

CS2105

An *Awesome* Introduction to Computer Networks

Lectures 2&3: The Application Layer

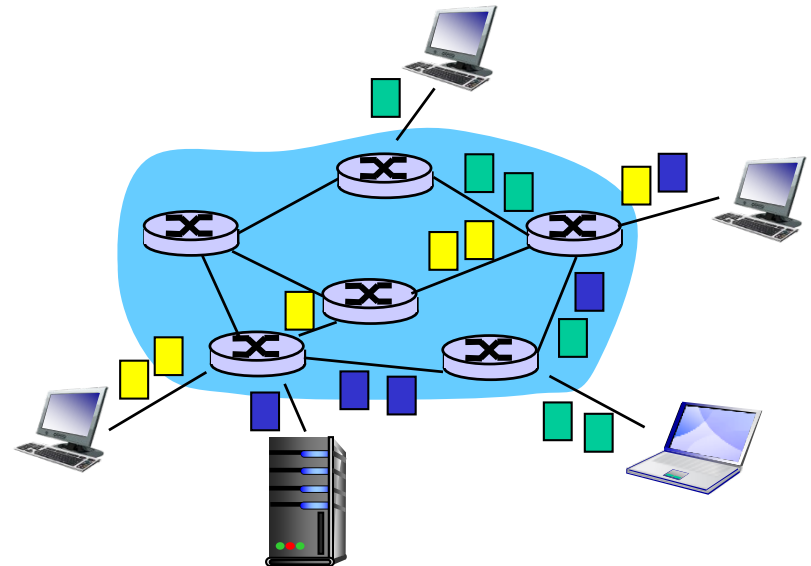


Department of Computer Science
School of Computing

Packet Switching

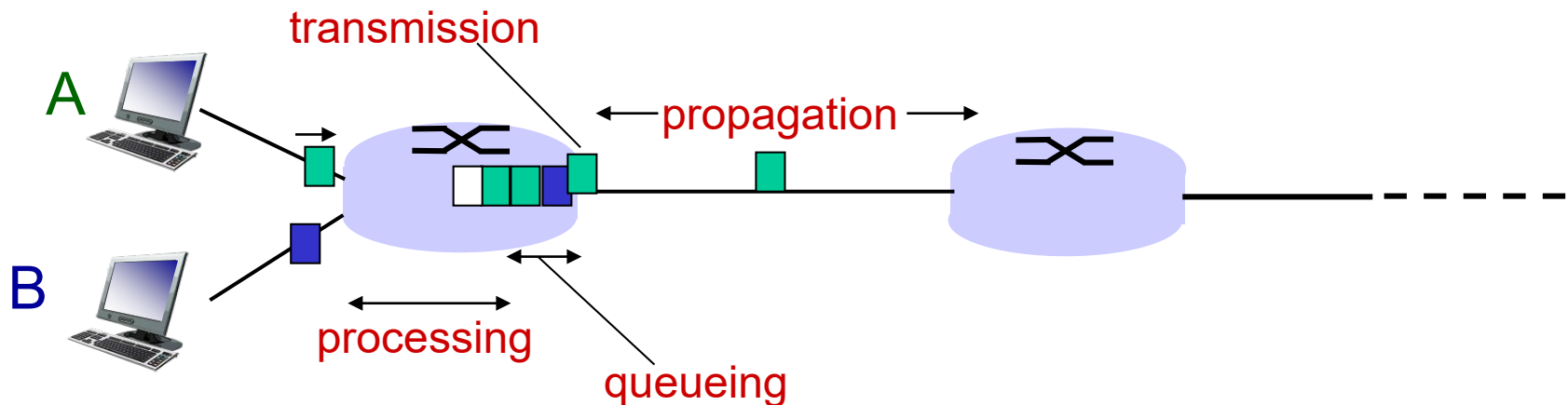
- ❖ The Internet is a **packet switching** network
 - Hosts share and contend network resources.
 - **Application message** is broken into a bunch of packets and sent onto the link one by one.
 - A router stores and forwards packets.
 - Receiver assembles all the packets to restore the **application message**.

Bandwidth division into
“pieces”
Dedicated allocation
Resource reservation



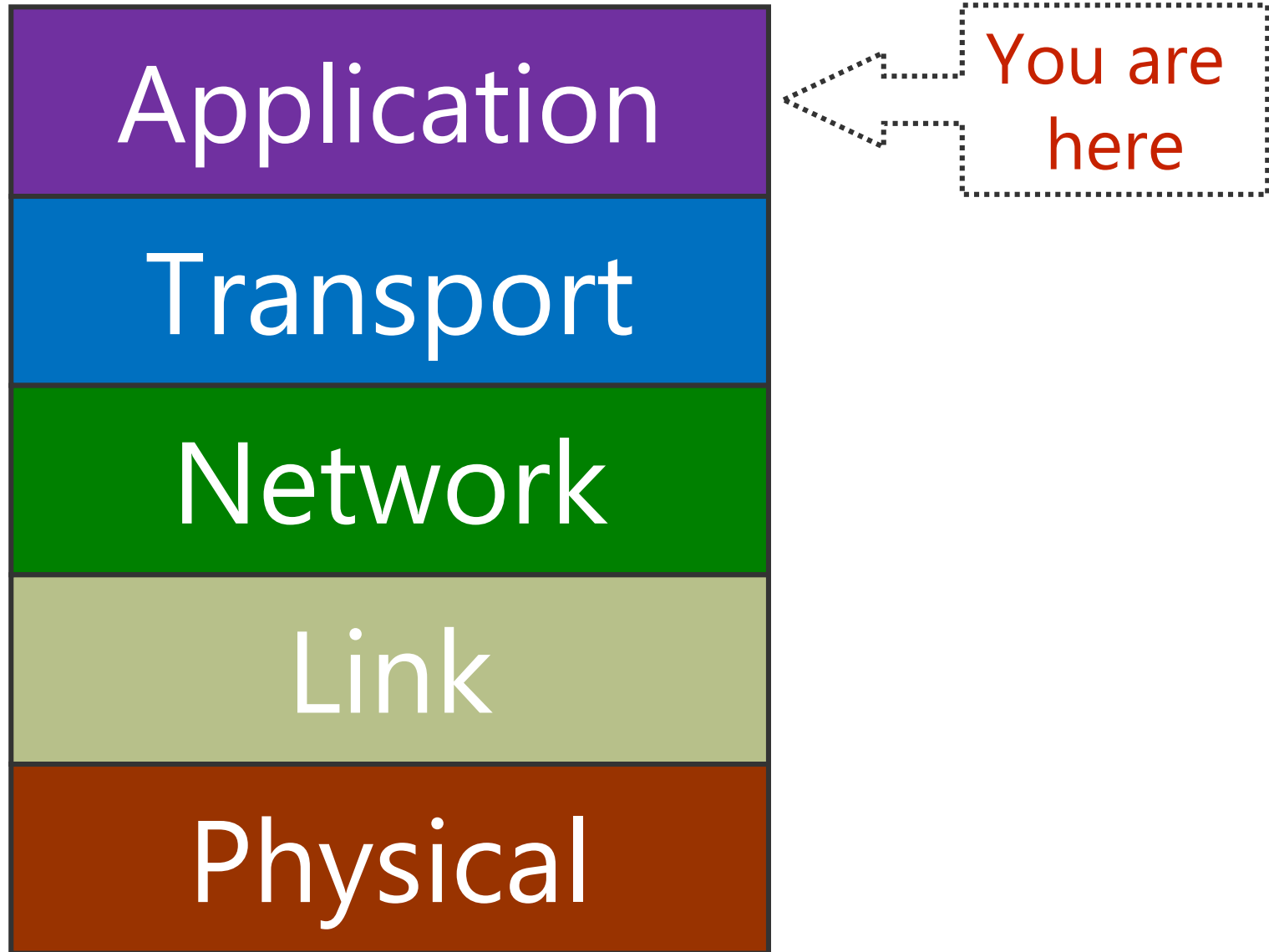
Packet Delay

- ❖ End-to-end delay is the time taken for a packet to travel from source to destination. It consists of:
 - processing delay
 - queueing delay
 - transmission delay
 - propagation delay



Network Protocols

- ❖ Networks are complex. There are many issues to consider, to support different applications running on large number of hosts through different access technology and physical media.
- ❖ **Protocols** regulate communication activities in a network.
 - Define the *format* and *order* of messages exchanged between hosts for a specific purpose.



Lectures 2&3: Application Layer

After this class, you are expected to:

- ❖ understand the basic HTTP interactions between the client and the server, the concepts of persistent and non-persistent connections.
- ❖ understand the services provided by DNS and how a DNS query is resolved.
- ❖ understand the concept of socket.
- ❖ be able to write simple client/server programs through socket programming.

Lectures 2&3: Roadmap

2.1 Principles of Network Applications

2.2 Web and HTTP

2.4 DNS

2.7 Socket programming



To discuss
next week

Kurose Textbook, Chapter 2
(Some slides are taken from the book)

Evolution of Network Applications

❖ Early days of Internet

- Remote access (e.g. telnet, now ssh)

❖ 70s – 80s

- Email, FTP

❖ 90s

- Web

❖ 2000s – now

- P2P file sharing
- Online games
- Instant Messaging, Skype
- YouTube, Facebook

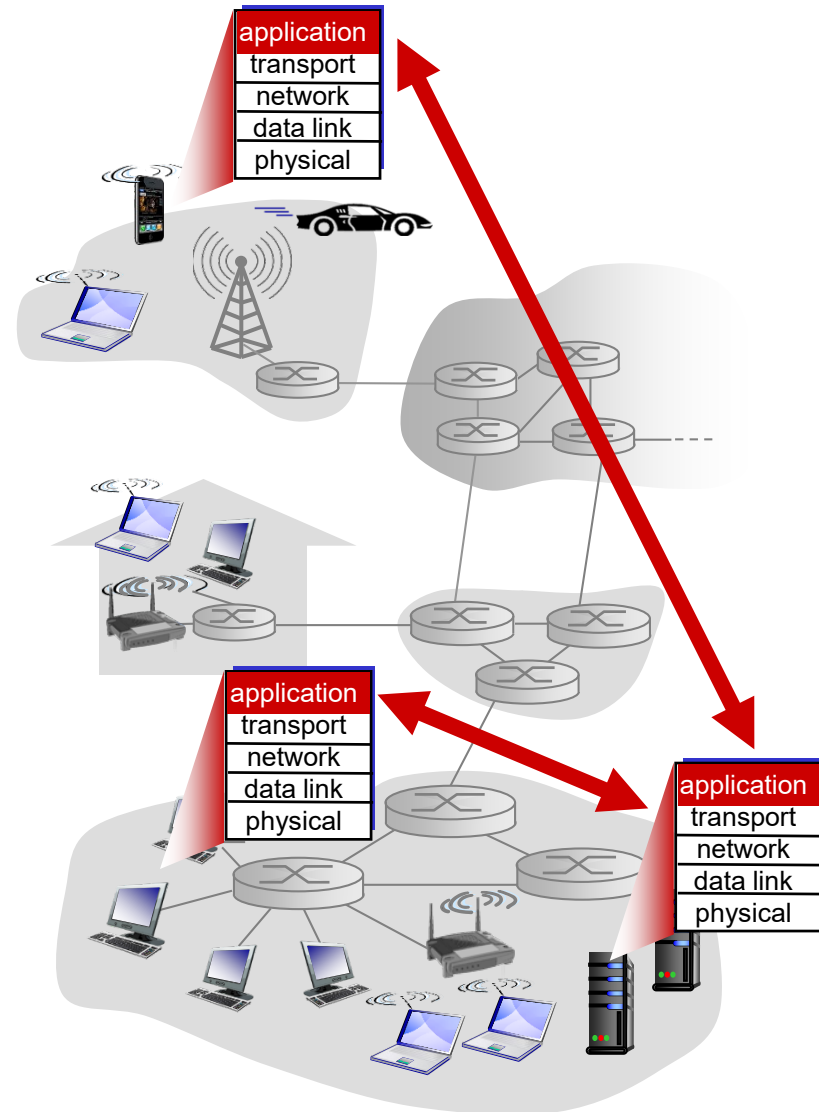
Creating Network Applications

write programs that:

- ❖ run on (different) *hosts*
- ❖ communicate over network
- ❖ e.g., web server software ↔ browser software

classic structure of network applications:

- ❖ Client-server
- ❖ Peer-to-peer (P2P)



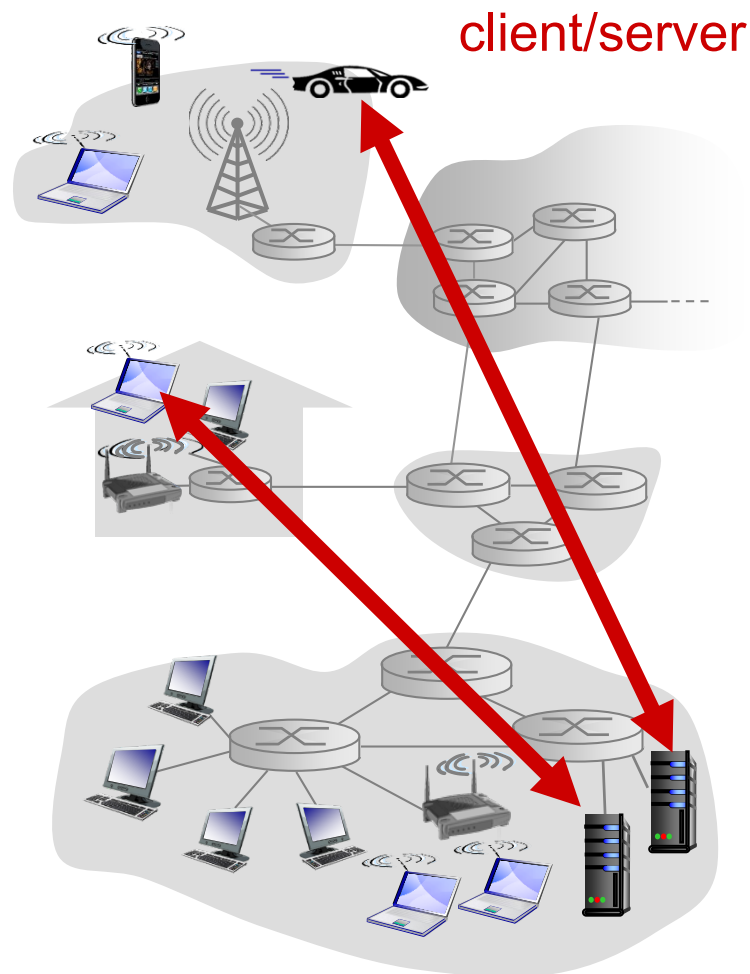
Client-Server Architecture

Server:

- ❖ Waits for incoming requests
- ❖ Provides requested service to client
- ❖ data centers for scaling

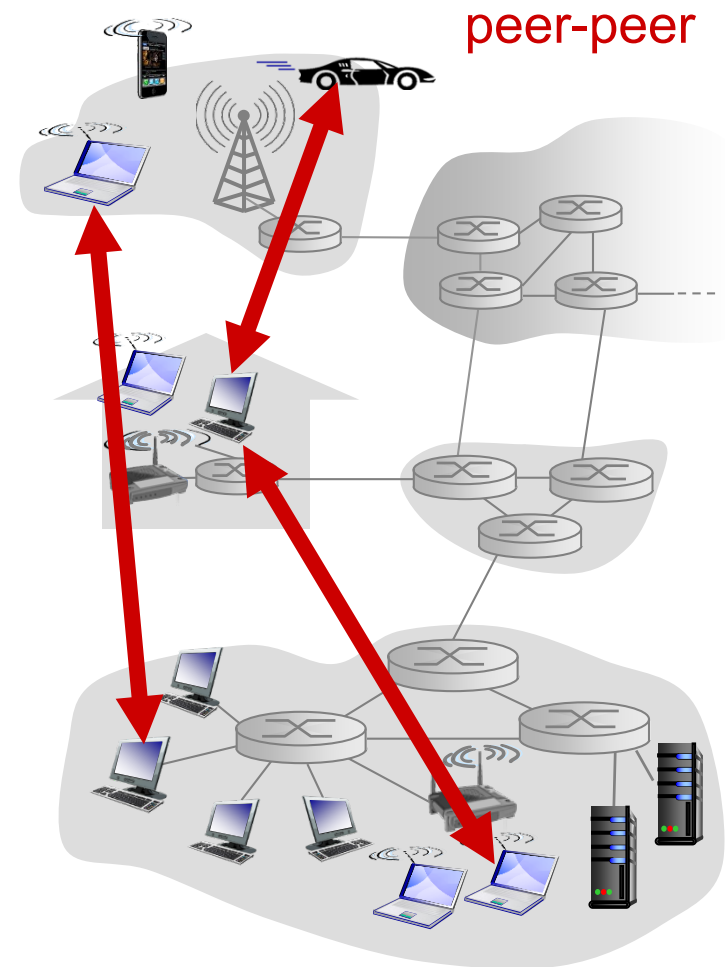
Client:

- ❖ Initiates contact with server (“speaks first”)
- ❖ Typically requests service from server
- ❖ For Web, client is usually implemented in browser



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, as well as new service demands
- ❖ Peers are intermittently connected and change IP addresses
 - complex management



What transport service does an app need?

Data integrity

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

Timing

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

Throughput

- ❖ some apps (e.g., multimedia) require minimum amount of bandwidth to be “effective”
- ❖ other apps (e.g., file transfer) make use of whatever throughput available

Security

- ❖ encryption, data integrity, authentication ...

Requirements of Example Apps

Application	Data loss	Throughput	Time-sensitive
File transfer	No loss	Elastic	No
Electronic mail	No loss	Elastic	No
Web documents	No loss	Elastic	No
Real-time audio/video	Loss-tolerant	Audio: 5kbps-1Mbps Video: 10kbps-5Mbps	Yes: 100s of msec
Stored audio/video	Loss-tolerant	Same as above	Yes: few seconds
Interactive games	Loss-tolerant	Few kbps – 10 kbps	Yes: 100s of msec
Text messaging	No loss	Elastic	Yes and no

App-layer Protocols Define...

- ❖ **types of messages exchanged**
 - e.g., request, response
- ❖ **message syntax:**
 - what fields in messages & how fields are delineated
- ❖ **message semantics**
 - meaning of information in fields
- ❖ **rules** for when and how applications send & respond to messages
- ❖ **open protocols:**
 - defined in RFCs
 - allows for interoperability
 - e.g., HTTP, SMTP
- ❖ **proprietary protocols:**
 - e.g., Skype

Transport Layer Protocols: TCP/UDP

- ❖ App-layer protocols ride on transport layer protocols:

TCP service:

- ❖ *reliable data transfer*
- ❖ *flow control*: sender won't overwhelm receiver
- ❖ *congestion control*: throttle sender when network is overloaded
- ❖ *does not provide*: timing, minimum throughput guarantee, security

UDP service:

- ❖ *unreliable data transfer*
- ❖ *no flow control*
- ❖ *no congestion control*
- ❖ *does not provide*: timing, throughput guarantee or security

Example App/Transport Protocols

Application	Application Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP [RFC 5321]	TCP
Remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
File transfer	FTP [RFC 959]	TCP
Streaming multimedia	HTTP (e.g., YouTube)	Typically TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary (e.g., Skype)	TCP or UDP

Lectures 2&3: Roadmap

2.1 Principles of Network Applications

2.2 Web and HTTP

2.4 DNS

2.7 Socket programming

The Web: Some Jargon

- ❖ A Web page typically consists of:
 - *base HTML file*, and
 - *several referenced objects*.
- ❖ An object can be HTML file, JPEG image, Java applet, audio file, ...
- ❖ Each object is addressable by a *URL*, e.g.,

`www.comp.nus.edu.sg/~cs2105/img/doge.jpg`

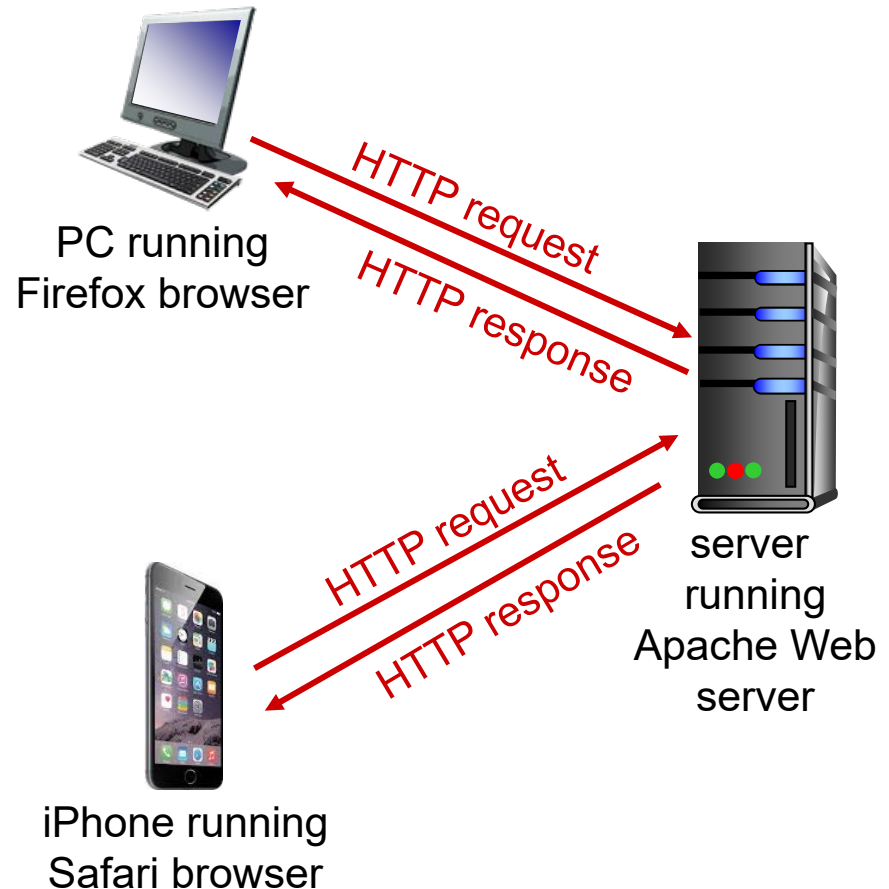
host name

path name

HTTP Overview

HTTP: Hypertext transfer protocol

- ❖ Web's application layer protocol
- ❖ Client/server model
 - *client*: usually is browser that requests, receives and displays Web objects
 - *server*: Web server sends objects in response to requests
- ❖ http 1.0: RFC 1945
- ❖ http 1.1: RFC 2616



HTTP Over TCP

HTTP uses TCP as transport service

- ❖ Client initiates TCP connection to server.
- ❖ Server accepts TCP connection request from client.
- ❖ HTTP messages are exchanged between browser (HTTP client) and Web server (HTTP server) over TCP connection.
- ❖ TCP connection closed.

Two Versions of HTTP

Non-persistent HTTP

- ❖ at most one object sent over a TCP connection
 - connection then closed
- ❖ downloading multiple objects required multiple connections

Persistent HTTP

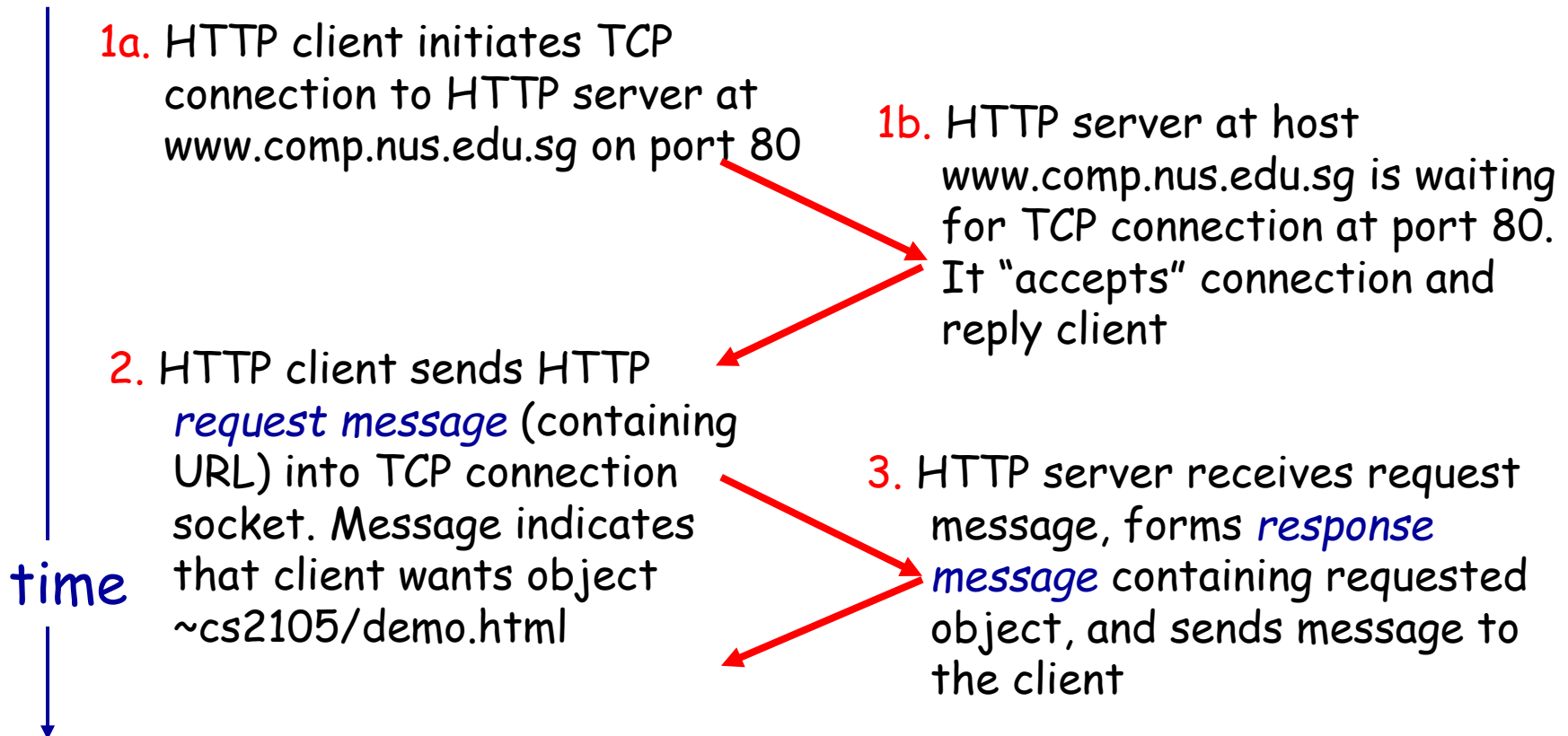
- ❖ multiple objects can be sent over single TCP connection between client, server

Non-persistent HTTP Example

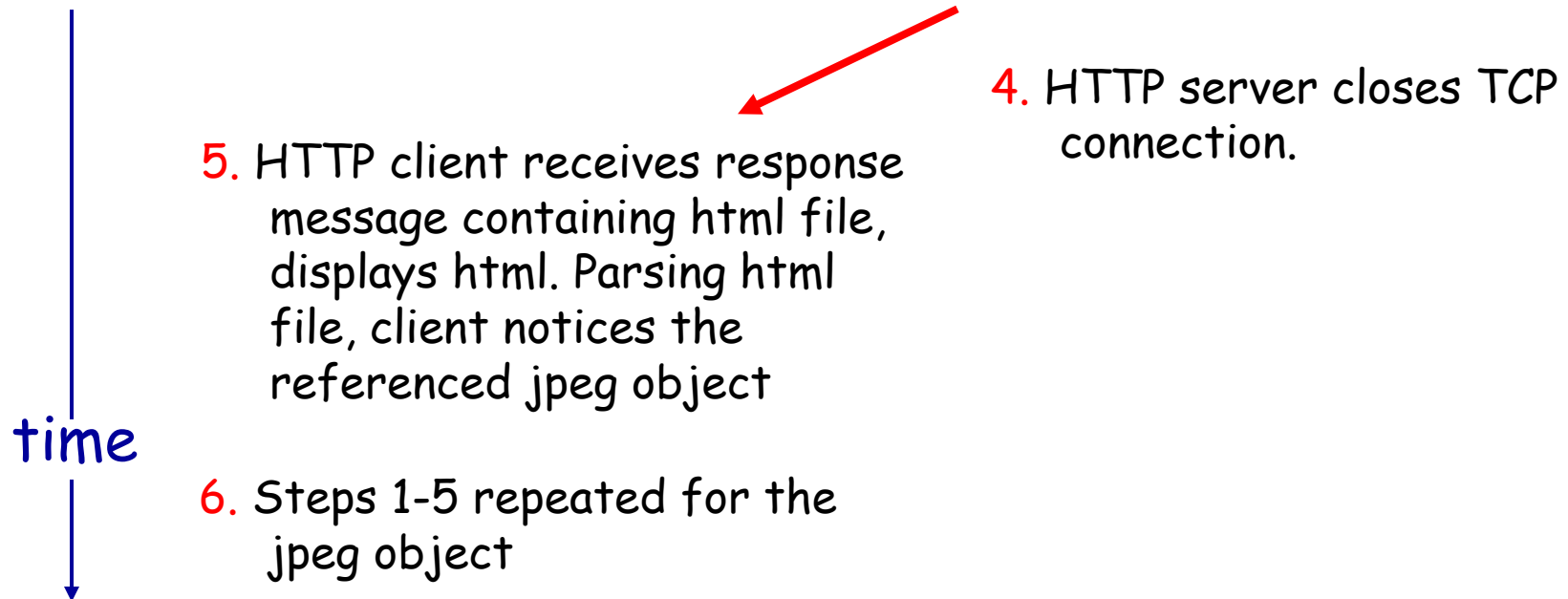
suppose user enters URL:

`www.comp.nus.edu.sg/~cs2105/demo.html`

(contains text,
reference to a
jpeg image)



Non-persistent HTTP Example



- ❖ This is an example of *non-persistent connection* (http 1.0).
 - One object per connection
- ❖ HTTP 1.1 allows *persistent connection* (to discuss).
 - Multiple objects per connection

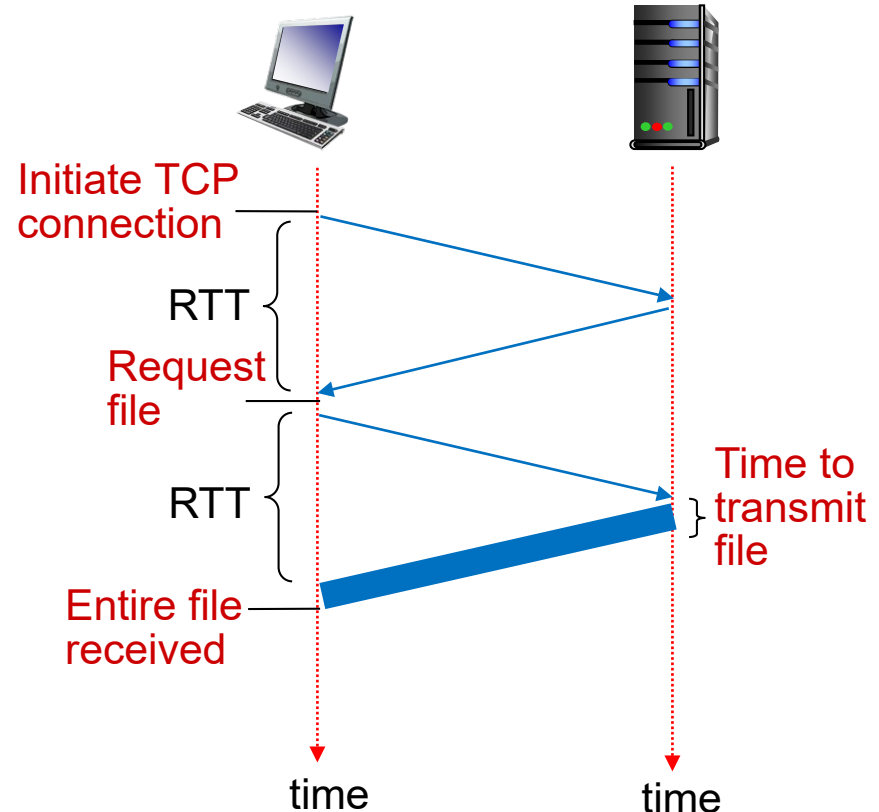
Non-persistent HTTP: Response Time

RTT: time for a packet to travel from client to server and go back

HTTP response time:

- ❖ one RTT to establish TCP connection
- ❖ one RTT for HTTP request and the first few bytes of HTTP response to return
- ❖ file transmission time
- ❖ non-persistent HTTP response time =

$$2 * \text{RTT} + \text{file transmission time}$$



Persistent HTTP

non-persistent HTTP

issues:

- ❖ requires 2 RTTs per object
- ❖ OS overhead for *each* TCP connection
- ❖ browsers often open parallel TCP connections to fetch referenced objects

persistent HTTP:

- ❖ server leaves connection open after sending response
- ❖ subsequent HTTP messages between same client/server sent over the same TCP connection
- ❖ moreover, client may send requests as soon as it encounters a referenced object (**persistent with pipelining**)
 - as little as one RTT for all the referenced objects

Example HTTP Request Message

- ❖ Two types of HTTP messages: *request, response*
- ❖ HTTP request message:

request line
(GET method)

GET /index.html HTTP/1.1\r\n

header
lines

Host: www.example.org\r\n

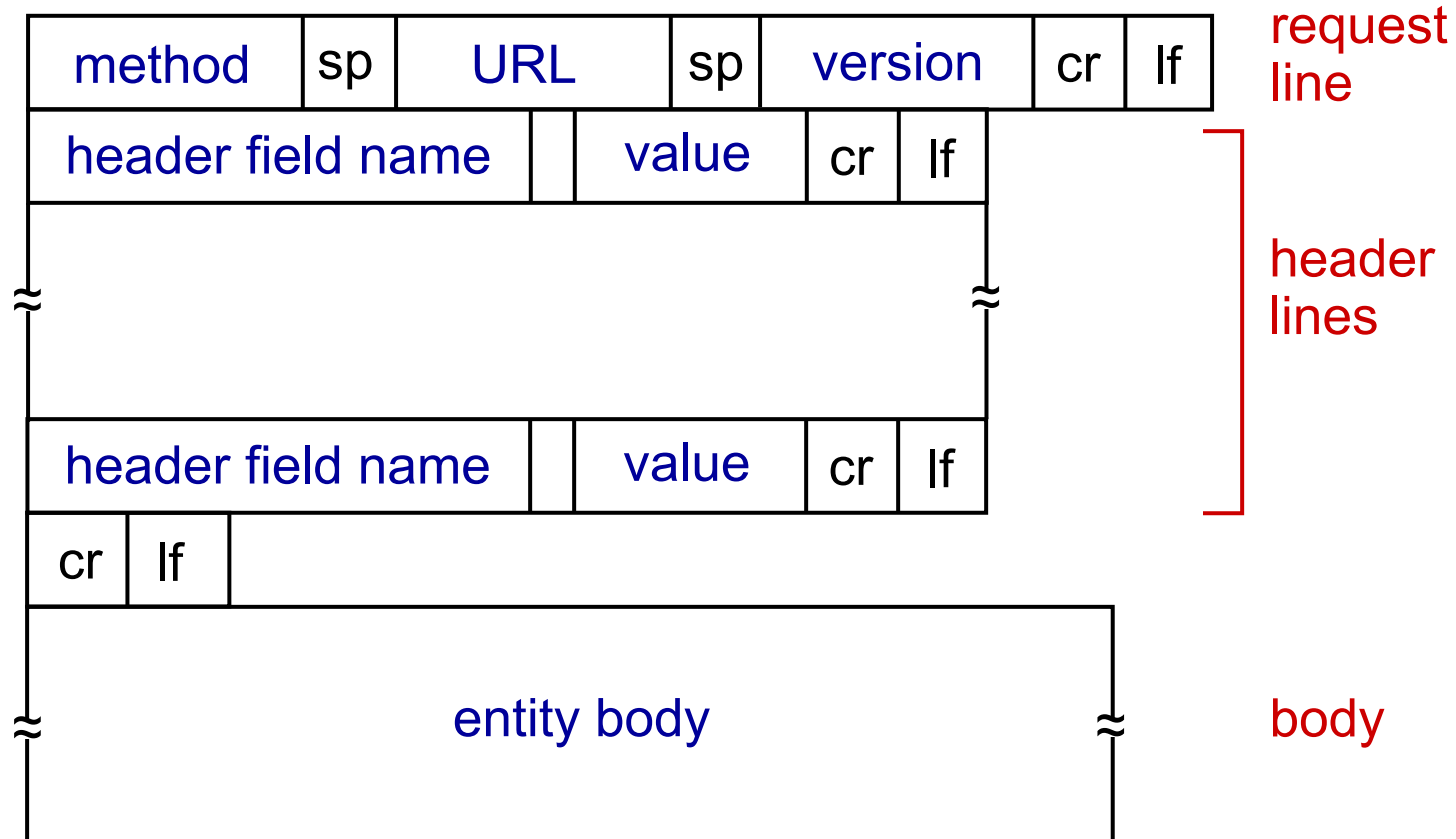
Connection: keep-alive\r\n

...

\r\n

Extra "blank" line indicates
the end of header lines

HTTP Request Message: General Format



HTTP Request Method Types

HTTP/1.0:

- ❖ GET
- ❖ POST
 - web page often includes form input
 - input is uploaded to server in entity body
- ❖ HEAD
 - asks server to leave requested object out of response

HTTP/1.1:

- ❖ GET, POST, HEAD
- ❖ PUT
 - uploads file in entity body to path specified in URL field
- ❖ DELETE
 - deletes file specified in the URL field

Example HTTP Response Message

status line

(protocol status code)

HTTP/1.1 200 OK\r\n

header
lines

Date: Wed, 23 Jan 2019 13:11:15 GMT\r\n

Content-Length: 606\r\n

Content-Type: text/html\r\n

...

\r\n

data data data data data ...

data, e.g.
requested
HTML file

For a full list of request/response header fields, check
<http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>

HTTP Response Status Codes

- ❖ Status code appears in 1st line in server-to-client response message.
- ❖ Some sample codes:

200 OK

- request succeeded, requested object later in this msg

301 Moved Permanently

- requested object moved, new location specified later in this msg (Location:)

403 Forbidden

- server declines to show the requested webpage

404 Not Found

- requested document not found on this server

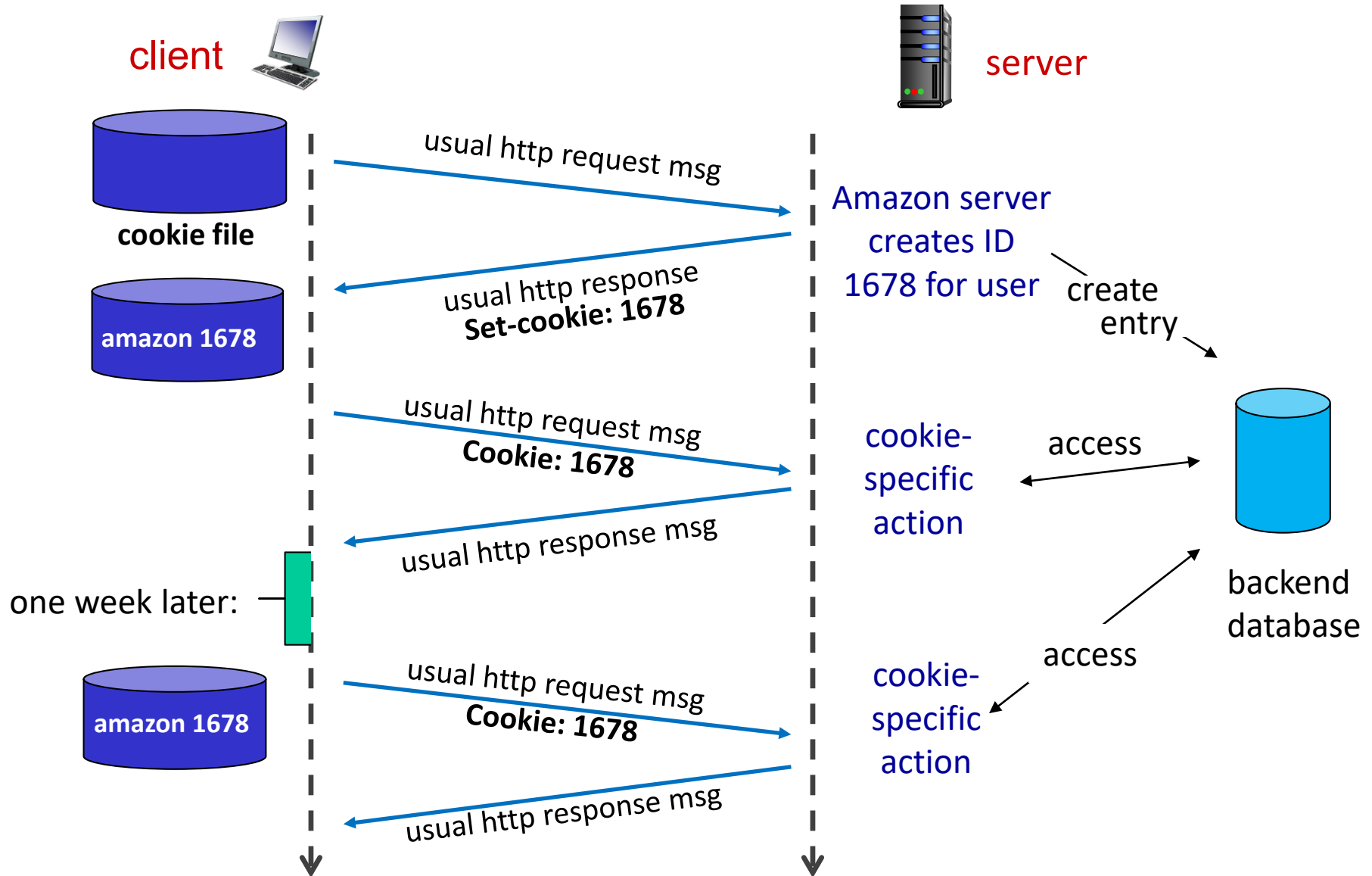
For a full list of status codes, check

<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

Cookies

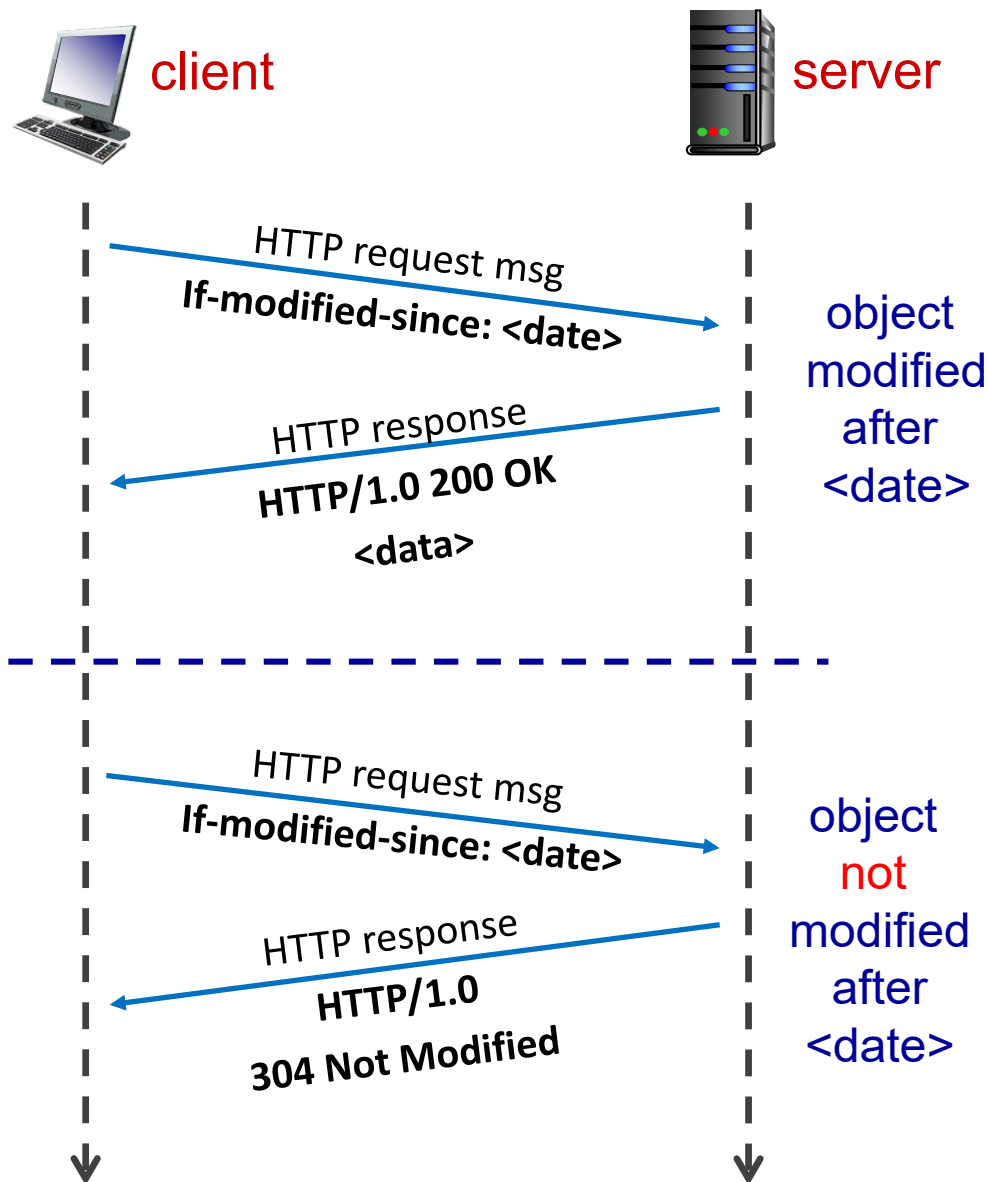
- ❖ HTTP is designed to be “stateless”.
 - Server maintains no information about past client requests.
- ❖ Sometimes it’s good to maintain states (history) at server/client over multiple transactions.
 - E.g. shopping carts
- ❖ Cookie: http messages carry “state”
 - 1) cookie header field of HTTP *request / response* messages
 - 2) cookie file kept on user’s host, managed by user’s browser
 - 3) back-end database at Web site

Keep User States with Cookie



Conditional GET

- ❖ *Goal*: don't send object if (client) cache has up-to-date cached version
- ❖ *cache*: specify date of cached copy in HTTP request
`If-modified-since: <date>`
- ❖ *server*: response contains no object if cached copy is up-to-date:
`HTTP/1.0 304 Not Modified`



Lectures 2&3: Roadmap

2.1 Principles of Network Applications

2.2 Web and HTTP

2.4 DNS

2.7 Socket programming

Domain Name System [RFC 1034, 1035]

- ❖ Two ways to identify a host:
 - **Hostname**, e.g., www.example.org
 - **IP address**, e.g., 93.184.216.34
- ❖ **DNS (Domain Name System)** translates between the two.
 - A client must carry out a DNS query to determine the IP address corresponding to the server name (e.g., www.example.org) prior to the connection.

DNS: Resource Records (RR)

- ❖ Mapping between host names and IP addresses (and others) are stored as 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., `nus.edu.sg`)
- **value** is hostname of authoritative name server for this domain

type = CNAME

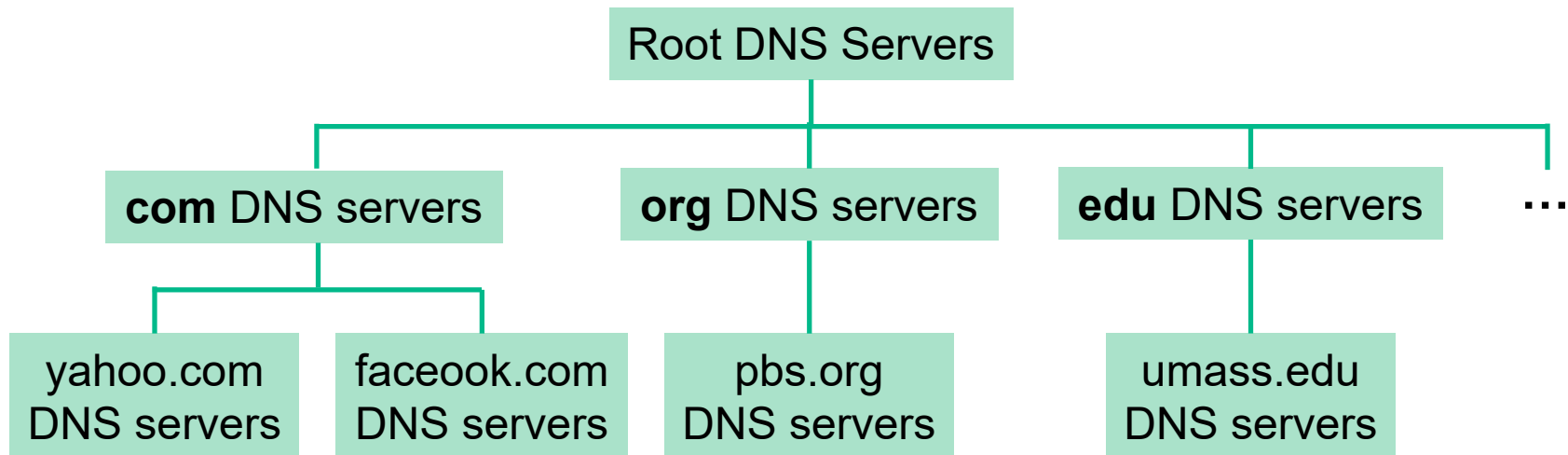
- **name** is alias name (e.g. `www.nus.edu.sg`) for some “canonical” (the real) name
- **value** is canonical name (e.g. `mgnzsqc.x.incapdns.net`)

type = MX

- **value** is name of mail server associated with **name**

Distributed, Hierarchical Database

- ❖ DNS stored RR in distributed databases implemented in hierarchy of many name servers.

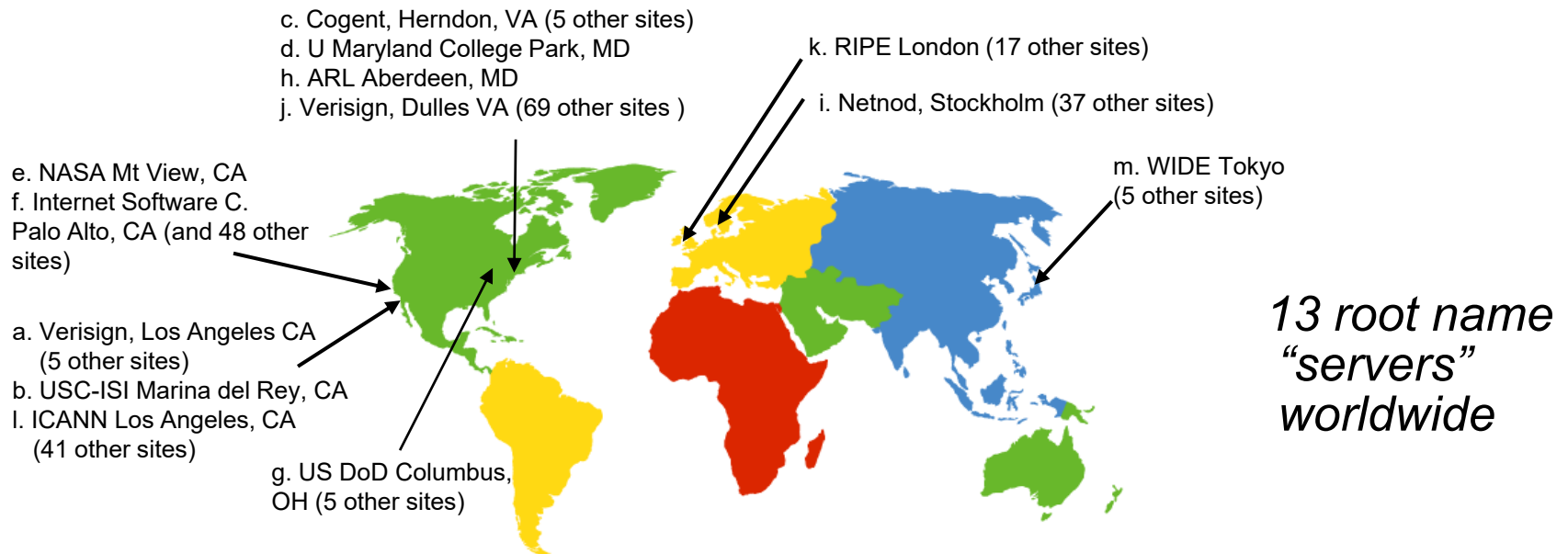


A client wants IP address for www.facebook.com:

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

Root Servers

- ❖ Answers requests for records in the root zone by returning a list of the authoritative name servers for the appropriate top-level domain (TLD).



TLD and Authoritative Servers

Top-level domain (TLD) servers:

- ❖ responsible for com, org, net, edu, ... and all top-level country domains, e.g., uk, sg, jp

Authoritative servers:

- ❖ Organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts (e.g. Web, mail)
- ❖ can be maintained by organization or service provider

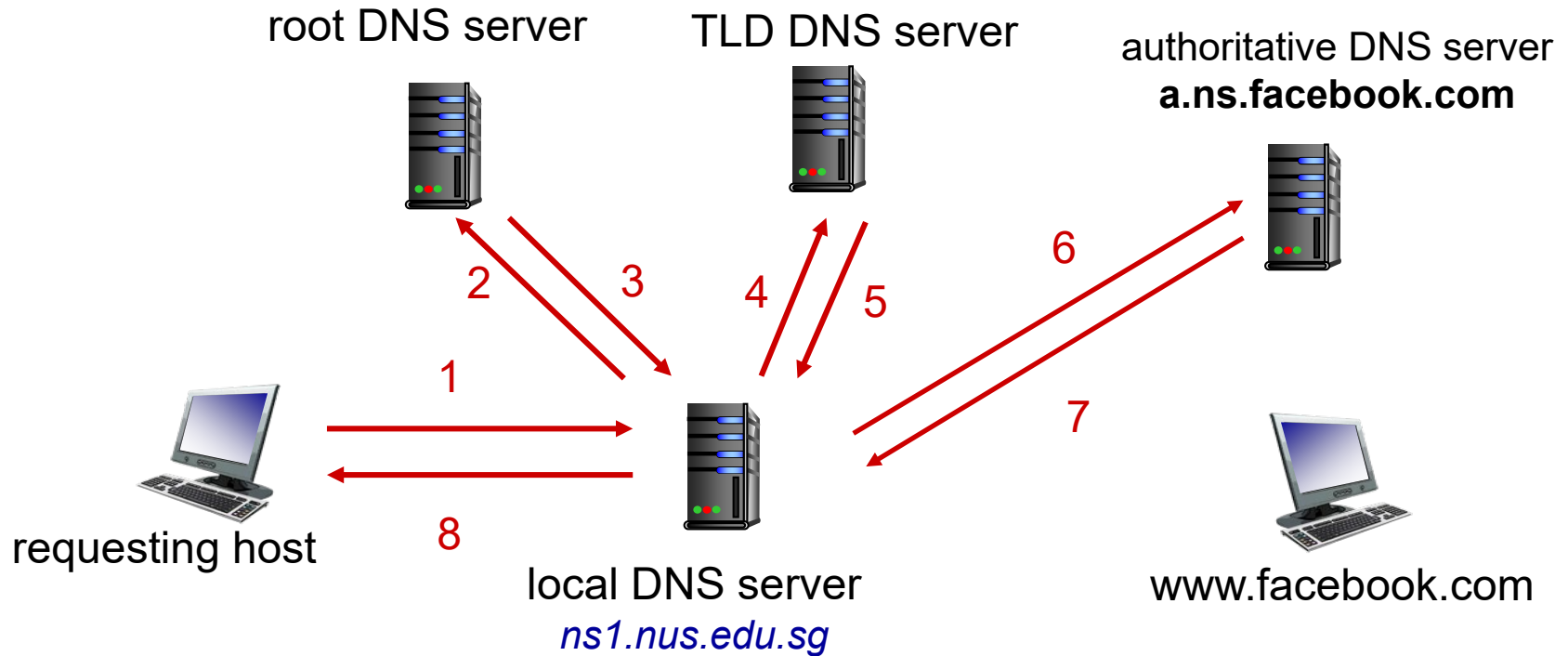
Local DNS Server

- ❖ Does not strictly belong to hierarchy
- ❖ Each ISP (residential ISP, company, university) has one local DNS server.
 - also called “default name server”
- ❖ When host makes a DNS query, query is sent to its local DNS server
 - Retrieve name-to-address translation from local cache
 - Local DNS server acts as proxy and forwards query into hierarchy if answer is not found locally

DNS Caching

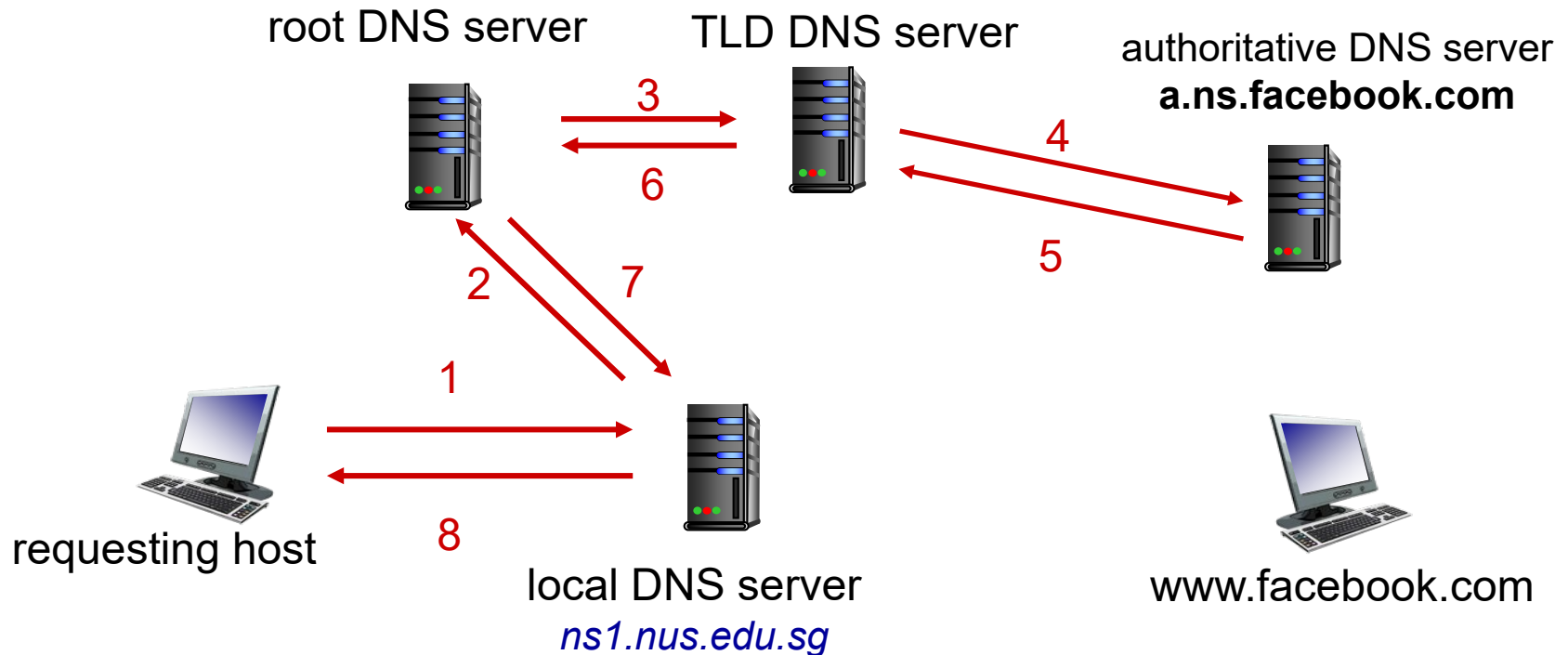
- ❖ Once a name server learns mapping, it *caches* mapping.
 - cached entries may be out-of-date (best effort name-to-address translation!)
 - cached entries expire after some time (TTL).
 - if name host changes IP address, may not be known Internet-wide until all TTLs expire.
- ❖ Update/notify mechanisms proposed IETF standard
 - RFC 2136
- ❖ DNS runs over **UDP**.

DNS Name Resolution



❖ This is known as *iterative query*.

DNS Name Resolution



❖ This is known as *recursive query*.

- rarely used in practice

Lectures 2&3: Roadmap

2.1 Principles of Network Applications

2.2 Web and HTTP

2.4 DNS

2.7 Socket programming: Creating Network Applications

Processes

- ❖ Process: program running within a host.
 - Within the same host, two processes communicate using **inter-process communication** (defined by OS).
 - Processes in different hosts communicate by exchanging **messages** (according to protocols).

In C/S model

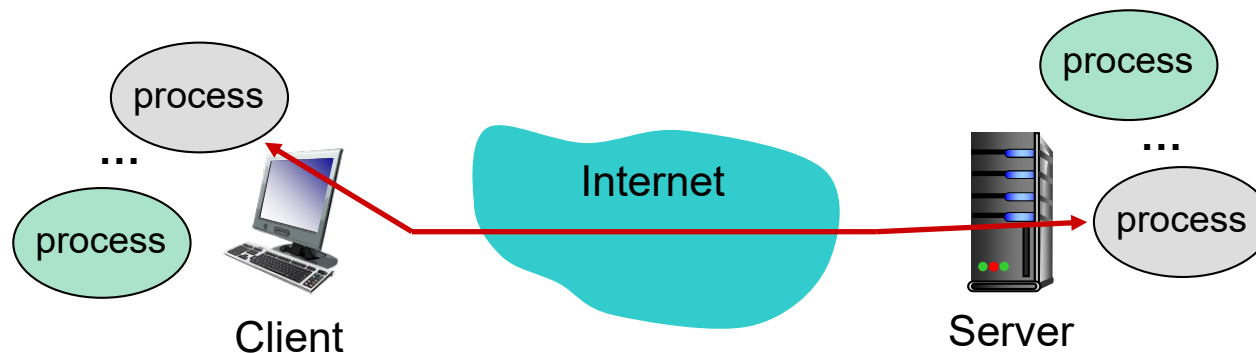
server process waits to be contacted

client process initiates the communication

Addressing Processes

- ❖ **IP address** is used to identify a host
 - A 32-bit integer (e.g. 137.132.21.27)
- ❖ **Question:** is IP address of a host suffice to identify a process running inside that host?

A: no, many processes may run concurrently in a host.



Addressing Processes

- ❖ A process is identified by (IP address, port number).
 - Port number is 16-bit integer (1-1023 are reserved for standard use).
- ❖ Example port numbers
 - HTTP server: 80
 - SMTP server: 25
- ❖ IANA coordinates the assignment of port number:
 - <http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml>

Analogy

Postal service:

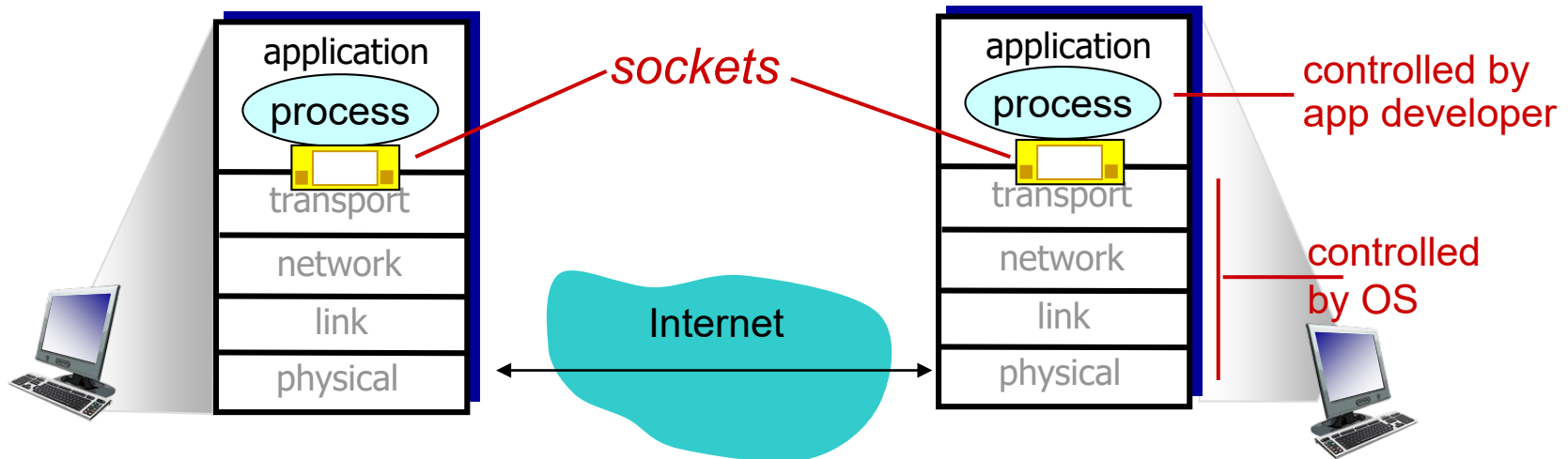
- ❖ *deliver letter to the doorstep:* home address
- ❖ *dispatch letter to the right person in the house:* name of the receiver as stated on the letter

Protocol service:

- ❖ *deliver packet to the right host:* IP address of the host
- ❖ *dispatch packet to the right process in the host:* port number of the process

Sockets

- ❖ **Socket** is the software interface between app processes and transport layer protocols.
 - Process sends/receives messages to/from its **socket**.
 - Programming-wise: a set of **APIs**



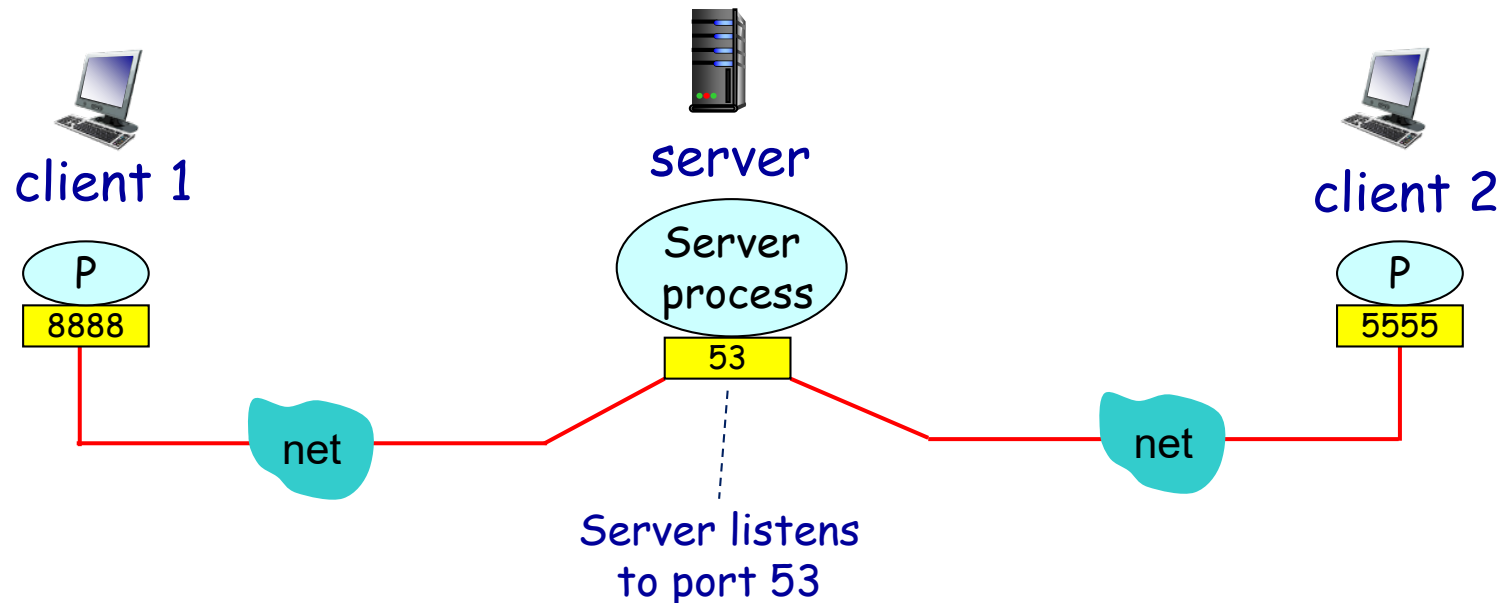
Socket Programming

- ❖ Applications (or processes) treat the Internet as a black box, sending and receiving messages through sockets.
- ❖ Two types of sockets
 - **TCP**: reliable, byte stream-oriented socket
 - **UDP**: unreliable datagram socket
- ❖ Now let's write a simple client/server application that **client sends a line of text to server, and server echoes it.**
 - We will demo both **TCP socket version** and **UDP socket version**.

Socket Programming with *UDP*

UDP: no “connection” between client and server

- ❖ Sender (client) explicitly attaches destination IP address and port number to each packet.
- ❖ Receiver (server) extracts sender IP address and port number from the received packet.



UDP: Client/server Socket Interaction

Server (running on `serverIP`)

create **serverSocket**,
port = **x**:

read datagram from
serverSocket

write reply to **serverSocket**
specifying client address,
port number

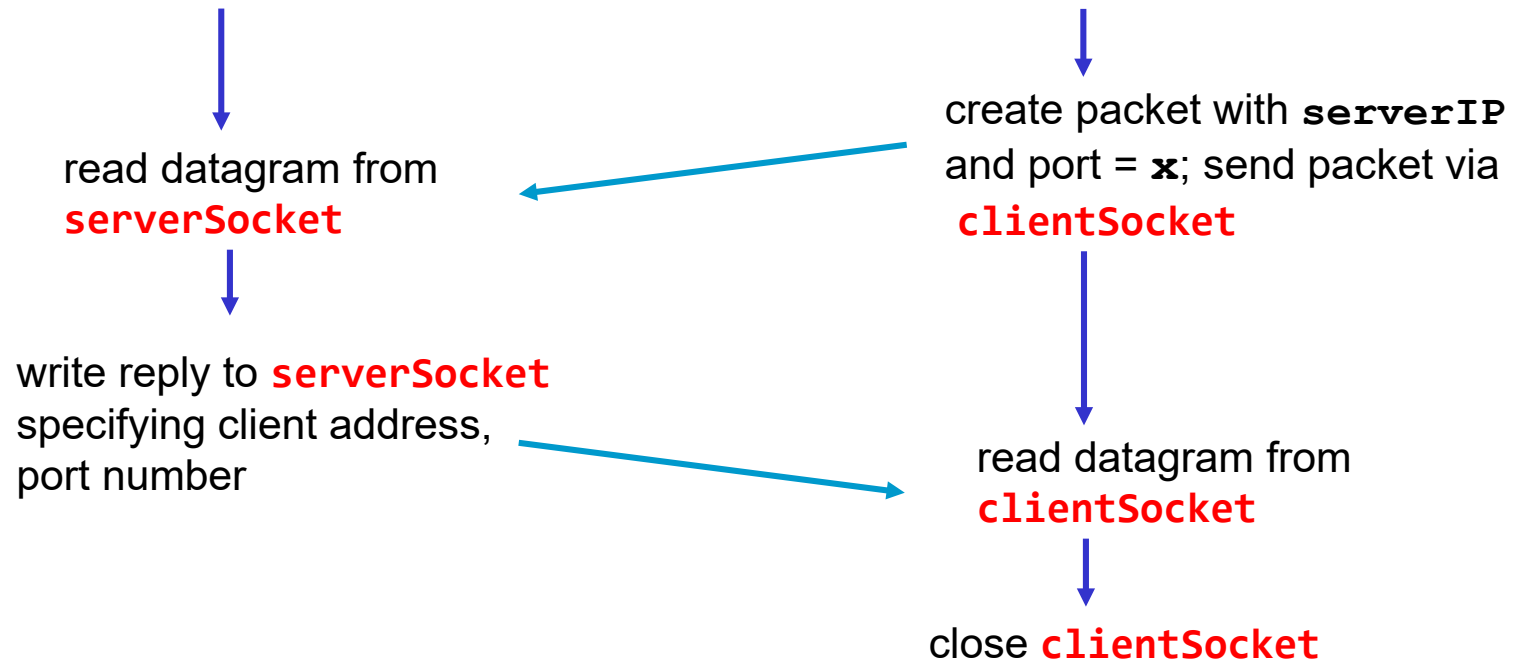
Client

create **clientSocket**

create packet with `serverIP`
and port = **x**; send packet via
clientSocket

read datagram from
clientSocket

close **clientSocket**



Example: UDP Echo Server

```
from socket import * ← include Python's socket library

serverPort = 2105

# create a socket
serverSocket = socket(AF_INET, SOCK_DGRAM)

# bind socket to local port number 2105
serverSocket.bind(('', serverPort))
print('Server is ready to receive message')

# extract client address from received packet
message, clientAddress = serverSocket.recvfrom(2048)

serverSocket.sendto(message, clientAddress)

serverSocket.close()
```

IPv4

UDP socket

receive datagram
buffer size: 2048B

Example: UDP Echo Client

```
from socket import *

serverName = 'localhost' # client, server on the same host
serverPort = 2105

clientSocket = socket(AF_INET, SOCK_DGRAM)

message = input('Enter a message: ')

# send msg to server address
clientSocket.sendto(message.encode(), (serverName, serverPort))

receivedMsg, serverAddress = clientSocket.recvfrom(2048)

print('from server:', receivedMsg.decode())

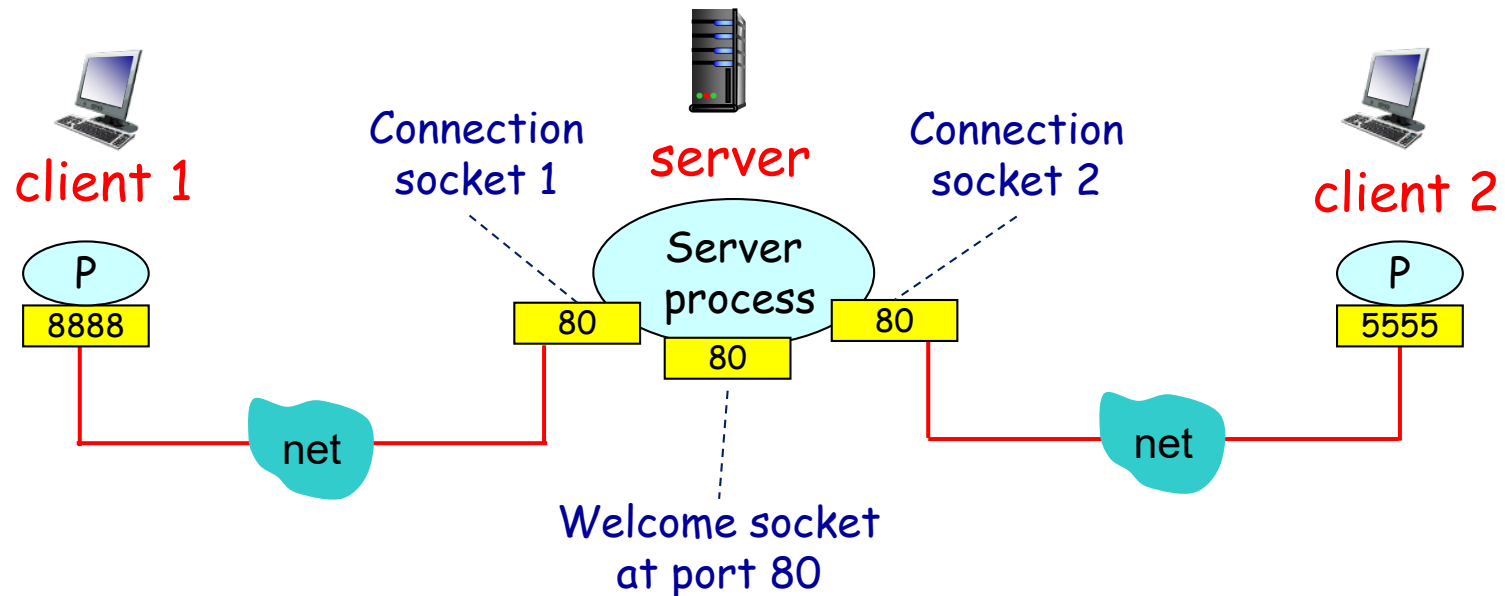
clientSocket.close()
```

convert message from string to bytes and send it

convert from bytes to string

Socket Programming with *TCP*

- ❖ When client creates socket, client TCP establishes a connection to server TCP.
- ❖ When contacted by client, **server TCP creates a new socket** for server process to communicate with that client.
 - allows server to talk with multiple clients individually.



TCP: Client/server Socket Interaction

Server (running on `serverIP`)

Client

create **serverSocket**, port = **x**

wait for incoming
connection request
connectionSocket

read request from
connectionSocket

write reply to
connectionSocket

close **connectionSocket**

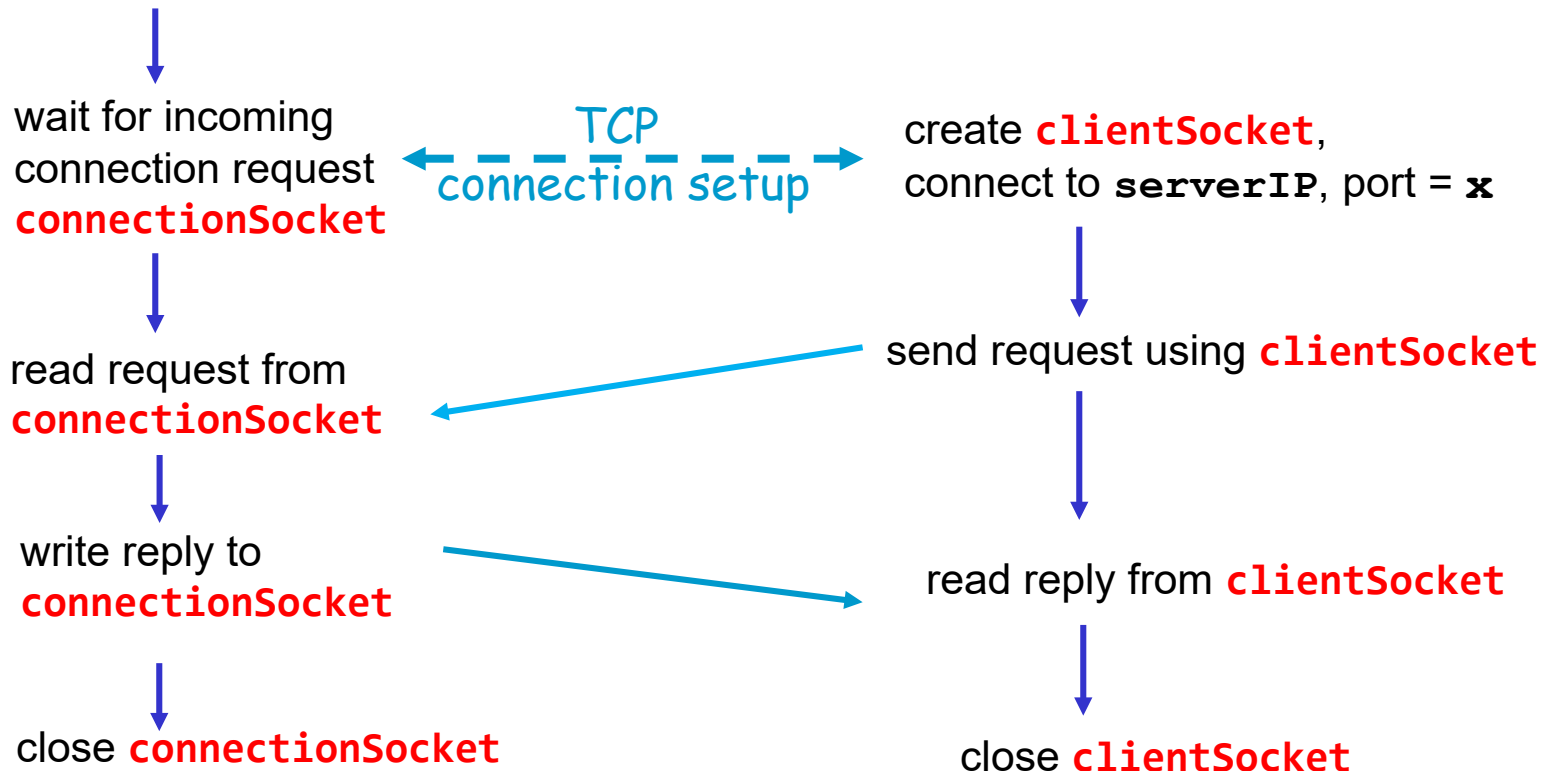
TCP
connection setup

create **clientSocket**,
connect to `serverIP`, port = **x**

send request using **clientSocket**

read reply from **clientSocket**

close **clientSocket**



Example: TCP Echo Server

```
from socket import *
```

```
serverPort = 2105
```

TCP socket



```
serverSocket = socket(AF_INET, SOCK_STREAM)
```

```
serverSocket.bind(('', serverPort))
```

```
serverSocket.listen()
```

listens for incoming TCP request
(not available in UDP socket)



```
print('Server is ready to receive message')
```

```
connectionSocket, clientAddr = serverSocket.accept()
```

```
message = connectionSocket.recv(2048)
```

returns a new socket
to communicate with
client socket



```
connectionSocket.send(message)
```

```
connectionSocket.close()
```

Example: TCP Echo Client

```
from socket import *

serverName = 'localhost'
serverPort = 2105

clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))

message = input('Enter a message: ')

clientSocket.send(message.encode())

receivedMsg = clientSocket.recv(2048)

print('from server:', receivedMsg.decode())

clientSocket.close()
```

← establish a connection

← no need to attach server name, port

TCP Socket vs. UDP Socket

- ❖ In TCP, two processes communicate as if there is a pipe between them. The pipe remains in place until one of the two processes closes it.
 - When one of the processes wants to send more bytes to the other process, it simply writes data to that pipe.
 - The sending process doesn't need to attach a destination IP address and port number to the bytes in each sending attempt as the logical pipe has been established (which is also reliable).
- ❖ In UDP, programmers need to form UDP datagram packets explicitly and attach destination IP address / port number to every packet.

Lectures 2&3: Summary

- ❖ Application architectures
 - Client-server
 - P2P
- ❖ Application service requirements:
 - reliability, throughput, delay, security
- ❖ Specific protocols:
 - HTTP
 - DNS
- ❖ Internet transport service model
 - connection-oriented, reliable: TCP
 - Connection-less, unreliable: UDP

Lectures 2&3: Summary

❖ Socket programming

▪ TCP socket

- When contacted by client, server TCP creates new socket.
- Server uses (client IP + port #) to distinguish clients.
- When client creates its socket, client TCP establishes connection to server TCP.

▪ UDP socket

- Server uses one socket to serve all clients.
- No connection is established before sending data.
- Sender explicitly attaches destination IP address and port # to each packet.
- Transmitted data may be lost or received out-of-order.