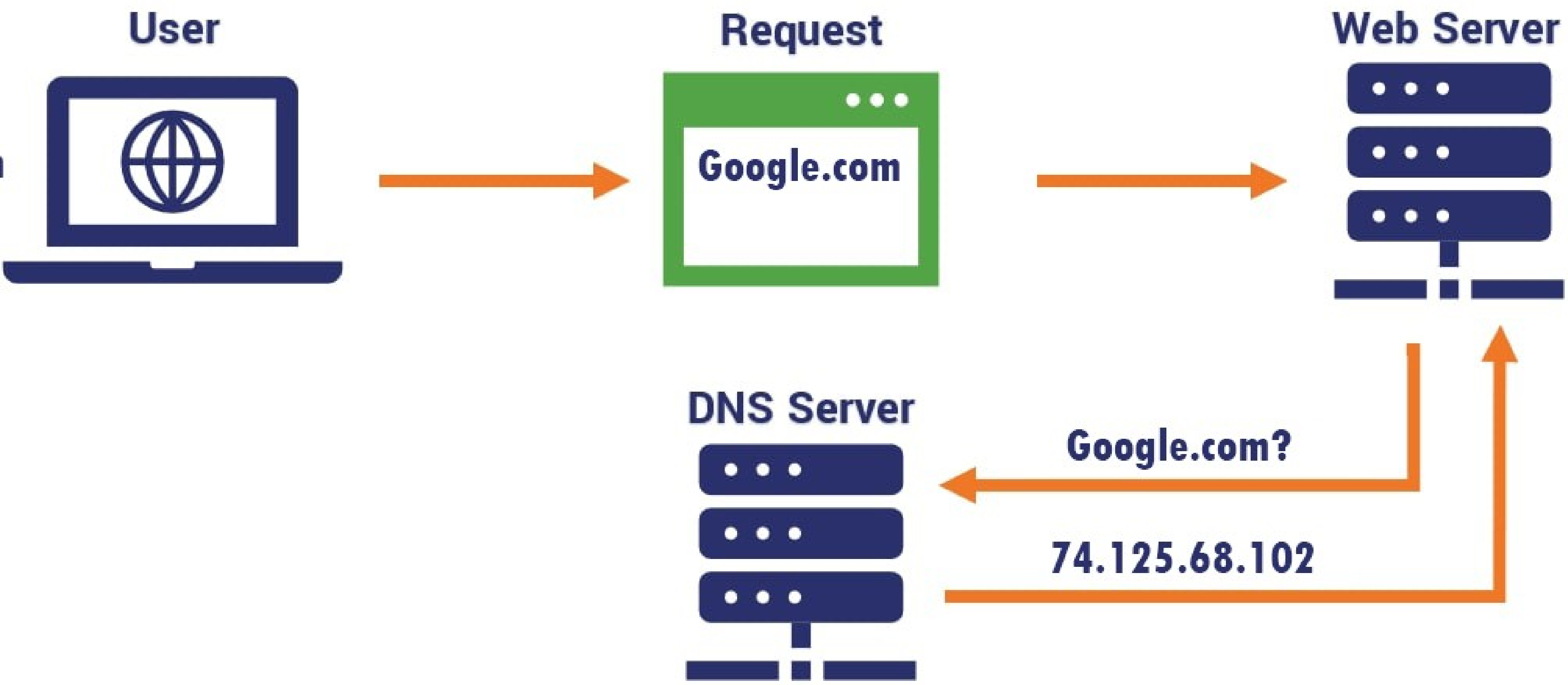
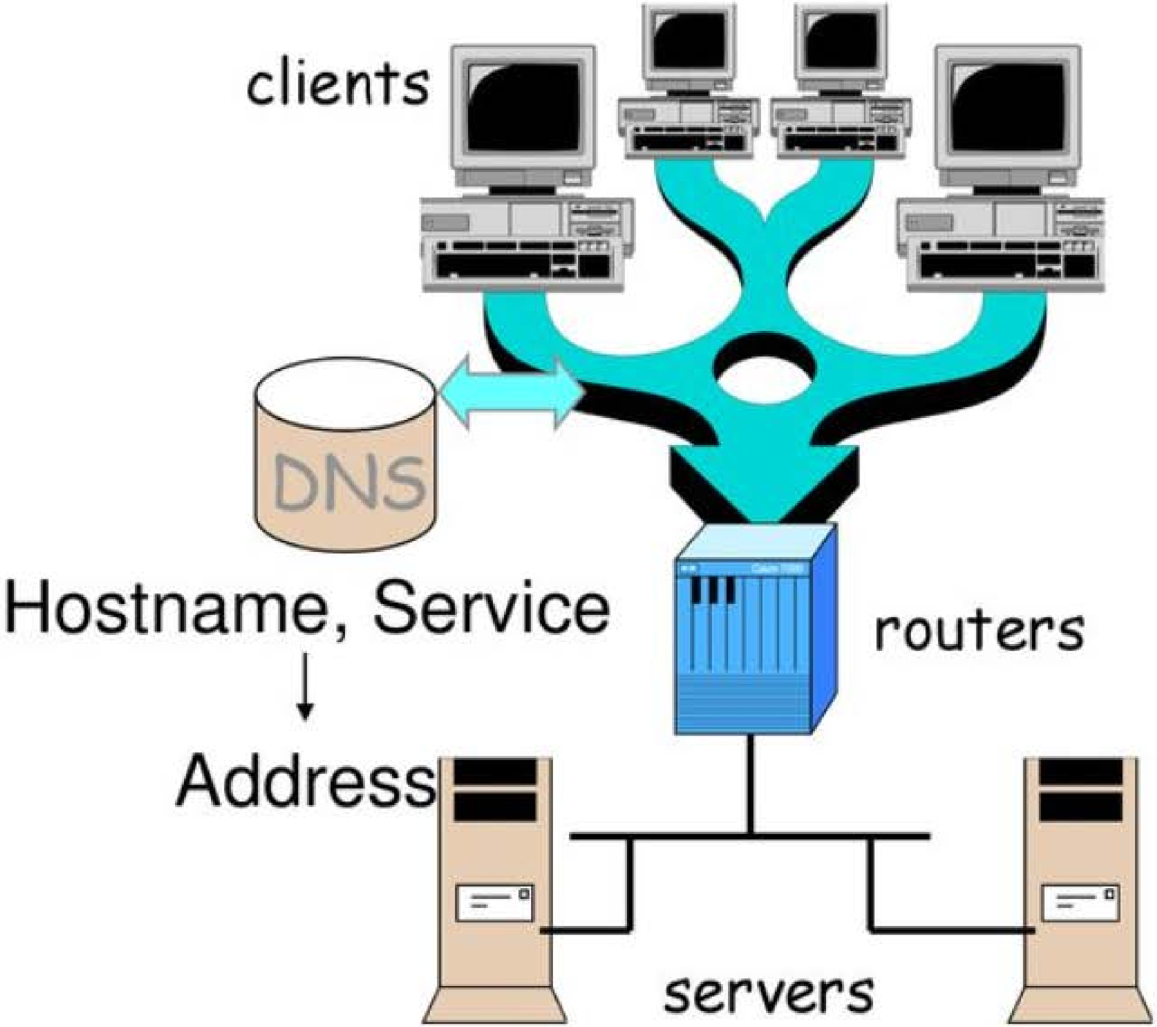
Application Layer: Overview

- 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


User
Request
Google.com
Web Server
DNS Server
Google.com?
74.125.68.102

Function

- map between (domain name, service) to value, e.g.,
  - (www.cs.yale.edu, Addr) -> 128.36.229.30
  - (cs.yale.edu, Email) -> netra.cs.yale.edu



```
Reply from 103.235.46.39: bytes=32 time=3ms TTL=56
Reply from 103.235.46.39: bytes=32 time=3ms TTL=56
Reply from 103.235.46.39: bytes=32 time=3ms TTL=56
Reply from 103.235.46.39: bytes=32 time=3ms TTL=56

Ping statistics for 103.235.46.39:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 3ms, Maximum = 3ms, Average = 3ms

C:\Users\rfan>nslookup
Default Server:  intudns-vip.ied.edu.hk
Address:  192.168.166.103

> www.baidu.com
Server:  intudns-vip.ied.edu.hk
Address:  192.168.166.103

Non-authoritative answer:
Name:    www.wshifen.com
Address:  103.235.46.39
Aliases:  www.baidu.com
          www.a.shifen.com
```

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 ?

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

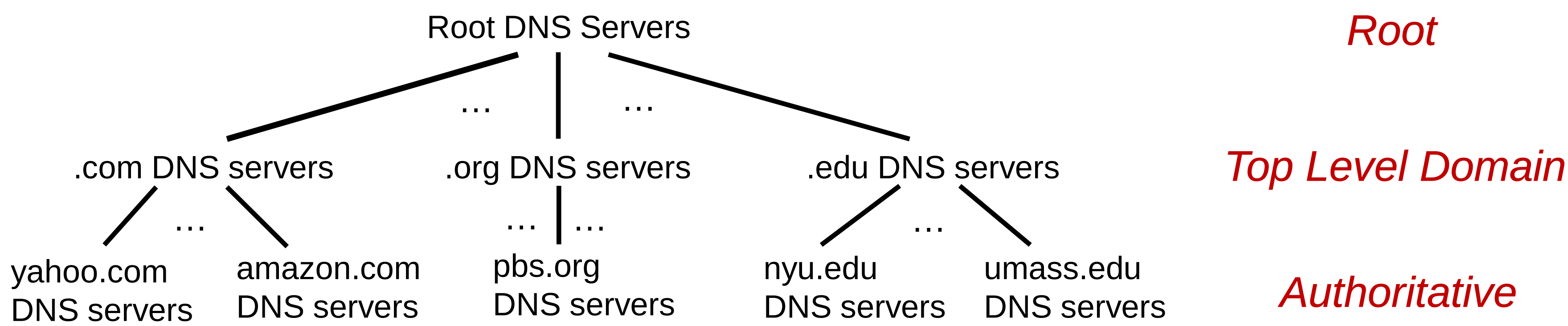
Q: Why not centralize DNS?

- 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



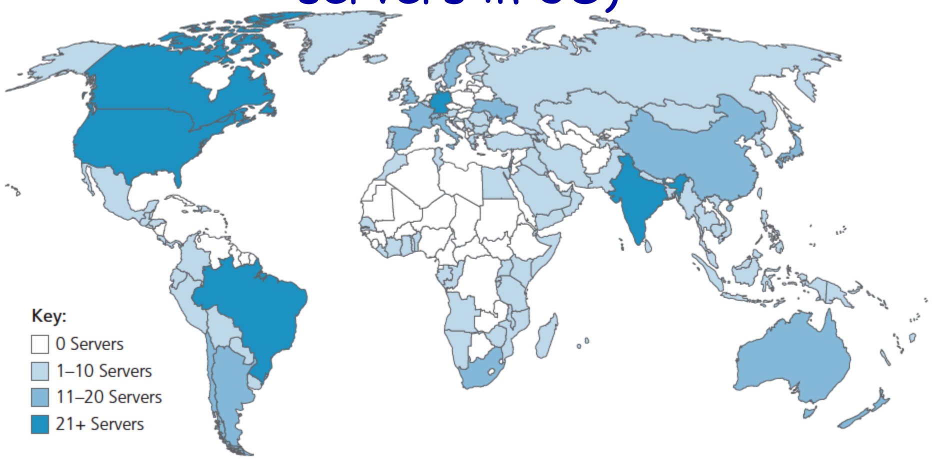
Client wants IP address for www.amazon.com; 1st approximation:

- 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

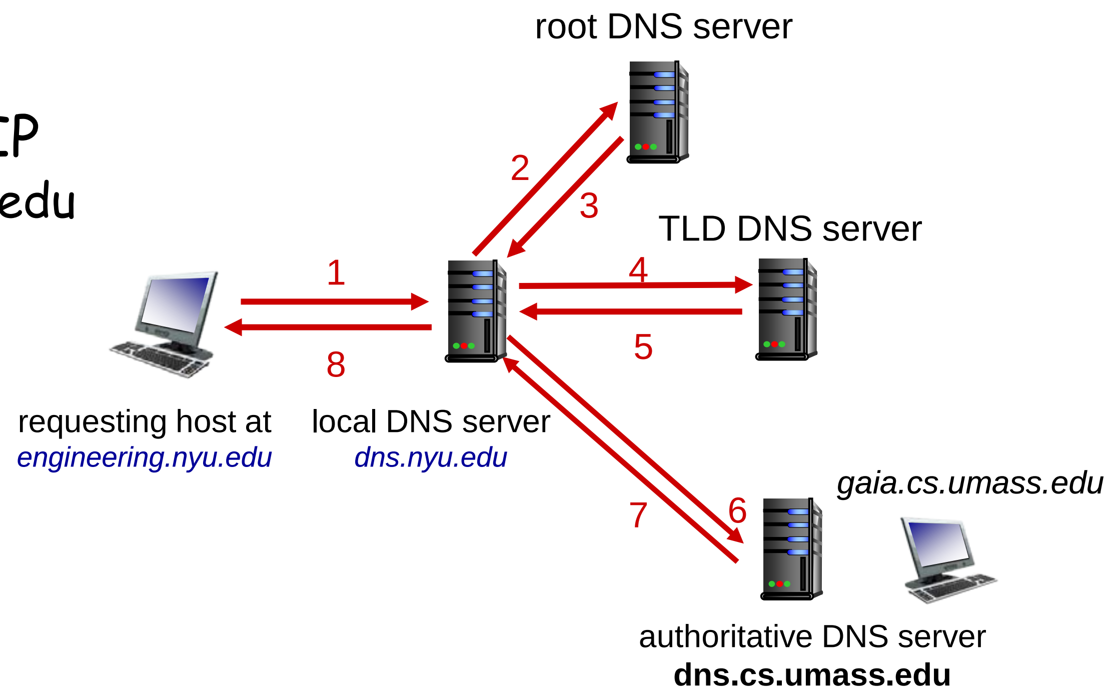
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called "default name server"
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts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

DNS name resolution: iterated query

Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu

Iterated query:

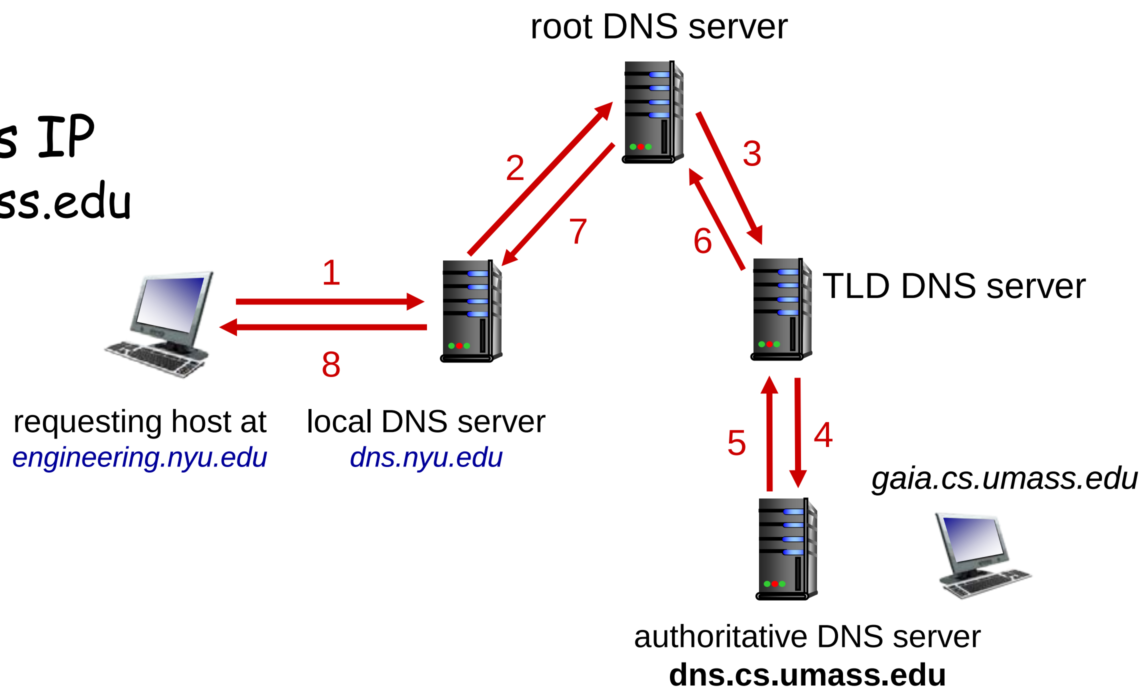
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS name resolution: recursive query

Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu

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



Caching, Updating DNS 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 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 bogus relies to DNS server, which caches

Exploit DNS for DDoS

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

DNSSEC [RFC 4033]

SOCKET PROGRAMMING

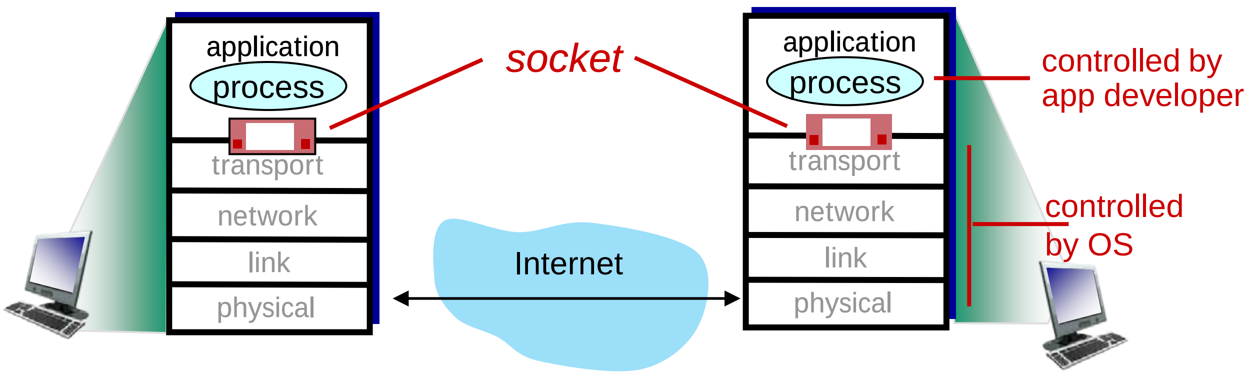
Application Layer: Overview

- 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



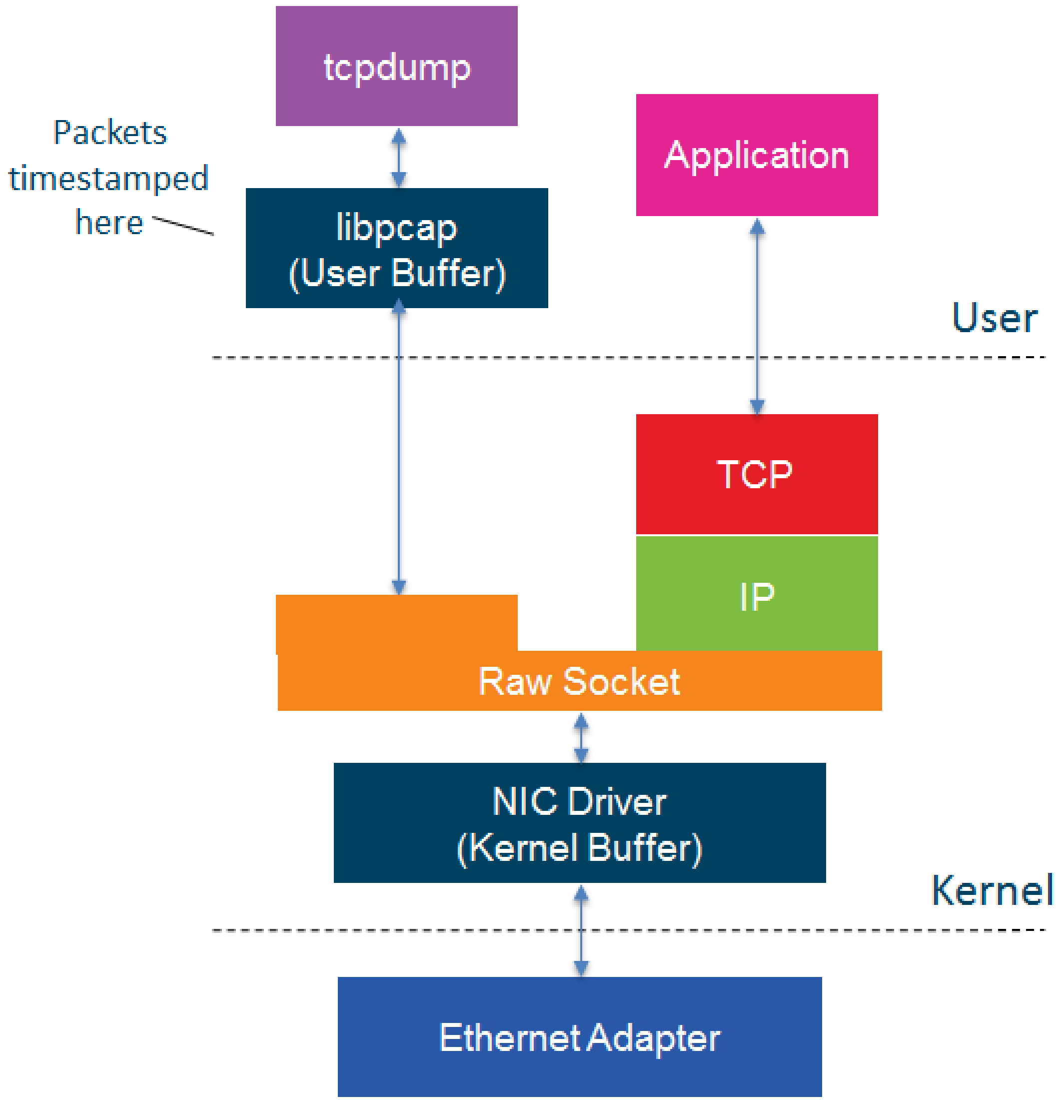
Socket programming

Two socket types for two transport services:

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

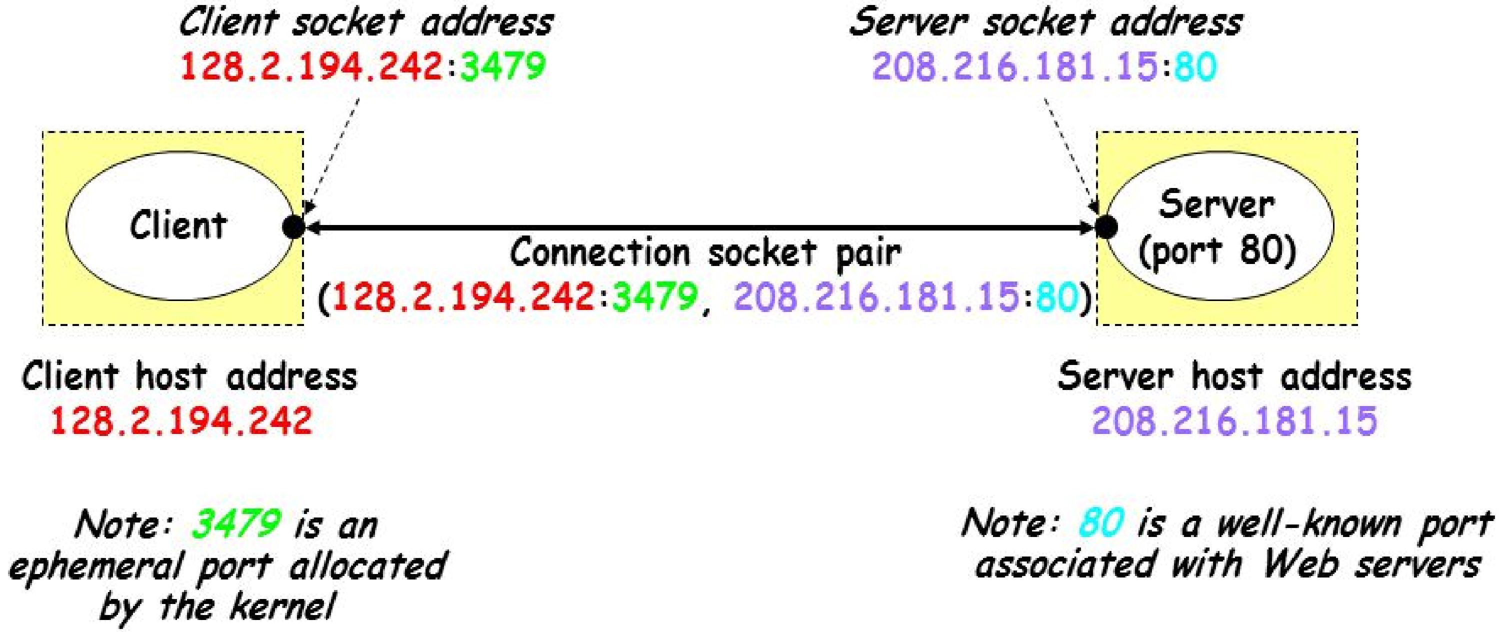
Application Example:

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


tcpdump
Packets timestamped here
libpcap
(User Buffer)
Application
User
TCP
IP
Raw Socket
NIC Driver
(Kernel Buffer)
Kernel
Ethernet Adapter

Internet Connections (TCP/IP)

- Address the machine on the network
  - By IP address
- Address the process
  - By the "port"-number
- The pair of *IP-address + port* – makes up a "*socket-address*"



What APIs Needed?

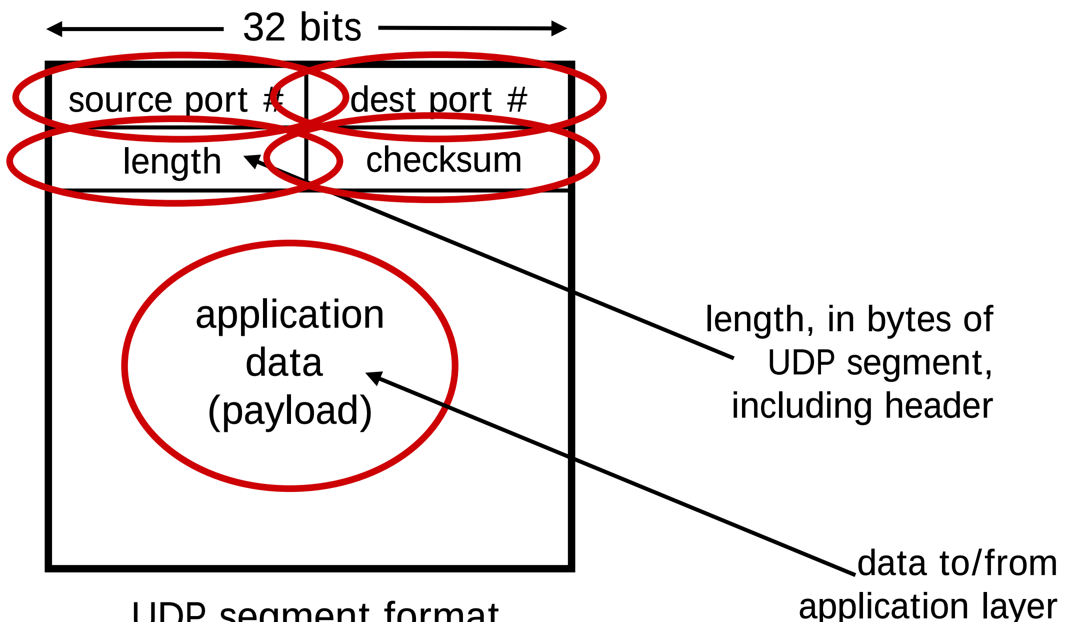
Connection-Oriented TCP

- How to create socket (door)
- How to establish connection
  - Client connects to a server
  - Server accepts client req.
- How to send/recv data
- How to identify socket
  - Bind to local address/port
- How to close socket (door)

Connectionless UDP

- How to create socket
- How to send/recv data
- How to identify socket
- How to close socket

UDP segment header



UDP segment format

Socket programming with UDP

UDP: no "connection" between client & server

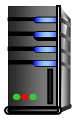
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

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()
```

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)
```

TCP/IP Socket

- Creating
  - int socket(int protocolFamily, int type, int protocol)
    - protocol Family: PF_INET
    - type: SOCK_STREAM, SOCK_DGRAM
    - protocol: IPPROTO_TCP, IPPROTO_UDP
    - return value: File descriptor (socket descriptor)
- Destroying
  - int close(int socket)
    - socket: socket descpriptor

TCP/IP Socket

- TCP **Client**
  - Create a TCP socket
  - Establish connection
    - int connect(int socket, struct sockaddr *foreignAddress, unsigned int addressLength)
  - Communicate
    - int send(int socket, const void *msg, unsigned int msgLength, int flags)
    - int recv(int socket, void *rcvBuffer, unsigned int bufferLength, int flags)
  - Close the connection

TCP/IP Socket

- TCP **Server**
  - Create a TCP socket
  - Assign a port to socket
    - int bind(int socket, struct sockaddr *localAddress, unsigned int addressLength)
  - Set socket to listen
    - int listen(int socket, int queueLimit)
  - Repeatedly
    - Accept new connection
      - int accept(int socket, struct sockaddr *clientAddress, unsigned int *addressLength)
    - Communicate
    - Close the connection

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:

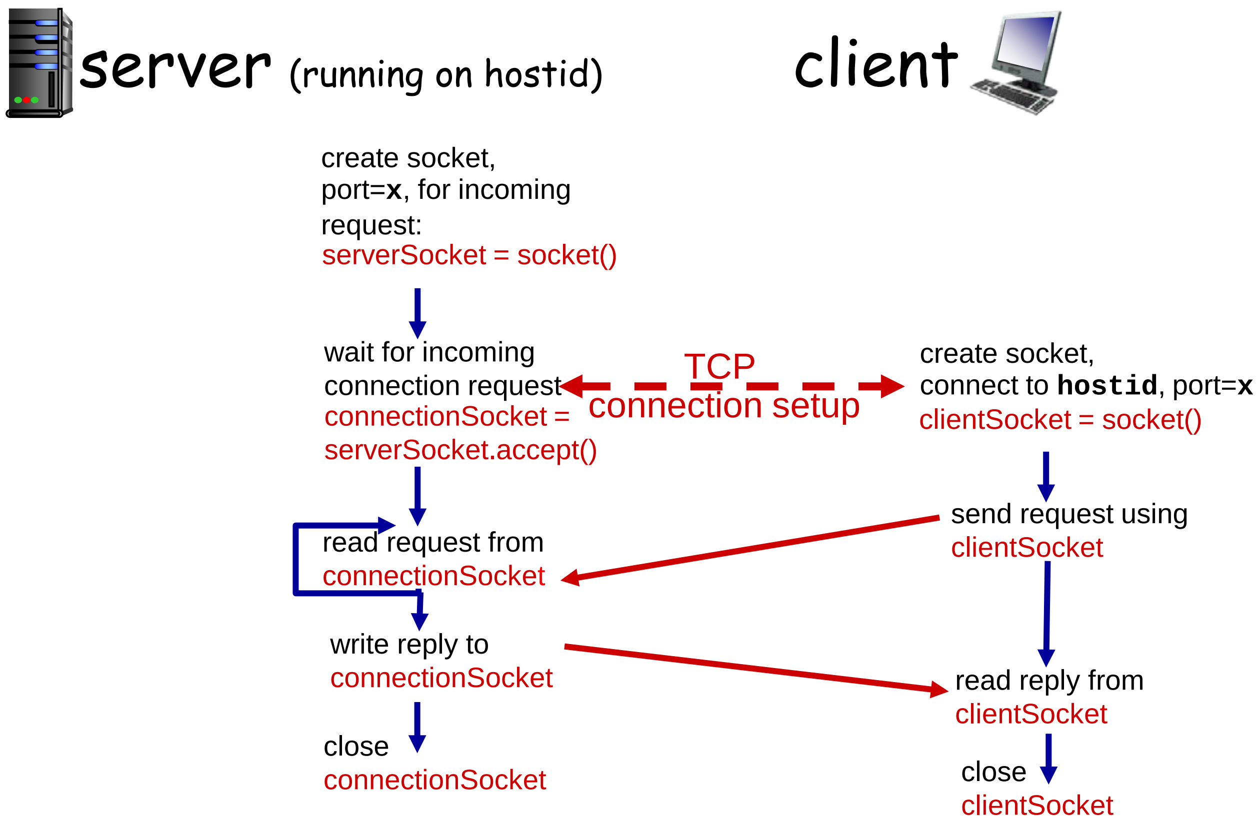
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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()
```

Chapter 2: Summary

our study of network application layer is now complete!

- 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
  - SMTP, IMAP
  - DNS
  - P2P: BitTorrent
- video streaming, CDNs
- socket programming: TCP, UDP sockets

Chapter 2: Summary

Most importantly: learned about *protocols*!

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - *headers*: fields giving info about data
  - *data:* info(payload) being communicated

important themes:

- centralized vs. decentralized
- stateless vs. stateful
- scalability
- reliable vs. unreliable message transfer
- "complexity at network edge"

Announcements

- 5/1 녹화동영상 강의 (Labor's Day – 학교 휴강 지침)
- 5/6 녹화동영상 강의 (대체공휴일)

- 5월 중 Quiz
- 5월 중하순 Midterm exam #2