CSCI 6760 - Computer Networks - Fall 2024

Instructor: Prof. Roberto Perdisci

perdisci@uga.edu

These slides are adapted from the textbook slides by J.F. Kurose and K.W. Ross

Application layer: overview

Our goals:

- conceptual *and* implementation aspects of application-layer protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm

- learn about protocols by examining popular application-layer protocols
  - HTTP
  - SMTP, IMAP
  - DNS
- programming network applications
  - socket API

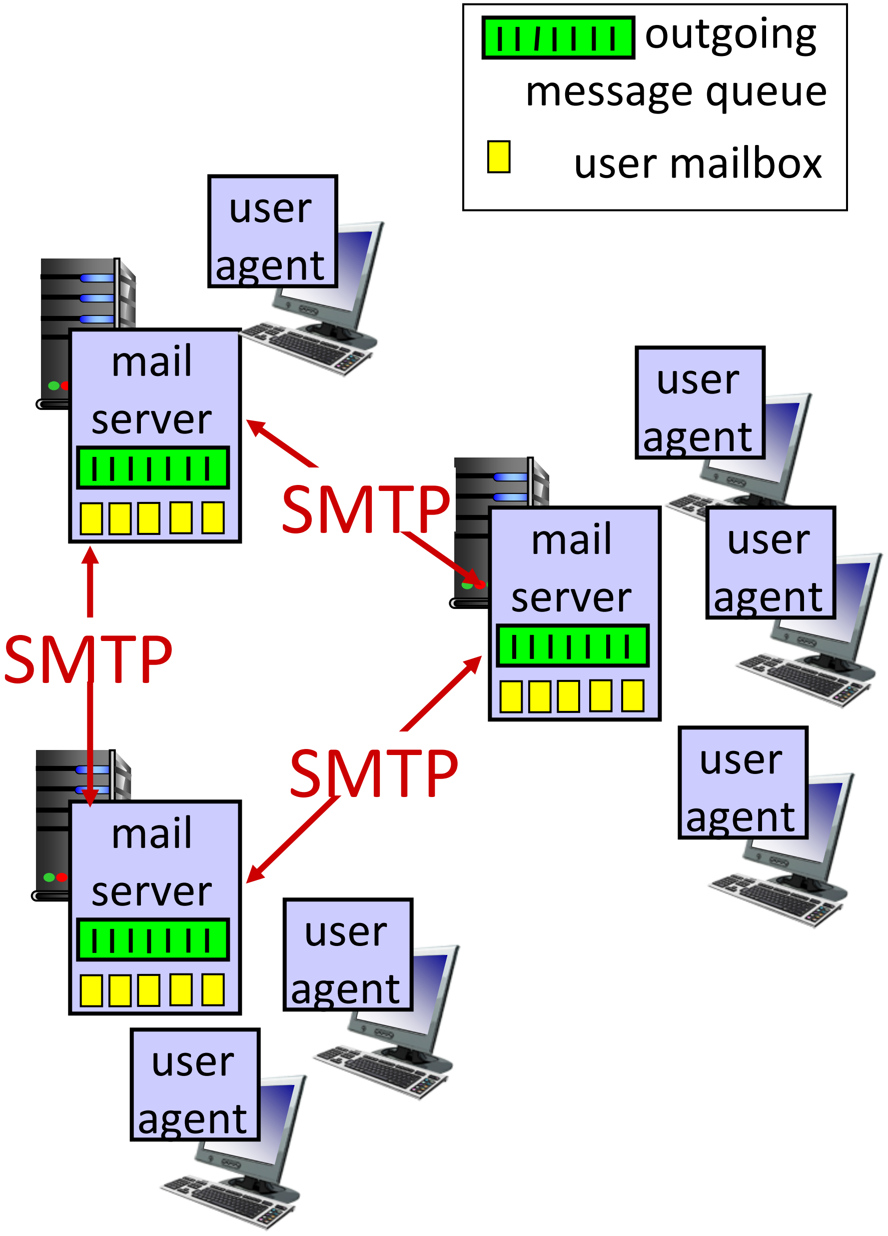
E-mail

Three major components:

- user agents
- mail servers
- simple mail transfer protocol: SMTP

User Agent

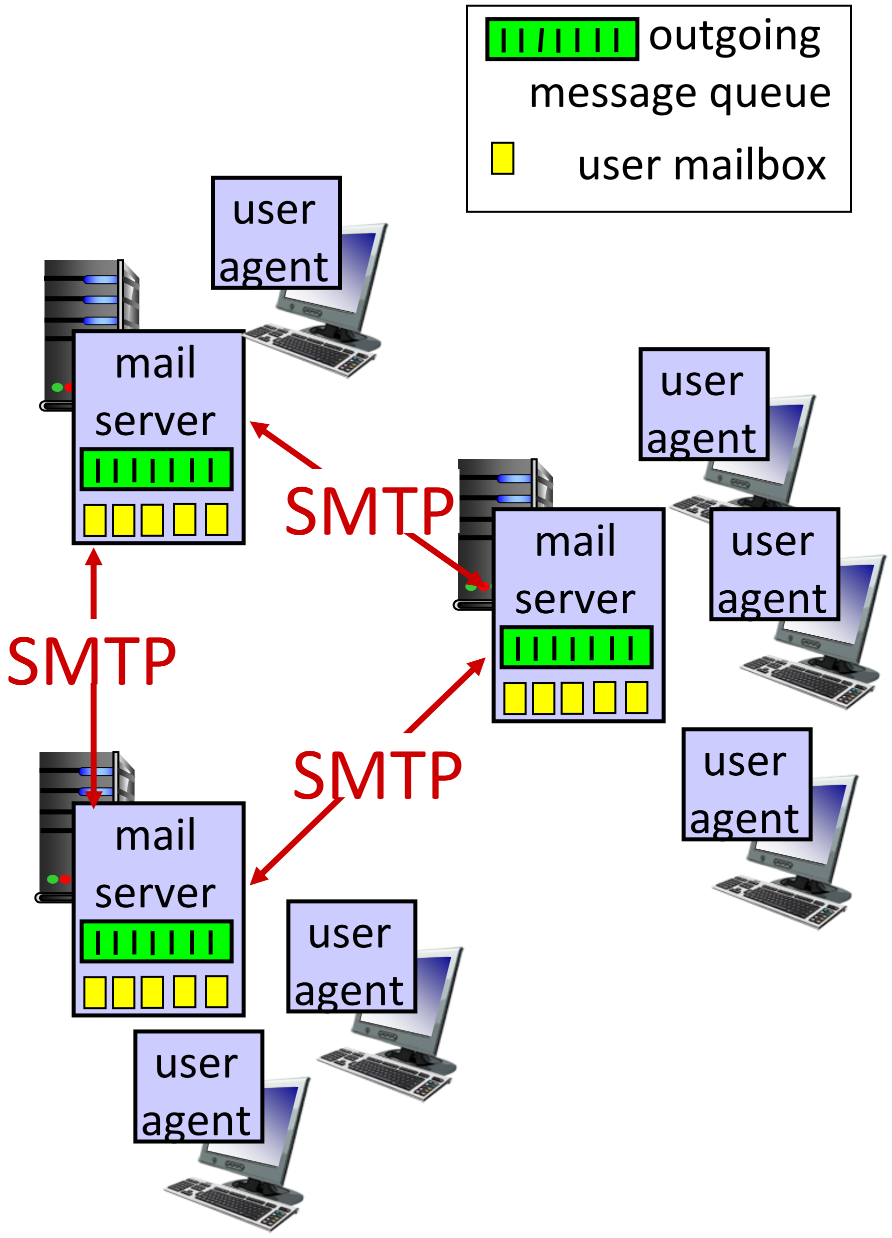
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Outlook, iPhone mail client
- outgoing, incoming messages stored on server



E-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



E-mail: the RFC (5321)

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

Scenario: Alice sends e-mail to Bob

1) Alice uses UA to compose e-mail message "to" bob@someschool.edu

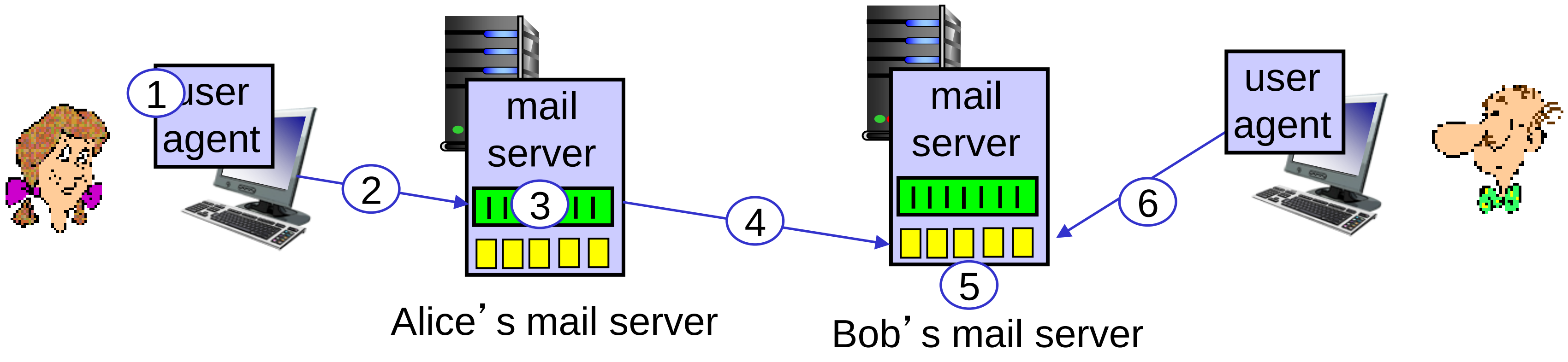
2) Alice's UA sends message to her mail server; message placed in message queue

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



Scenario: Alice sends e-mail to Bob

1) Alice uses UA to compose e-mail message “to” bob@someschool.edu

2) Alice’s UA sends message to her mail server; message placed in message queue

3) client side
connection

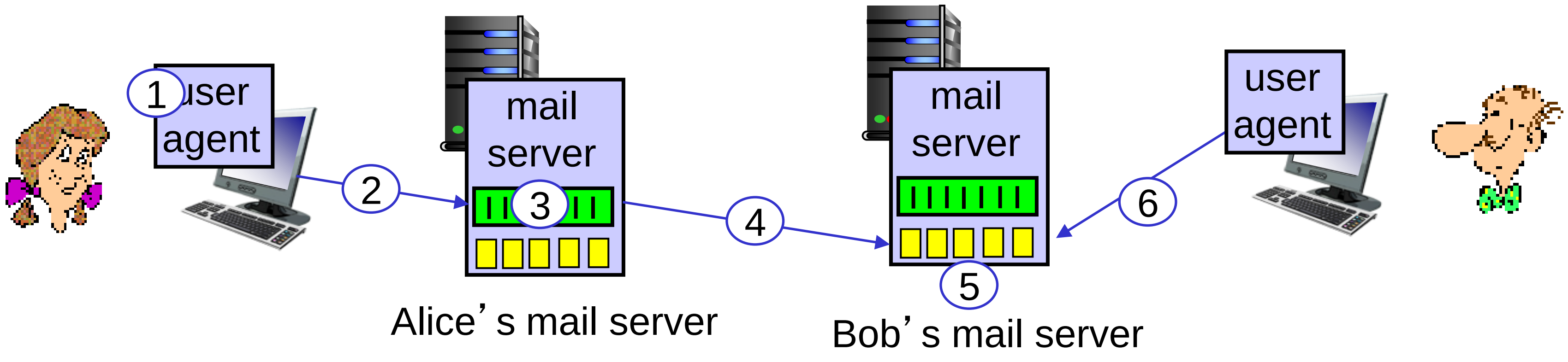
4) SMTP client sends Alice’s message over the TCP connection

5) Bob’s mail server places the message in Bob’s mailbox

nvokes his user
to read message

Alice’s mail server will need to issue a DNS query of type MX to find the name of Bob’s mail server

(qname=someschool.edu, qtype=MX)



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
```

Try SMTP interaction for yourself:

telnet <servername> 25

- see 220 reply from server
- enter HELO, MAIL FROM:, RCPT TO:, DATA, QUIT commands

above lets you send email without using e-mail client (reader)

Note: this will only work if <servername> allows telnet connections to port 25 (this is becoming increasingly rare because of security concerns)

SMTP: closing observations

comparison with HTTP:

- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response message
- SMTP: multiple objects sent in multipart message

- 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

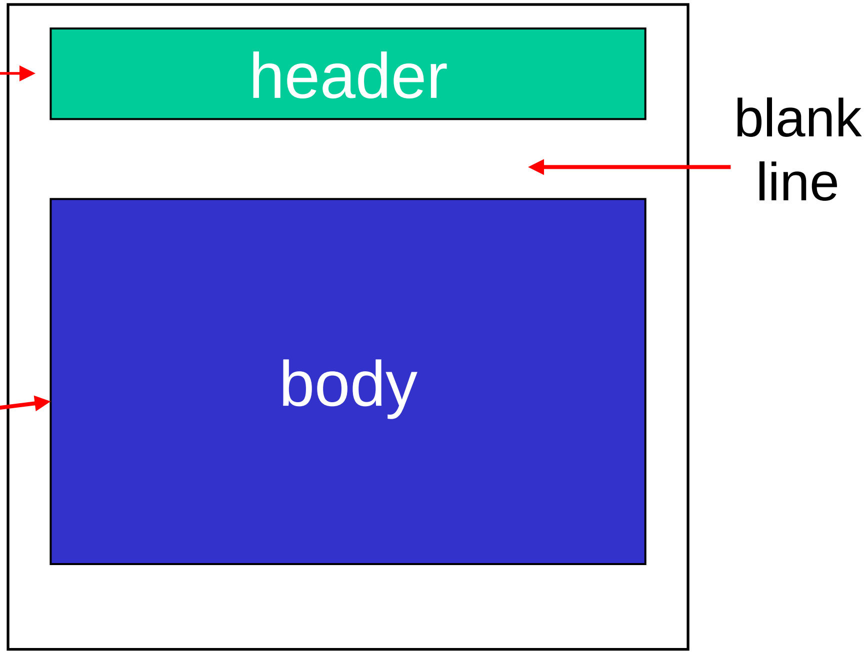
Mail message format

SMTP: protocol for exchanging e-mail messages, defined in RFC 531 (like HTTP)

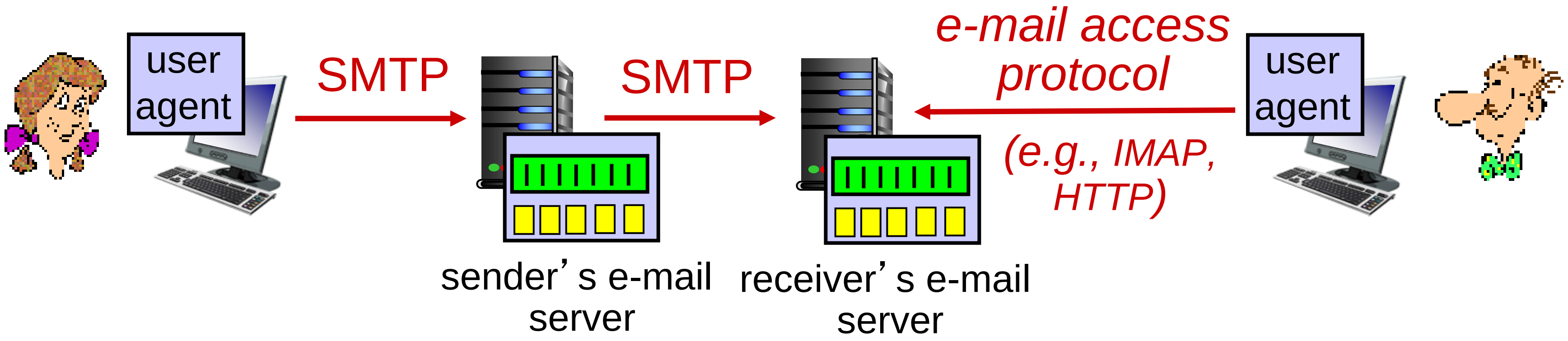
RFC 822 defines *syntax* for e-mail message itself (like HTML)

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

  these lines, within the body of the email message area different from SMTP MAIL FROM:, RCPT TO: commands!
- Body: the "message" , ASCII characters only



Mail access protocols



- SMTP: delivery/storage of e-mail messages to receiver’s server
- mail access protocol: retrieval from server
  - IMAP: Internet Mail Access Protocol [RFC 3501]: messages stored on server, IMAP provides retrieval, deletion, folders of stored messages on server
- HTTP: gmail, Hotmail, Yahoo!Mail, etc. provides web-based interface on top of STMP (to send), IMAP (or POP) to retrieve e-mail messages

DNS: Domain Name System

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 ?

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: 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

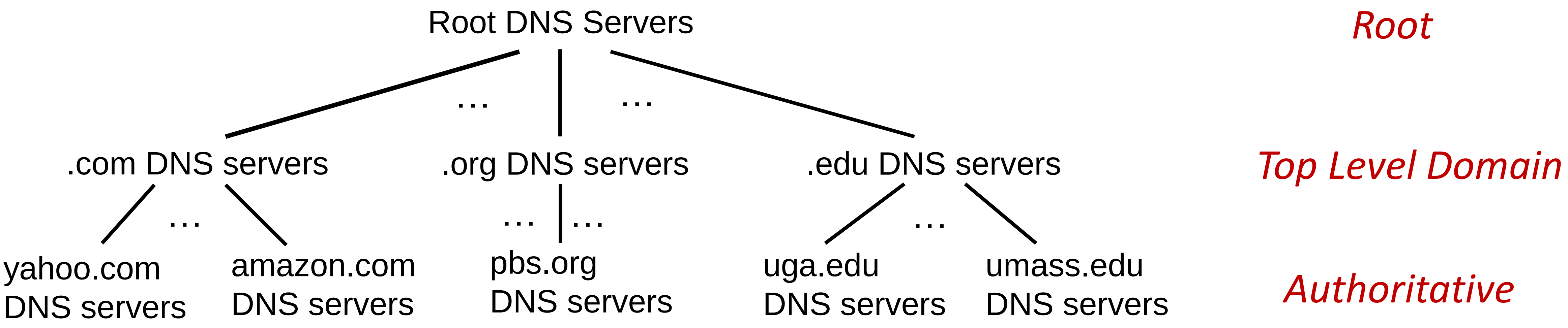
Q: Why not centralize DNS?

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

A: doesn't scale!

- Comcast DNS servers alone: 600B DNS queries per day

DNS: a distributed, hierarchical database



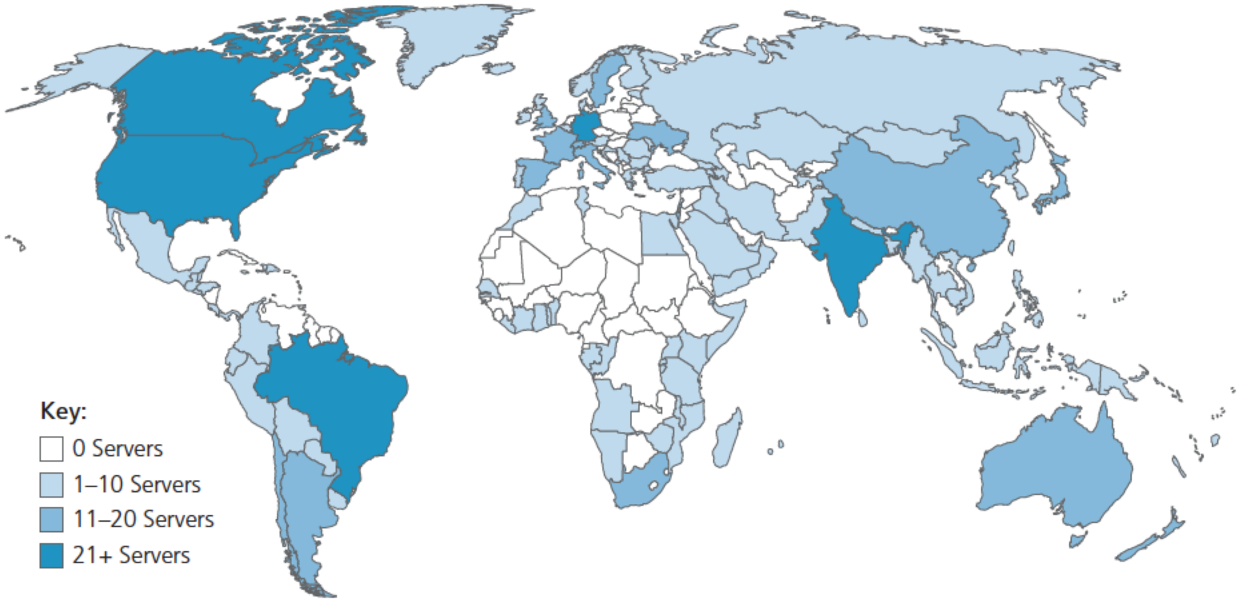
Client wants IP address for www.amazon.com; 1st approximation:

- 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

- official, contact-of-last-resort by name servers that can not resolve name
- *incredibly important* Internet function
  - Internet couldn't function without it!
  - DNSSEC – provides security (authentication and message integrity)
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

13 logical root name "servers" worldwide each "server" replicated many times (~200 servers in US)



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.: .cn, .uk, .fr, .ca, .jp
- Network Solutions: authoritative registry for .com, .net TLD
- Educause: .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

Local DNS name servers (resolvers)

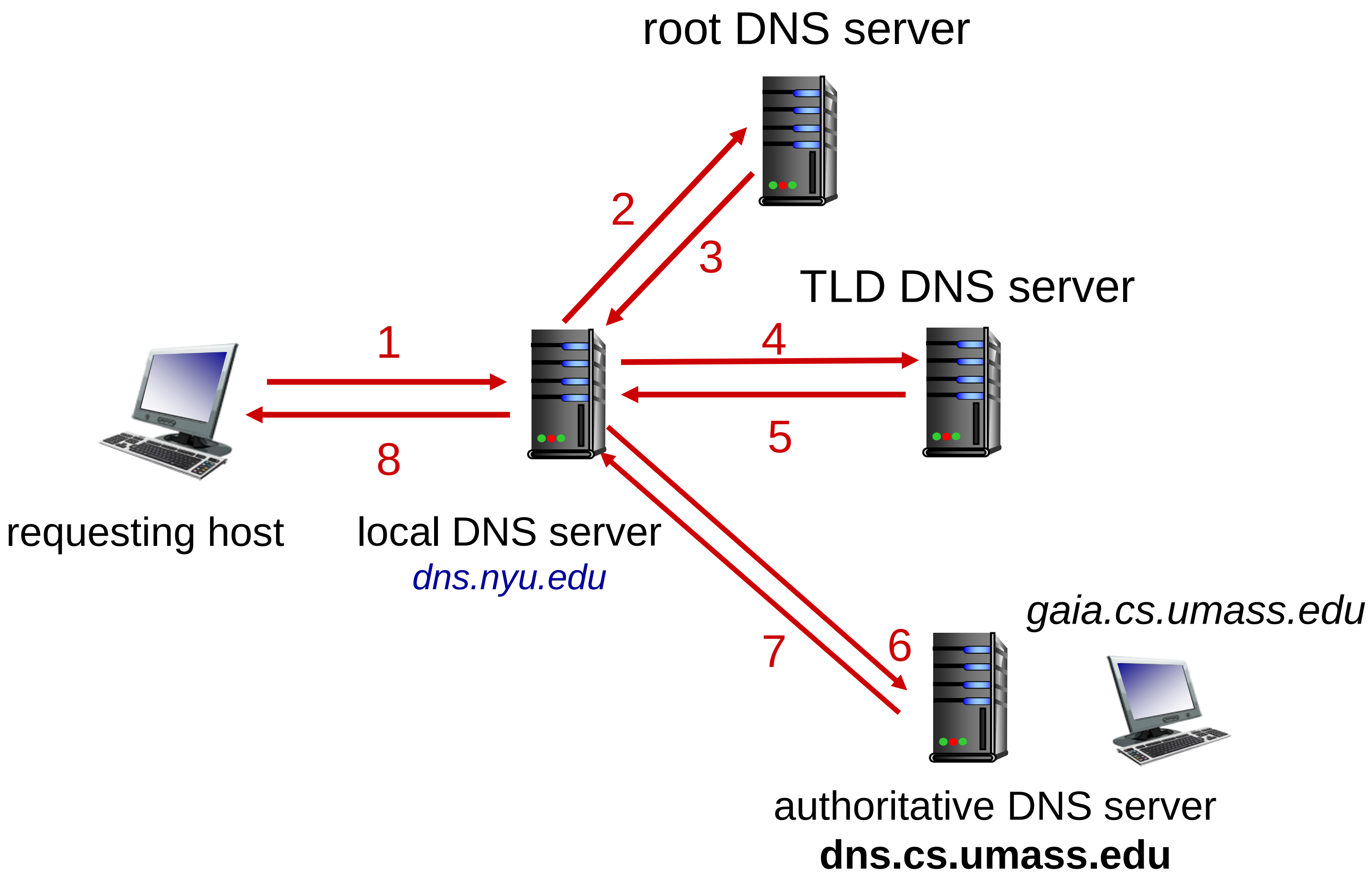
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called "default name server"
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts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

DNS name resolution: iterated query

Example: want 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”

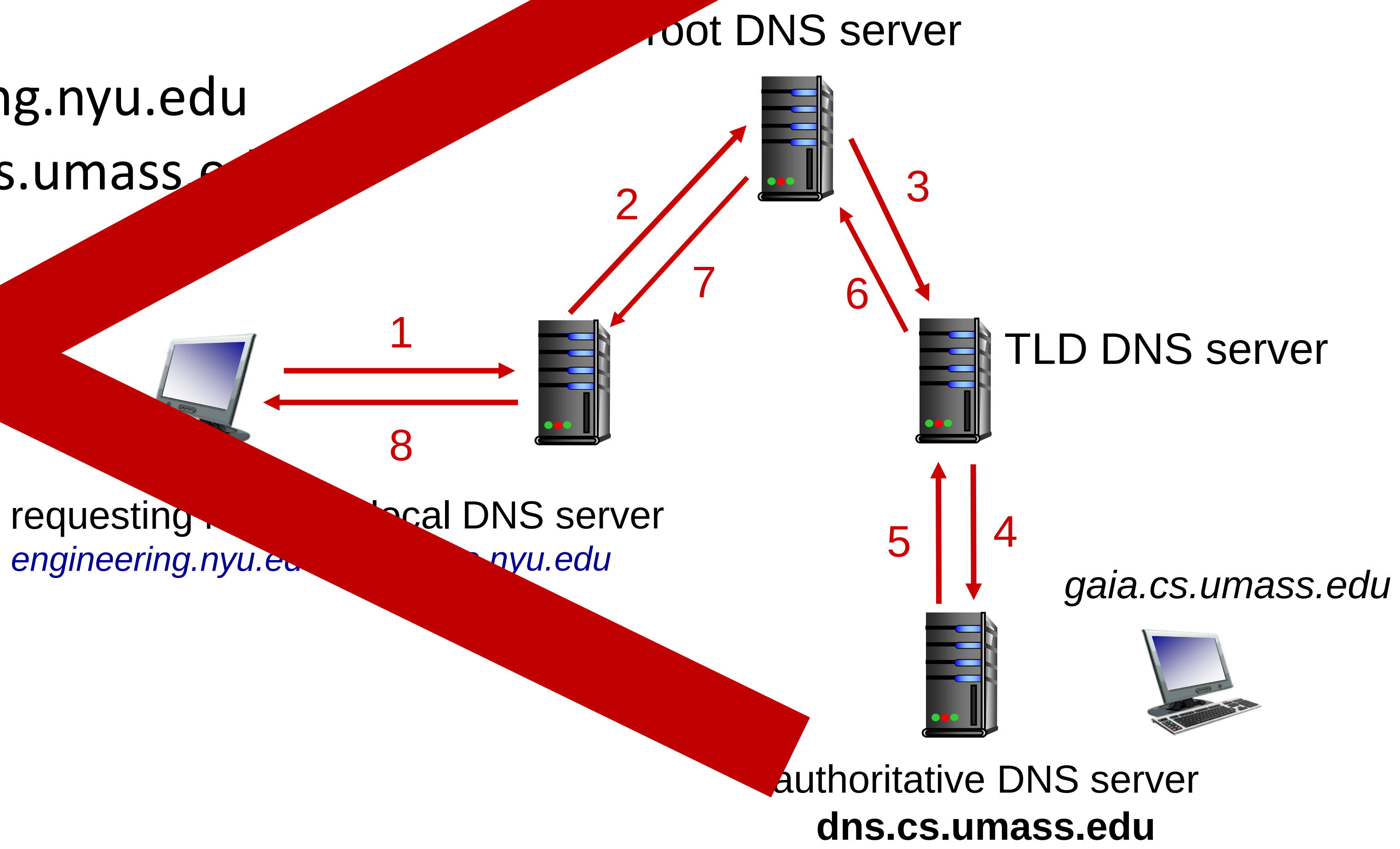


DNS name resolution: recursive qu…

Ex… : host at engineering.nyu.edu wants I… …ss for gaia.cs.umass.e…

Recursive query:

- puts burden of … …e resolutio… co… …d name …ver
- heavy load at upper levels of hierarchy?



Caching, Updating DNS Records

- once DNS resolver 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 records

DNS: distributed database 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`

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 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

Exercise: Setup and run your own NS

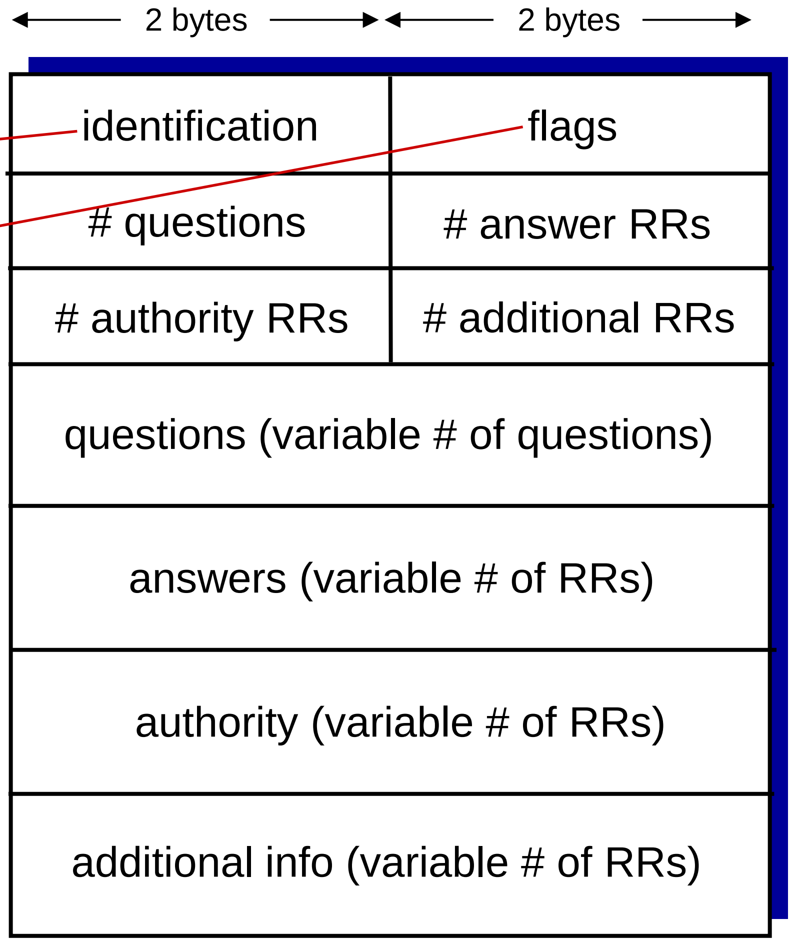
- Install BIND on your VM
- Configure your own domain name zone
  - Example: https://help.dyn.com/how-to-format-a-zone-file/
  - Tutorial: https://www.linuxbabe.com/ubuntu/set-up-authoritative-dns-server-ubuntu-18-04-bind9
- Use dig to make queries against your own NS
  - Query for existing and non existing records
  - Map multiple IPs to a single domain
  - Try setting up non-A records, such as AAAA, MX, CNAME, etc.

DNS protocol messages

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

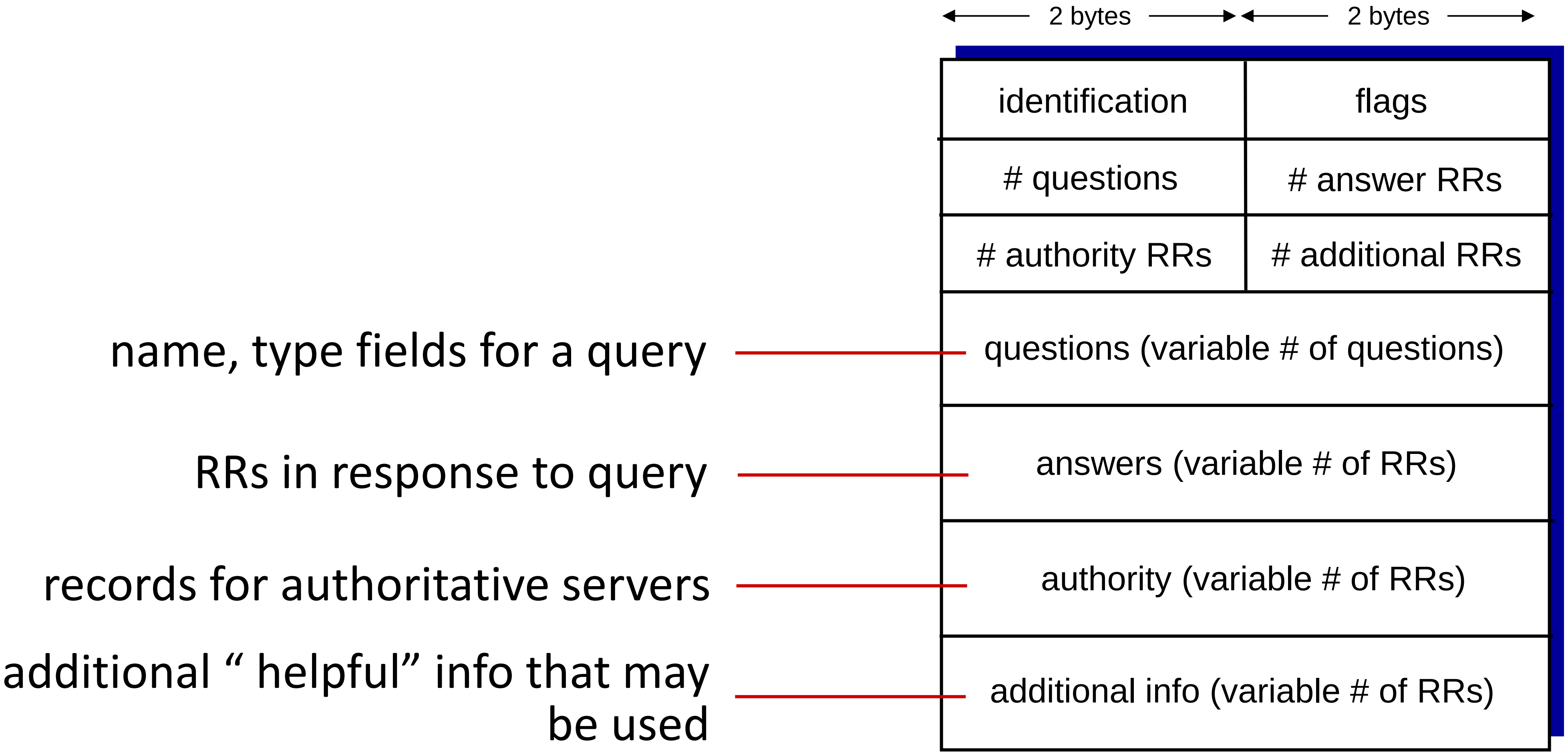
message header:

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



DNS protocol messages

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



DNS requests using Scapy

```
>>> dns_q = DNS(id=12345, qd=DNSQR(qtype='A', qname='nasa.gov.’))
>>> ip = IP(dst=‘1.1.1.1')/UDP(sport=54321, dport=53)/dns_q
>>> dns_resp = sr1(ip)
```

DNS attacks

- Attacks on integrity
  - MITM – change RR values
  - Cache Poisoning
  - Defense: DNSSEC
- Attacks on confidentiality
  - DNS traffic monitoring
  - Defense: DoH, DoT
- Attacks on availability
  - DDoS towards DNS servers
  - Abusing DNS servers for DRDoS attacks

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

Redirect attacks

- man-in-middle
  - intercept DNS queries
  - DNS poisoning
    - send forged replies to DNS server, which caches

DNSSEC [RFC 4033]

Exploit DNS for DRDoS amplification

- send queries with spoofed source address: target IP
- requires amplification

“Blind” DNS cache poisoning

- UDP is connectionless
- First valid response is accepted by the local DNS server
- Only mechanism for “authenticating” the responses is to check that
  - Query section matches the request
  - TXID is same as request
  - SrcIP matches DstPort of request
  - DstPort matches SrcPort of request
- Consequence of successful attack
  - All subsequent queries for poisoned domain will be resolved to attacker’s IP address
  - E.g., all visitors of www.google.com could be redirected to an attacker’s server!

Interesting uses of DNS

- Content Delivery Networks
  - E.g., dig www.microsoft.com

- DoH – DNS over HTTPS
  - Example:
    - https://developers.cloudflare.com/1.1.1.1/encrypted-dns/dns-over-https/make-api-requests
    - https://developers.google.com/speed/public-dns/docs/secure-transports#doh
    - https://developers.google.com/speed/public-dns/docs/doh/json

    - https://dns.quad9.net:5053/dns-query?name=quad9.net&type=NS

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 & server

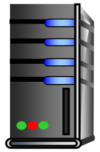
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

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

Client/server socket interaction: 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

Example app: UDP client

Python UDPClient

include Python's socket library → from socket import *

serverName = 'hostname'

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.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()

```
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()
```

Example app: UDP server

Python UDPServer

create UDP socket
bind socket to local port number 12000
loop forever
Read from UDP socket into message, getting client's address (client IP and port)
send upper case string back to this client

```
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)
```

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:

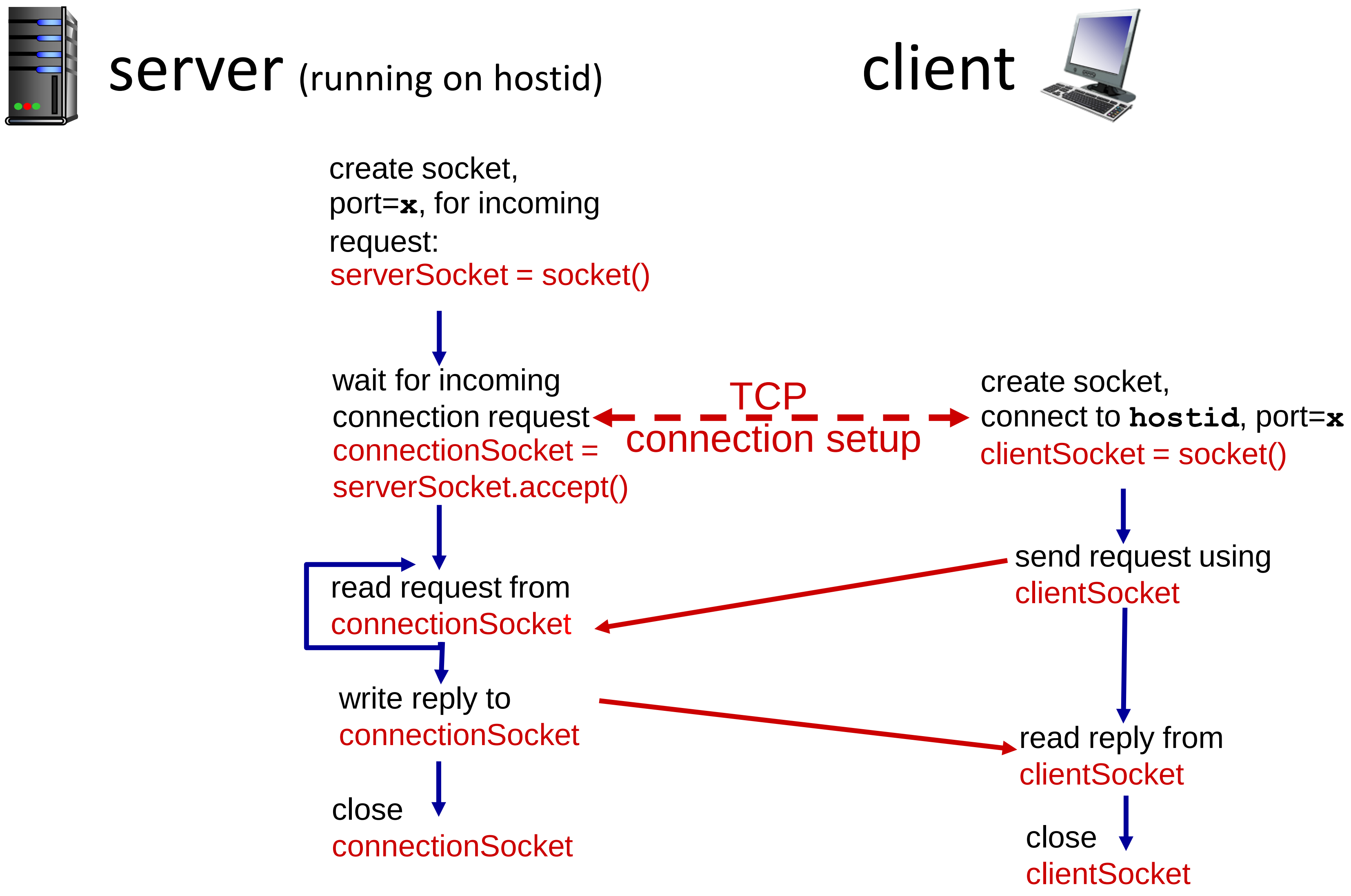
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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 (more in Chap 3)

Application viewpoint

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

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

```
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(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

Example app: TCP server

Python TCPServer

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 True:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                            encode())
    connectionSocket.close()
```

TCP/IP socket programming in C

- See example from book:
  - "TCP/IP Sockets in C: Practical Guide for Programmers, Second Edition" by Michael J. Donahoo and Kenneth L. Calvert