around the world
- (host name-IP address) mappings are distributed across them
- three classes: root, top-level domain (TLD), and authoritative

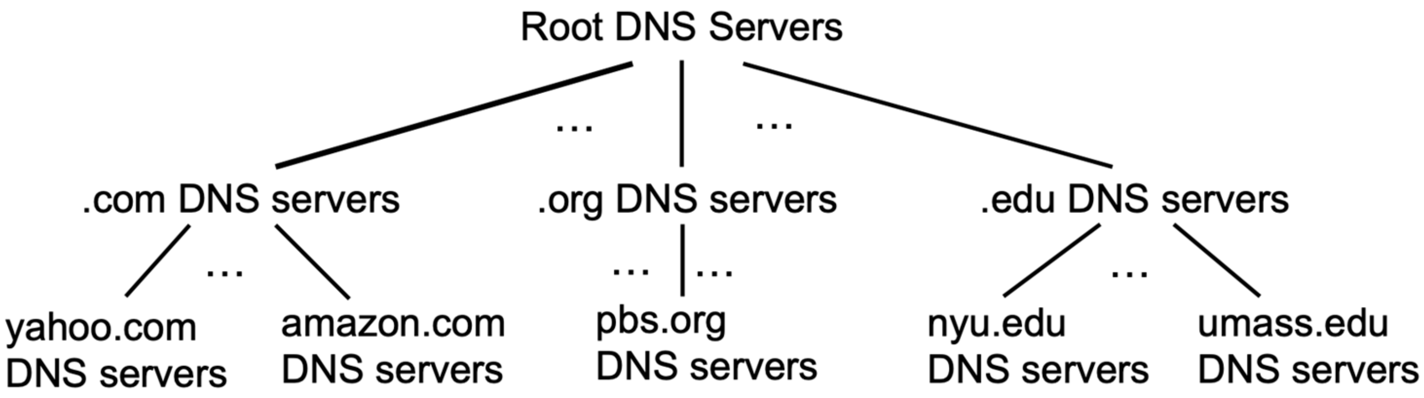
DNS: a distributed, hierarchical database

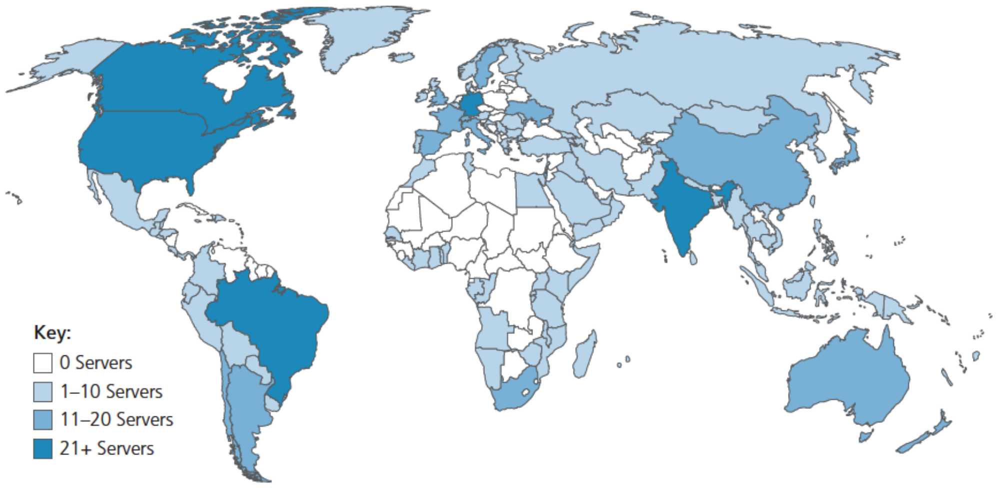


Client wants IP address for www.amazon.com

- 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: root name servers



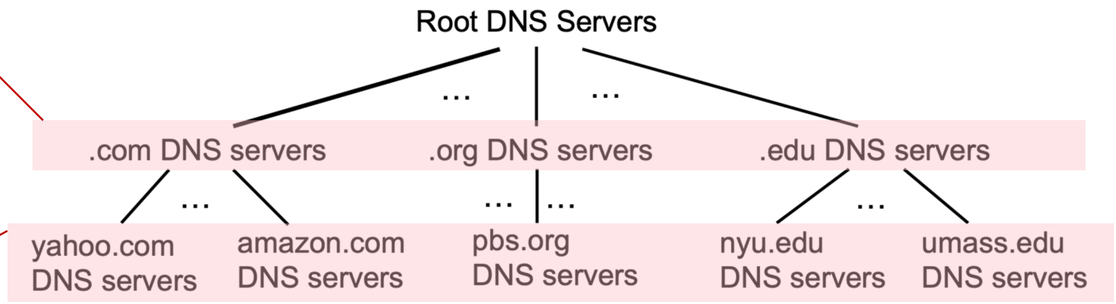


- copies of 13 different root servers
- more than 1000 root servers all over the world
- managed by 12 different organizations
- coordinated by Internet Assigned Numbers Authority
- provide TLD IP add.

Top-Level Domain, 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.: .cn, .uk, .fr, .ca, .jp
- Verisign: authoritative registry for .com
- Educause: .edu TLD
- provide IP add. for authoritative ones



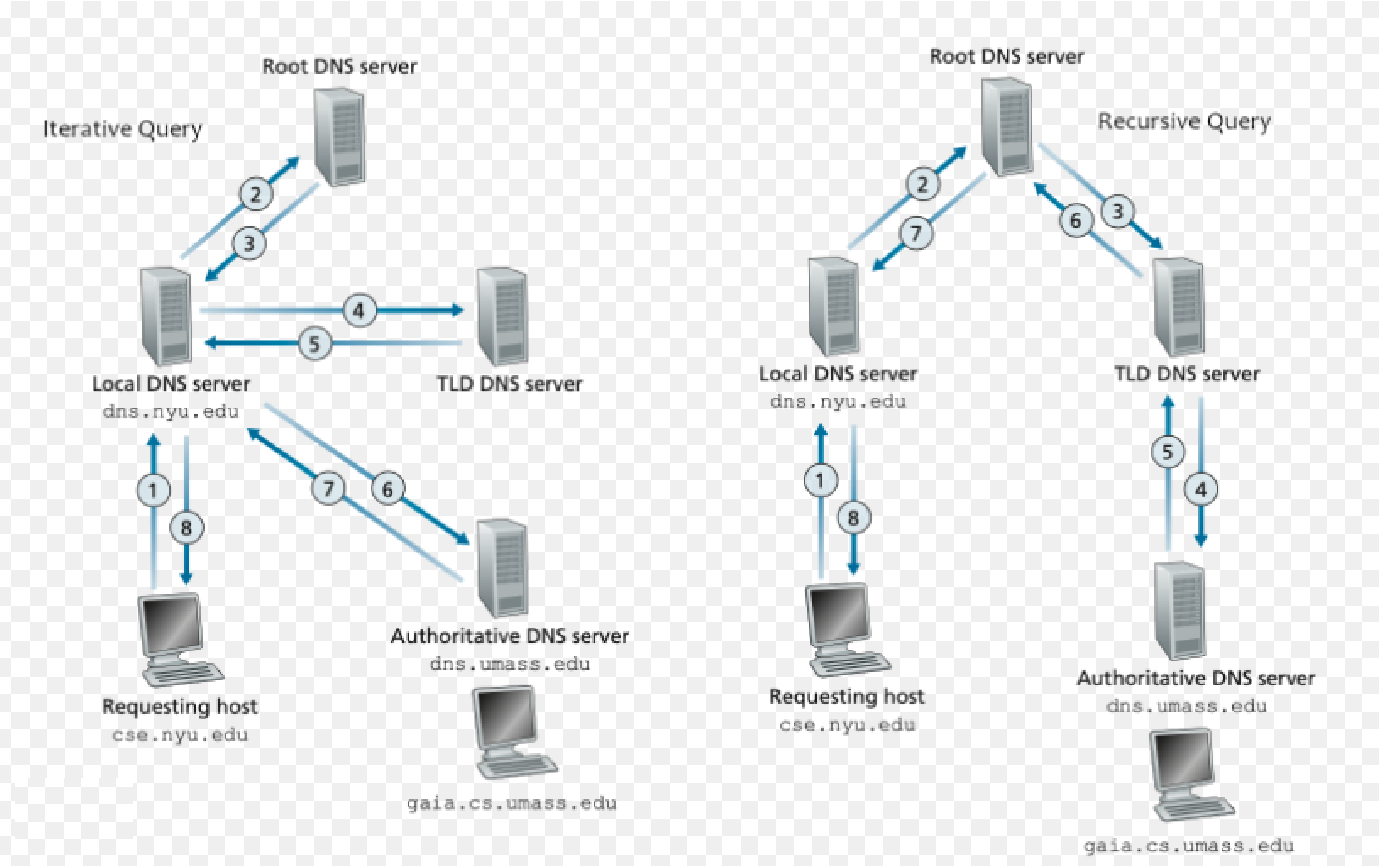
authoritative DNS servers:

- owned by organizations/companies, provides authoritative hostname to IP mappings (records)
- can be maintained by organization or service provider

Local DNS name servers

- another one not in the hiearchy
- owned by ISP’s (default name server)
- close to host
- when host makes DNS query, it is sent to its *local* DNS server
  - Local DNS server returns reply, answering:
    - from its local cache of recent name-to-address translation pairs (possibly out of date!)
    - forwarding request into DNS hierarchy for resolution
  - each ISP has local DNS name server; to find yours:
    - MacOS: `% scutil --dns`
    - Windows: `>ipconfig /all`

DNS name resolution: iterated & recursive query



Iterative query:
client communicates directly with each DNS server

Recursive query:
- one DNS server communicates with several other DNS servers on behalf of the client.
- puts burden of name resolution on contacted name server

Caching DNS Information

- an important feature
- once (any) name server learns mapping, it *caches* mapping, and *immediately* returns a cached mapping in response to a query
  - caching improves response time, reduces # of DNS messages
  - cache entries timeout (disappear) after some time (TTL)
  - TLD servers typically cached in local name servers
- cached entries may be *out-of-date*
  - if named host changes IP address, may not be known Internet-wide until all TTLs expire!
  - *best-effort name-to-address translation!*

DNS records

DNS: distributed database storing resource records **(RR)**

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

ttl: determines when a resource should be removed from a cache

type=A

- `name` is hostname (e.g. relay1.bar.foo.com)
- `value` is IP address (e.g. 145.37.93.126)

type=NS

- `name` is domain (e.g., foo.com)
- `value` is hostname of authoritative name server for this domain (e.g. dns.foo.com)
- is used to route queries

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 "canonical" (the real) name of SMTP mail server associated with `name` (alias)

DNS records

Question

- If a DNS server is **authoritative** for a particular hostname, then the DNS server will contain which type of record for the hostname?

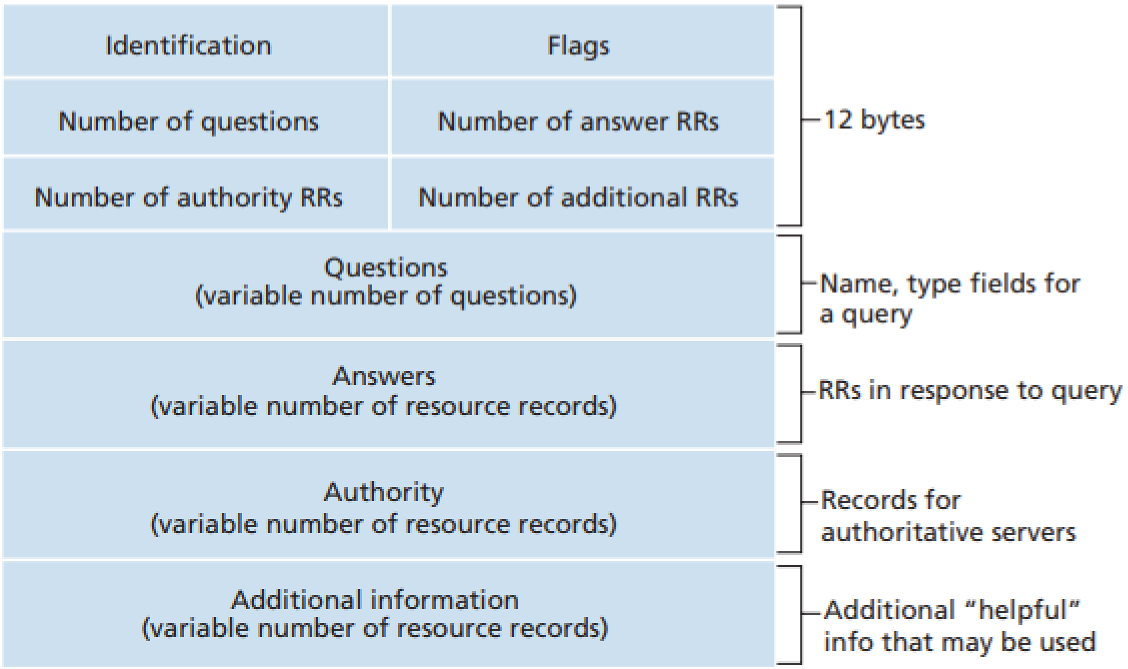
- If a DNS server is **not authoritative** for a hostname, then the server will contain which type of record(s)?

DNS protocol messages

DNS *query* and *reply* messages, both have same *format:*

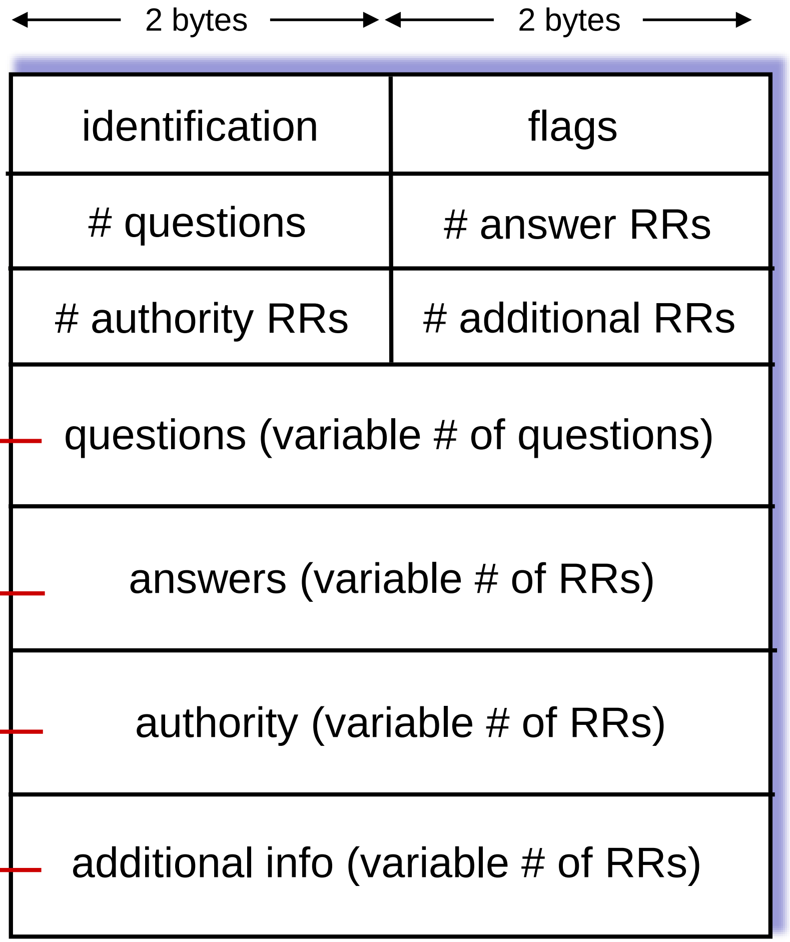
message header:

- identification: 16 bit # for query, reply to query uses same #
- flags (1 bit):
  - query (0) or reply (1)
  - recursion desired (in query, 1)
  - recursion available (reply, 1)
  - reply is authoritative (in reply, 1)



DNS protocol messages

DNS *query* and *reply* messages, both have same *format:*



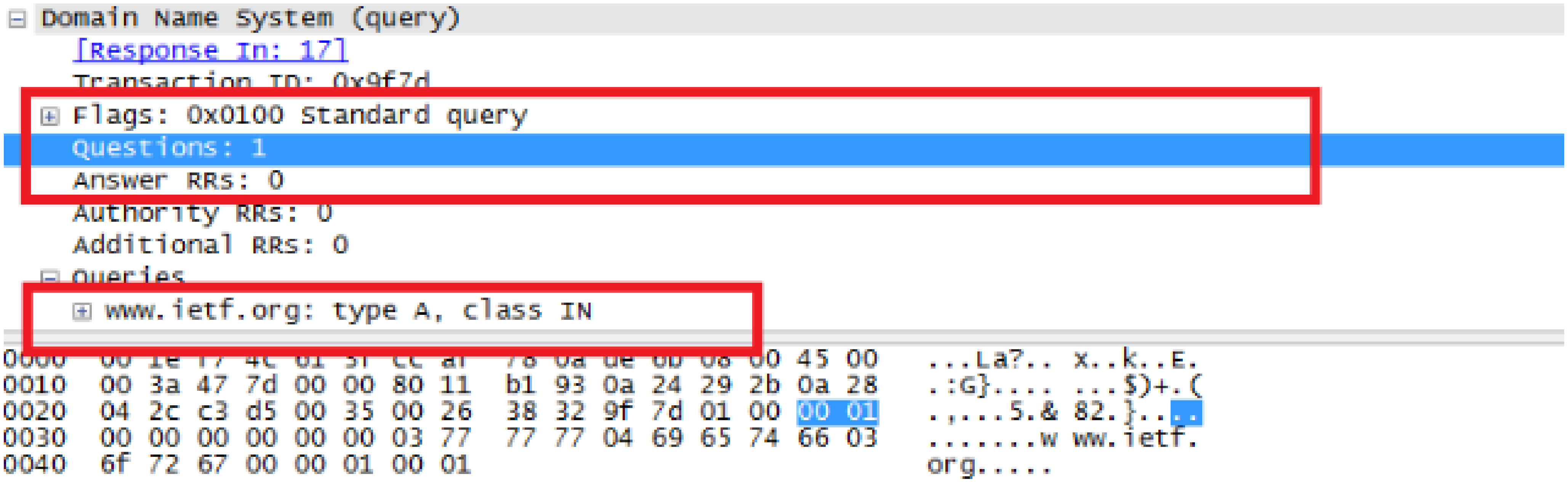
name, type fields for a query

RRs in response to query (name, value, TTL)
can be multiple RRs (multiple IP addresses)

records for authoritative servers

additional “ helpful” info that may be used

DNS protocol messages



DNS protocol messages

```
⊞ Flags: 0x8180 Standard query response, No error
  Questions: 1
  Answer RRs: 1
  Authority RRs: 6
  Additional RRs: 11
⊟ Queries
  ⊟ www.ietf.org: type A, class IN
      Name: www.ietf.org
      Type: A (Host address)
⊟ Answers
  ⊞ www.ietf.org: type A, class IN, addr 64.170.98.30
⊟ Authoritative nameservers
  ⊞ ietf.org: type NS, class IN, ns ns1.yyz1.afilias-nst.info
  ⊞ ietf.org: type NS, class IN, ns ns0.ietf.org
  ⊞ ietf.org: type NS, class IN, ns ns1.sea1.afilias-nst.info
  ⊞ ietf.org: type NS, class IN, ns ns1.ams1.afilias-nst.info
  ⊞ ietf.org: type NS, class IN, ns ns1.mia1.afilias-nst.info

000  cc af 78 0a de 6b 00 1e  f7 4c 61 3f 08 00 45 00   ..x..k.. .La?..E.
010  01 cb 63 b4 40 00 7e 11  55 cb 0a 28 04 2c 0a 24   ..c.@.~. U..(.,.$
020  29 2b 00 35 c3 d5 01 b7  1a 58 9f 7d 81 80 00 01   )+.5.... .X.}....
030  00 01 00 06 00 0b 03 77  77 77 04 69 65 74 66 03   .......w ww.ietf.
040  6f 72 67 00 00 01 00 01  c0 0c 00 01 00 01 00 00   org..... ........
050  07 08 00 04 40 aa 62 1e  c0 10 00 02 00 01 00 00   ....@.b. ........
```

Getting your info into the DNS

example: new startup "Network Utopia"

- register name networkuptopia.com at *DNS registrar* (a commercial entity e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts NS, A RRs into .com TLD server:

    `(networkutopia.com, dns1.networkutopia.com, NS)`
    `(dns1.networkutopia.com, 212.212.212.1, A)`
- create authoritative server locally with IP address `212.212.212.1`
  - type A record for www.networkuptopia.com
  - type MX record for networkutopia.com

DNS security

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

Spoofing attacks

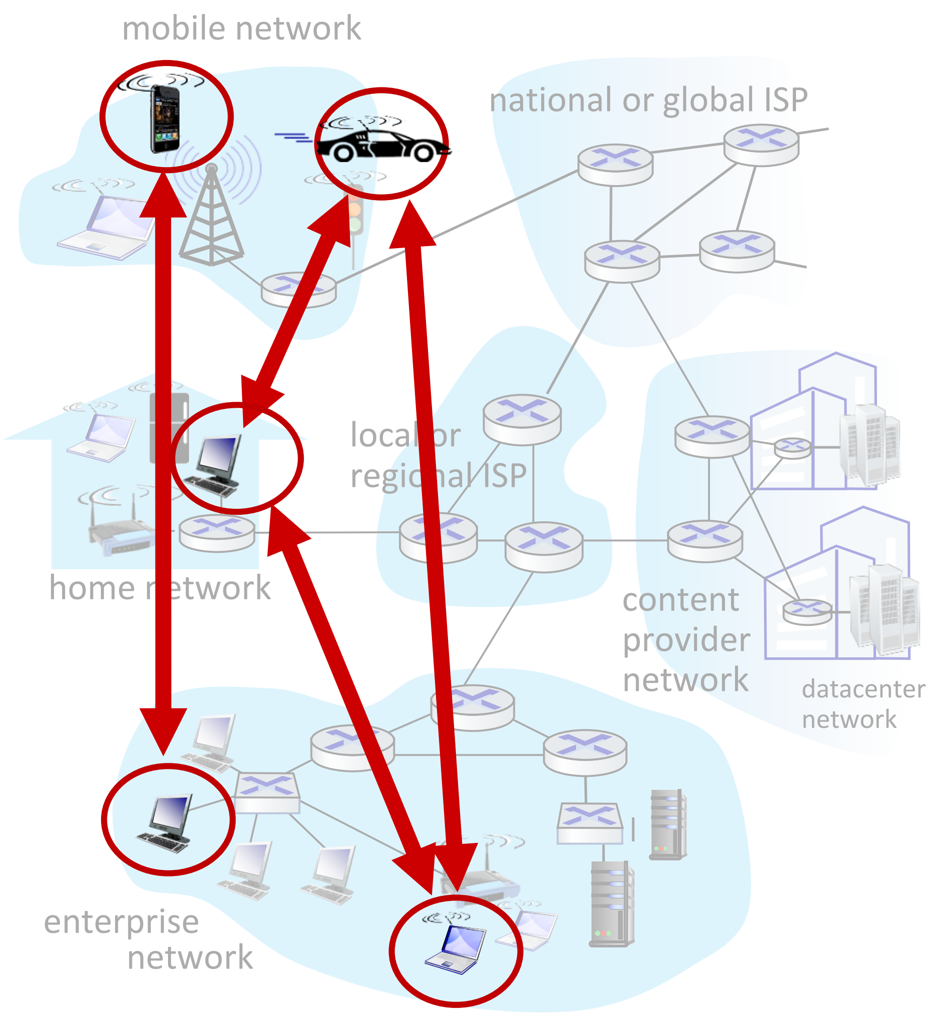
- intercept DNS queries, returning bogus replies
  - DNS cache poisoning
  - RFC 4033: DNSSEC (security extensions): strengthens authentication by PKI, DNS data is encrypted

Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

Peer-to-peer (P2P) architecture

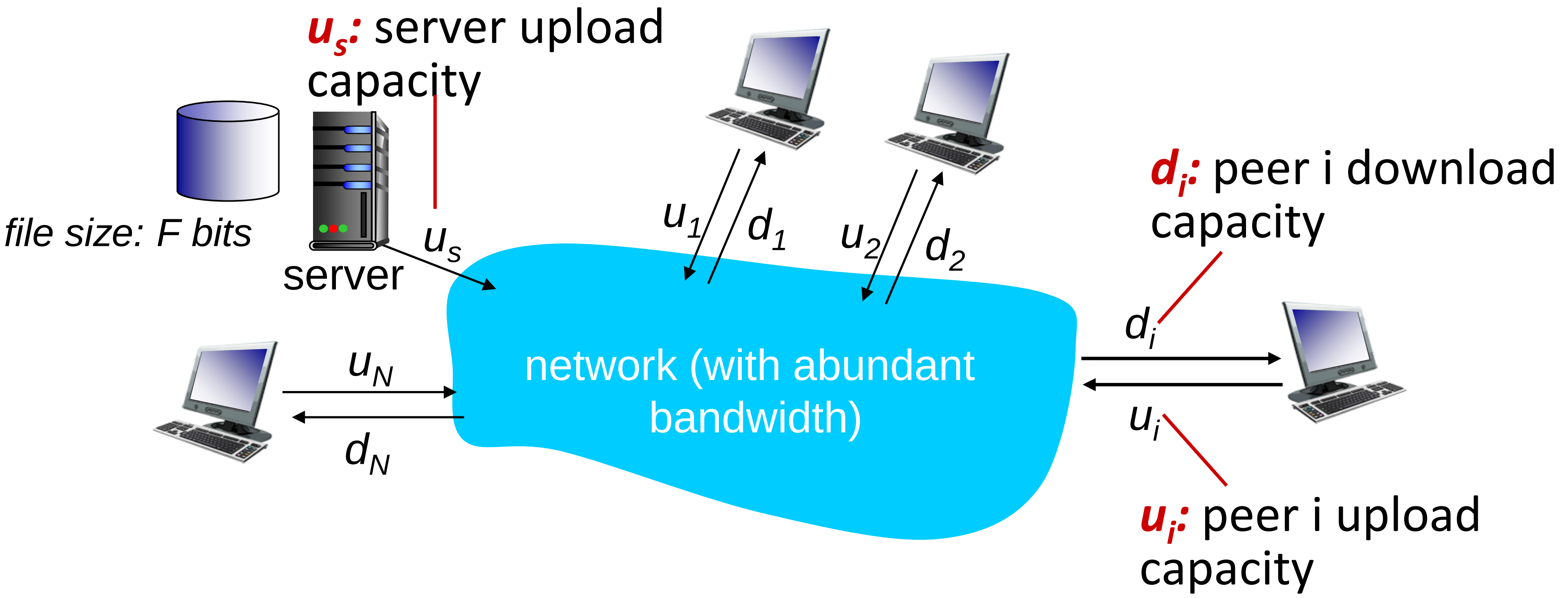
- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, and new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- examples: P2P file sharing (BitTorrent), streaming (KanKan), VoIP (Skype)



File distribution: client-server vs P2P

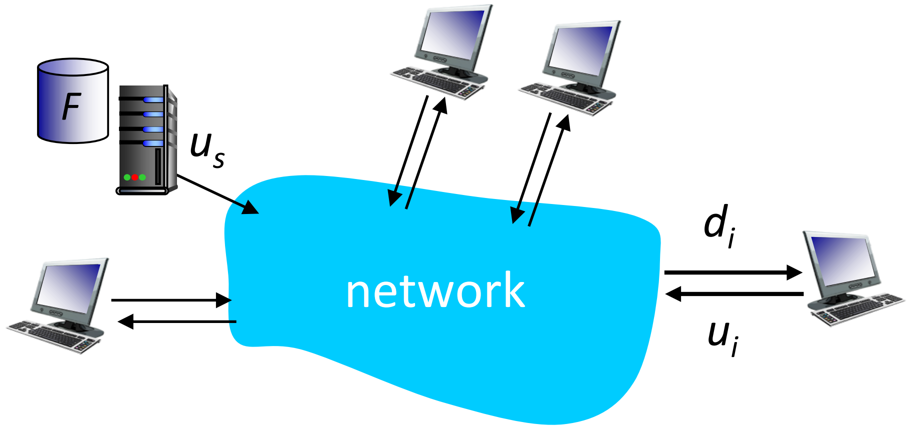
Q: how much time to distribute file (size *F*) from one server to *N peers*?

- peer upload/download capacity is limited resource



File distribution time: client-server

- *server transmission:* must sequentially send (upload) *N* file copies to each peer:
  - time to send one copy: F/u_s
  - time to send *N* copies: NF/u_s



- *client:* each client must download file copy
  - d_{min} = min client download rate
  - client download time at least: F/d_{min}

time to distribute file to N clients using client-server approach $> max\{NF/u_s, F/d_{min}\}$

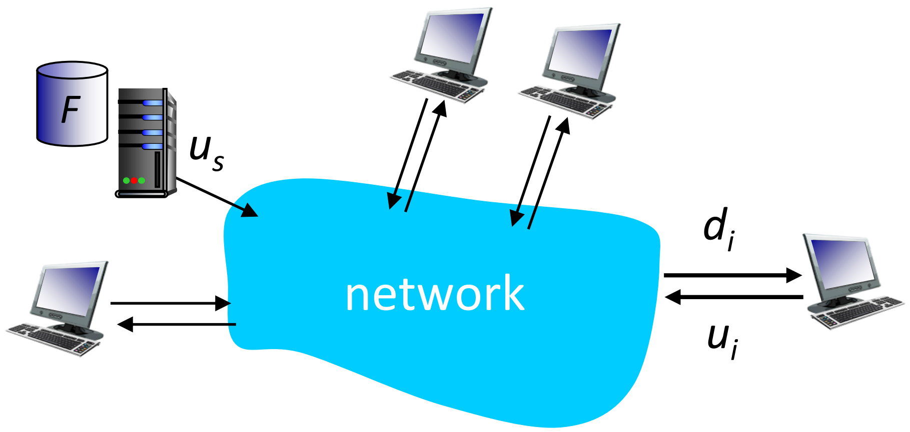
File distribution time: P2P

- each peer can assist the server in distributing the file. When a peer receives some file data, it can use its own upload capacity to redistribute the data to other peers.

- *server transmission:* must upload at least one copy:
  - time to send one copy: F/u_s

- *client:* each client must download file copy
  - client download time at least: F/d_{min}

- *clients:* as aggregate must download NF bits
  - max upload rate is $u_s + \Sigma u_i$

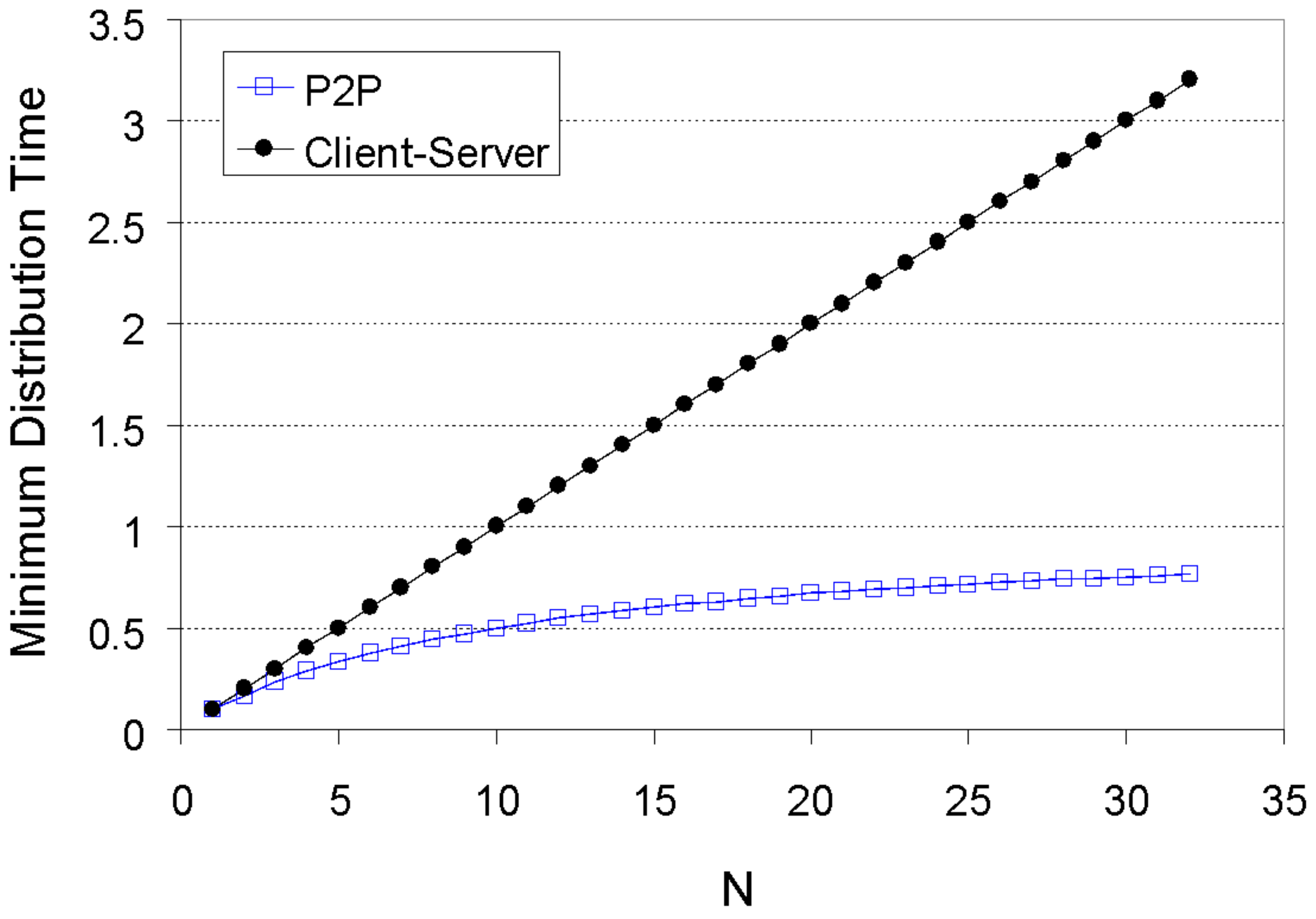


time to distribute F to N clients using P2P approach

$$> max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

Client-server vs. P2P: example

client upload rate = u, F/u = 1 hour, $u_s = 10u$, $d_{min} \geq u_s$



as the number of peers increases:

* client-server : distribution time increases linearly and without bound
* p2p: distribution time is always less than C-S

Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks (e.g. Netflix, YouTube and Amazon Prime)
- socket programming with UDP and TCP

Video Streaming and CDNs: context

- stream video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube, Amazon Prime: 80% of residential ISP traffic (2020)
- *challenge:* scale - how to reach ~1B users?
- *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, compress
  - 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)

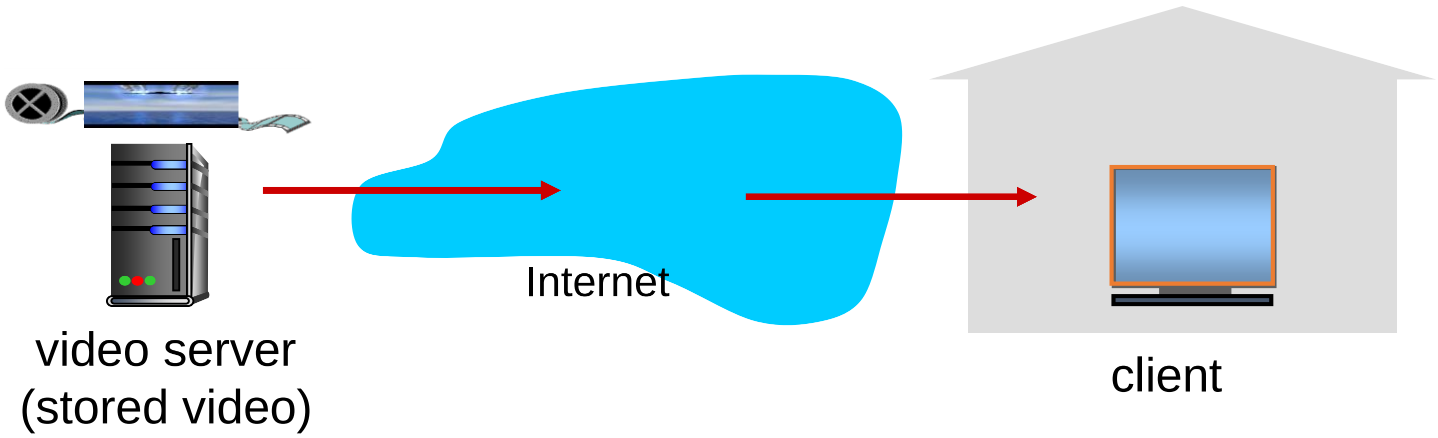
frame *i*

frame *i+1*

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

Streaming stored video

simple scenario:



Challenges:
* server-to-client bandwidth variations stem from changing network congestion levels
* packet loss, delay due to congestion will delay playout or result in poor video quality

Streaming multimedia:

HTTP streaming:

- *video is stored at an HTTP server*
- *a TCP connection, http request, and responses*
- *on client, bytes are collected in a client application buffer*
- *when number of bytes in buffer exceeds a predetermined threshold, the client application begins playback*

- ***shortcoming**: all clients receive the same encoding of the video, but large variations of available bandwidth possible*

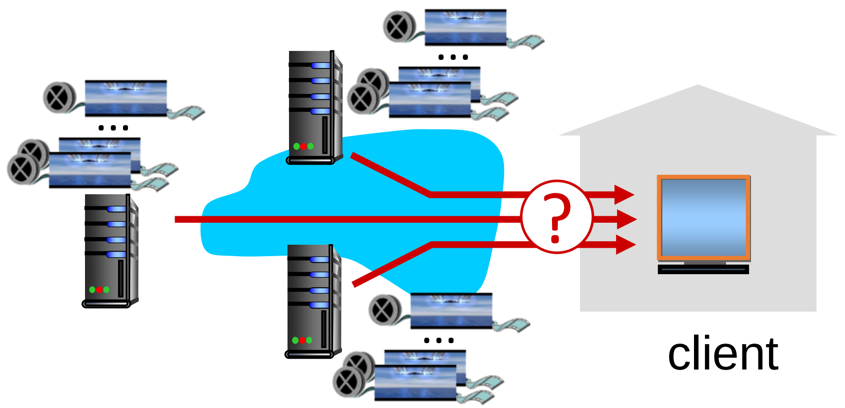
Streaming multimedia: DASH

*D*ynamic, *A*daptive
*S*treaming over *H*TTP

HTTP streaming:

* server:

- divides video file into multiple chunks
- each chunk encoded at multiple different rates (different quality levels)
- different rate encodings stored in different files
- files replicated in various CDN nodes
- *manifest file:* provides URLs for different chunks



* client:

- periodically estimates server-to-client bandwidth
- consulting manifest, requests one chunk at a time
  - chooses maximum coding rate available with the current bandwidth
  - can choose different coding rates at different points in time (depending on available bandwidth at time), and from different servers

Content distribution networks (CDNs)

challenge: how to stream content (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 (and possibly congested) path to distant clients
  - a single video will be sent many times over the same communication links, waste bandwith

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

Content distribution networks (CDNs)

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)
- 2 server placement approaches:
  - *enter deep:* CDN servers are in access networks (access ISP's)
    - close to users
    - Akamai: 240,000 servers deployed in > 120 countries (2015)
  - *bring home:* CDN servers are in IXP's
    - used by Limelight

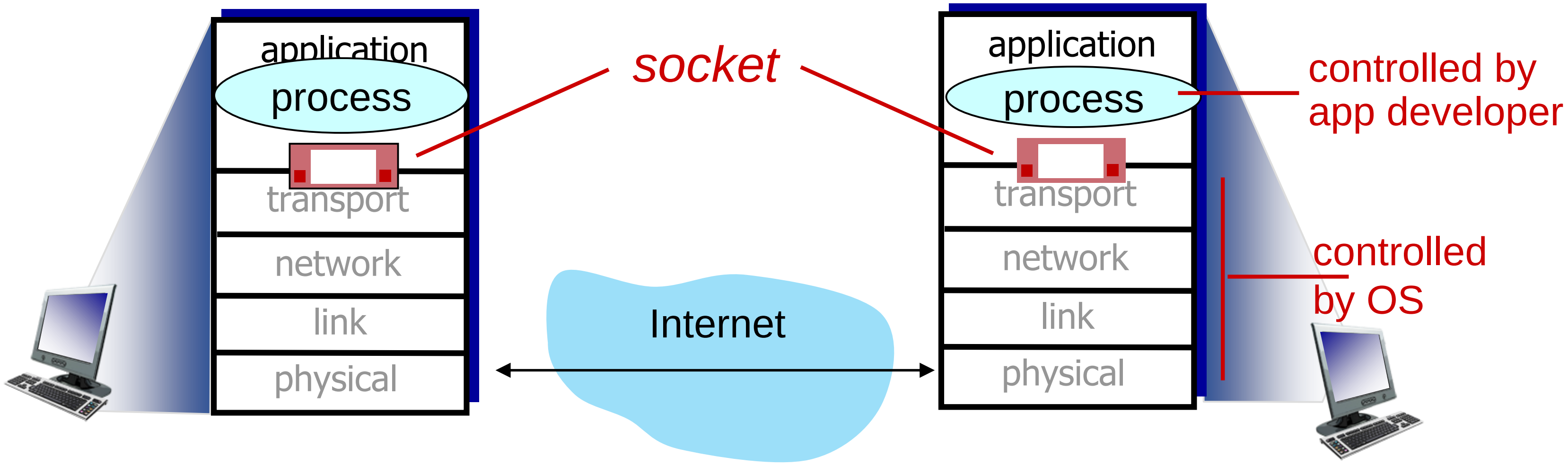
Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

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 data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

Socket programming with UDP

UDP: no "connection" between client and server:

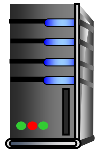
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

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 processes

Client/server socket interaction: UDP



server (running on serverIP)

client

creates the socket
* AF_INET: network is using IPv4
* DGRAM: datagram so UDP

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

read datagram from serverSocket

write reply to serverSocket specifying client address, port number

create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with serverIP address And port=x; send datagram via clientSocket

read datagram from clientSocket

close clientSocket

Example app: UDP client

Python UDPClient

include Python's socket library →

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
```

create UDP socket for server → clientSocket = socket(AF_INET, SOCK_DGRAM)

OS specifies the client port

get user keyboard input → message = raw_input('Input lowercase sentence:')

attach server name and port to message; send into socket → clientSocket.sendto(message.encode(), (serverName, serverPort))

read reply characters from socket into string → modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

print out received string and close socket → print modifiedMessage.decode(); clientSocket.close()

The slides are based on the slides by Computer Networking: A Top-Down Approach 8th edition, Jim Kurose, Keith Ross, Pearson, 2020

Example app: UDP server

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)

bind socket to local port number 12000 → serverSocket.bind(('', serverPort))

loop forever → while True:

Read from UDP socket into message, getting client's address (client IP and port) → message, clientAddress = serverSocket.recvfrom(2048)

send upper case string back to this client → serverSocket.sendto(modifiedMessage.encode(), clientAddress)

Socket programming with 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:

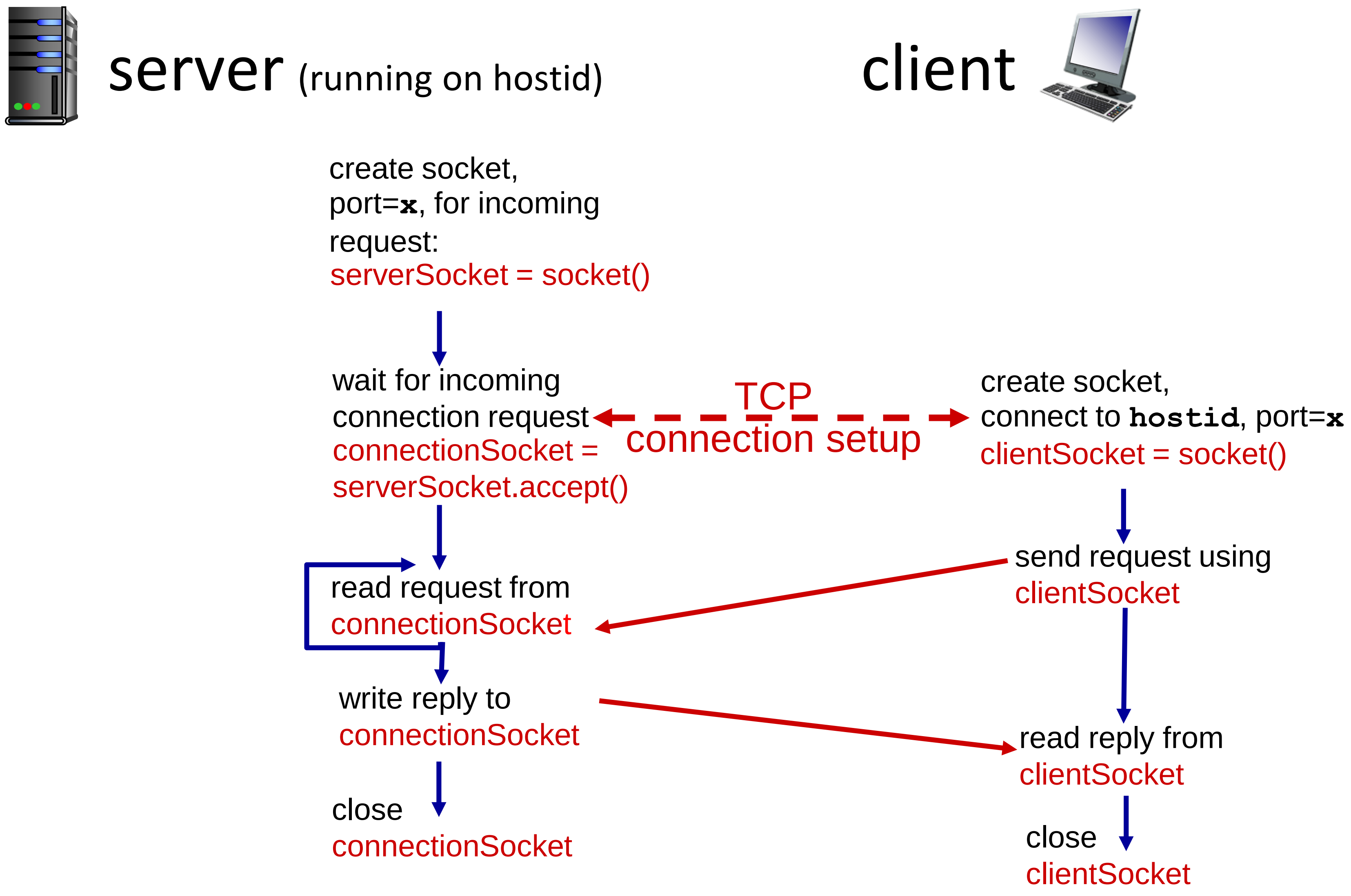
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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

Application viewpoint

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

Client/server socket interaction: TCP



Example app: TCP client

Python TCPClient

create TCP socket for server, remote port 12000

No need to attach server name, port

OS specifies the client port

```
from socket import *
serverName = ’servername’
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input(‘Input lowercase sentence:’)
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(2048)
print (‘From Server:’, modifiedSentence.decode())
clientSocket.close()
```

Example app: TCP server

Python TCPServer

create TCP welcoming socket

server begins listening for incoming TCP requests (max # of queued connections, at least 1)

loop forever

server waits on accept() for incoming requests, new socket created on return (dedicated to this particular client)

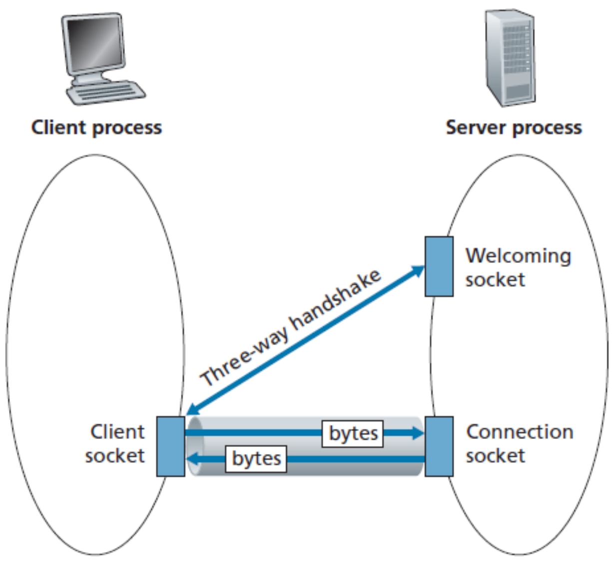
read bytes from socket (but not address as in UDP)

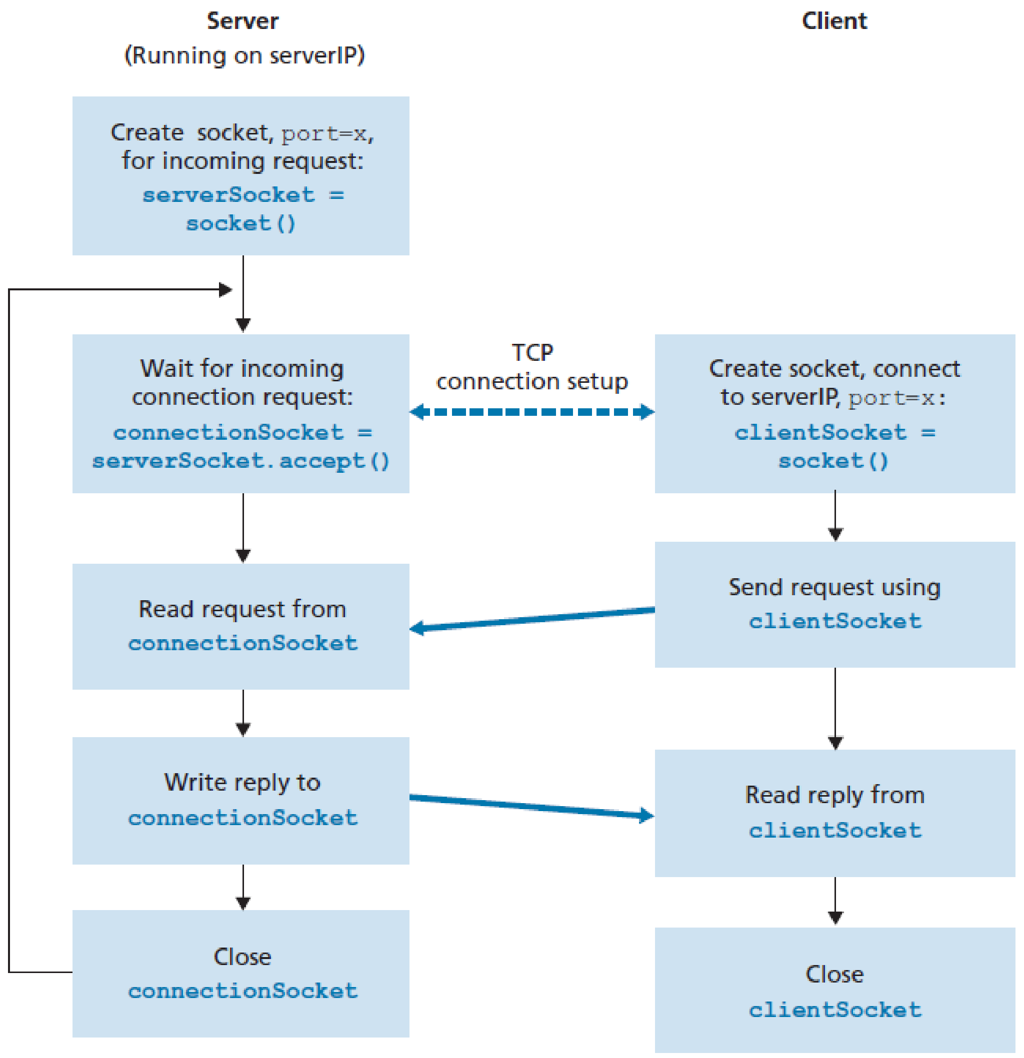
close connection to this client (but *not* welcoming socket)

```
 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 True:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(2048).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                         encode())
    connectionSocket.close()
```

General flow





TCPServer process has two sockets:

- serverSocket for welcoming the client and 3way hs
- connectionSocket for the TCP connection