Multimedia Communications
@CS.NCTU

Lecture 2: Networking – Application Layer

[Computer Networking, Ch2]

Instructor: Kate Ching-Ju Lin (林靖茹)

Slides modified from
"Computer Networking: A Top-Down Approach" 6th Edition

Outline

- **Principles of network applications**
- Web and HTTP
- P2P Applications (later lecture)
- Video Streaming and CDN (later lecture)
- Socket programing with UDP and TCP

Some Network Applications

- e-mail
- web
- text messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- voice over IP (e.g., Skype)
- real-time video conferencing
- social networking
- search
- …
- …

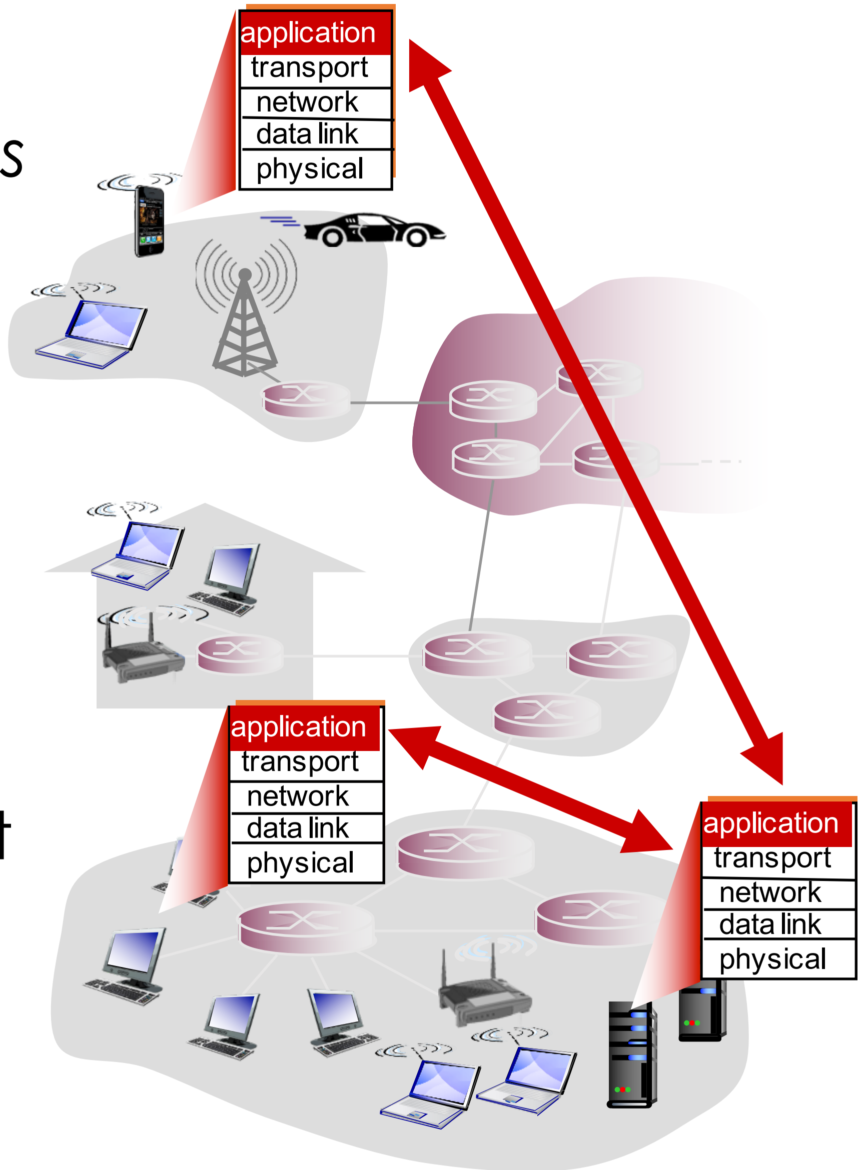
Creating a Network App

write programs that:

- run on (different) *end systems*
- communicate over network
- e.g., web server software communicates with browser software

no need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



Application Architectures

QUIZ

- Application architecture is different from the network architecture (five layer)

- Possible structure of applications:
  - **client-server**
    - Web, gmail, Facebook, etc
  - **peer-to-peer (P2P)**
    - BitTorrent, Skype, PPStream, etc

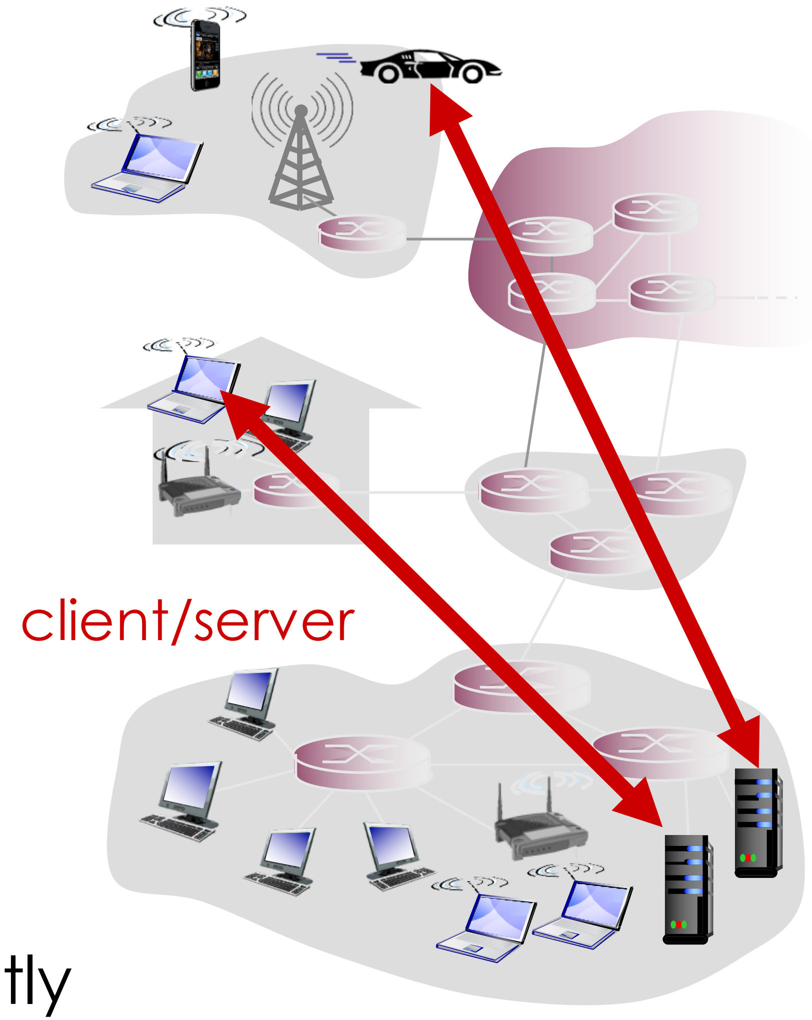
Client-Server Architecture

server:

- always-on host
- permanent IP address
- data centers for scaling

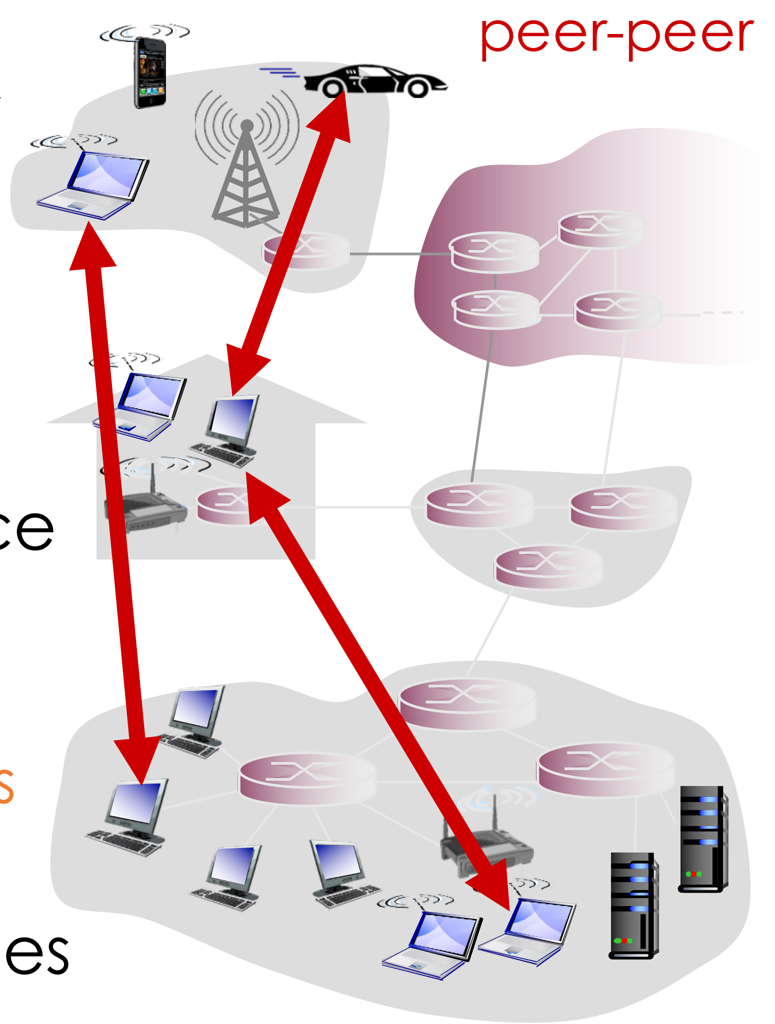
clients:

- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other



Peer-to-Peer (P2P) Architecture

- No always-on server
- Intermittently connected hosts, called **peers** (equally important)
- Arbitrary end systems directly communicate
- Peers request service from other peers, and provide service in return to other peers
    - ☺ **self scalability** – new peers bring new service capacity, as well as new service demands
- Peers might change IP addresses
    - complex management



Challenges of P2P Architecture

Why P2P is less common?

- **ISP Friendly**
  - Most residential ISPs usually support asymmetrical bandwidth, but P2P has high upstream demands, which is not friendly to ISPs
- **Security**
  - Hard to achieve due to the distributed nature of P2P
- **Incentives**
  - Need to convince peers to contribute their bandwidth, storage and computation resource → Human are selfish!

Processes Communicating

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

clients, servers

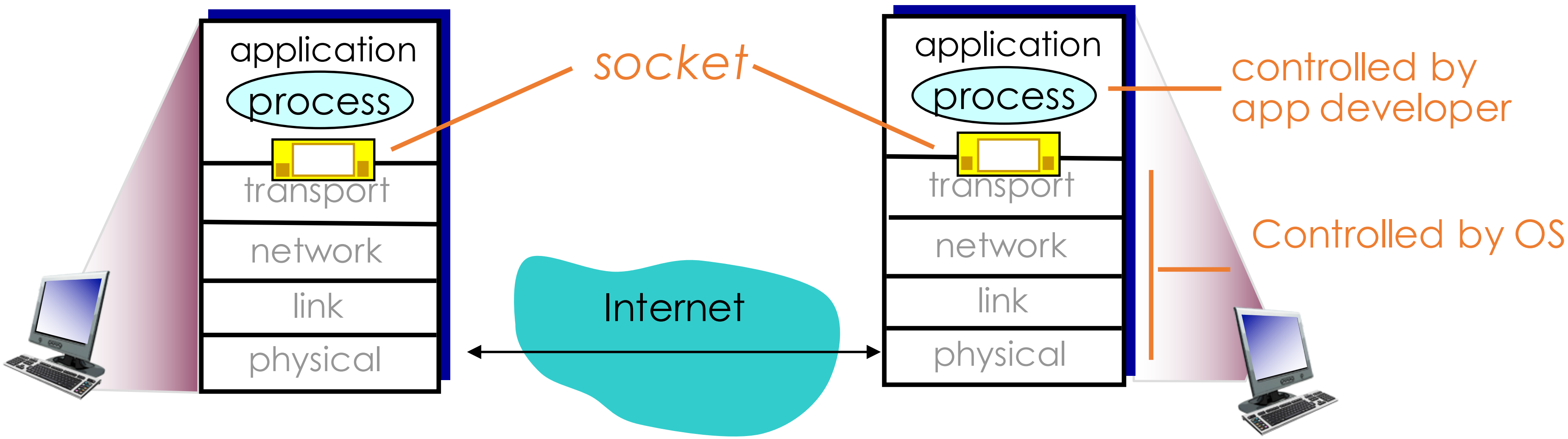
client process: process that *initiates* communication

server process: process that *waits* to be contacted

- applications with P2P architectures have **both** client processes & server processes

Sockets

- Process sends/receives messages to/from its **socket**, an interface between the app and transport layers
- Controls: 1) choice of the transport protocol; 2) setup transmit-layer parameters, e.g., buffer size
- Socket analogous to door
  - sending process pushes message out door, to the door (socket) at receiving process



Addressing Processes

- ***identifier*** includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to *gaia.cs.umass.edu* web server:
  - IP address: 128.119.245.12
  - port number: 80
- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- Q: does IP address of host on which process runs suffice for identifying the process?
- A: no, many processes can be running on same host

App-layer Protocol Defines

- 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 processes send & respond to messages

Open protocols:

- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

Proprietary protocols:

- e.g., Skype

What transport service does an app need?

Reliability

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

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

Throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be "effective"
- other apps ("elastic apps") make use of whatever throughput they get

security

- encryption, data integrity, ...

Transport Service Requirements

application	data loss	throughput	time sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps -5Mbps	yes, 100’s msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100’s msec
text messaging	no loss	elastic	yes and no

Transport Services for Apps

TCP service:

- *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, minimum throughput guarantee, security
- *connection-oriented:* setup required between client and server processes

UDP service:

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

Q: why bother? Why is there a UDP?

Application and Transport Protocols

application	application layer protocol	underlying transport protocol
e-mail	SMTP [RFC 2821]	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), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP

Q: Something **better than TCP and UDP or in-between TCP and UDP**?

Outline

- Principles of network applications
- **Web and HTTP**
- P2P Applications (later lecture)
- Video Streaming and CDN (later lecture)
- Socket programing with UDP and TCP

Web and HTTP

First, a review…

- **Web page** consists of **objects**
- Object can be HTML file, JPEG image, Java applet, audio file,…
- Web page consists of **base HTML-file**, which includes **several referenced objects**
- Each object is addressable by a **URL**, e.g.,

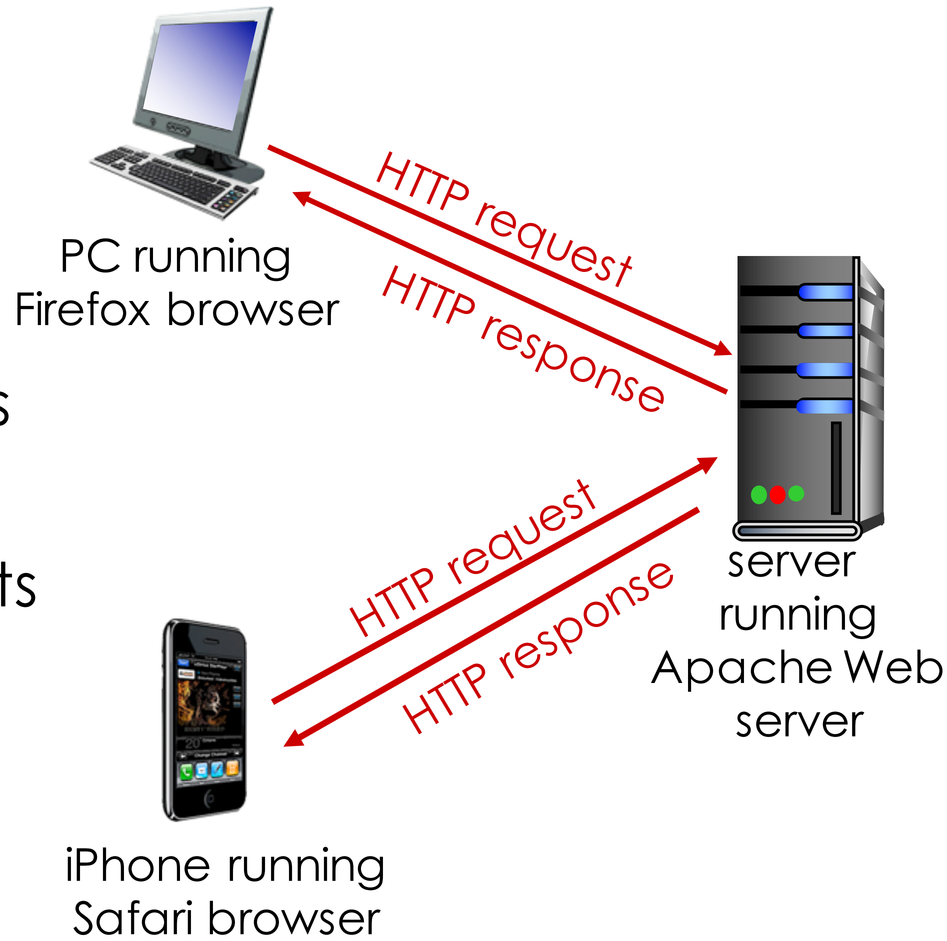
```
www.someschool.edu/someDept/pic.gif
```

host name | path name

HTTP overview

HTTP: HyperText Transfer Protocol

- Web's application layer protocol
- Client/server model
  - ***client:*** browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - ***server:*** Web server sends (using HTTP protocol) objects in response to requests



HTTP Overview (cont.)

- **Uses TCP**
  - Client initiates TCP connection (creates socket) to server, default port 80
  - Server accepts TCP connection from client
  - HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
  - Reliable transmissions
- **HTTP is "stateless"**
  - Server maintains no information about past client requests

aside

protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

HTTP Connections

non-persistent HTTP

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

persistent HTTP

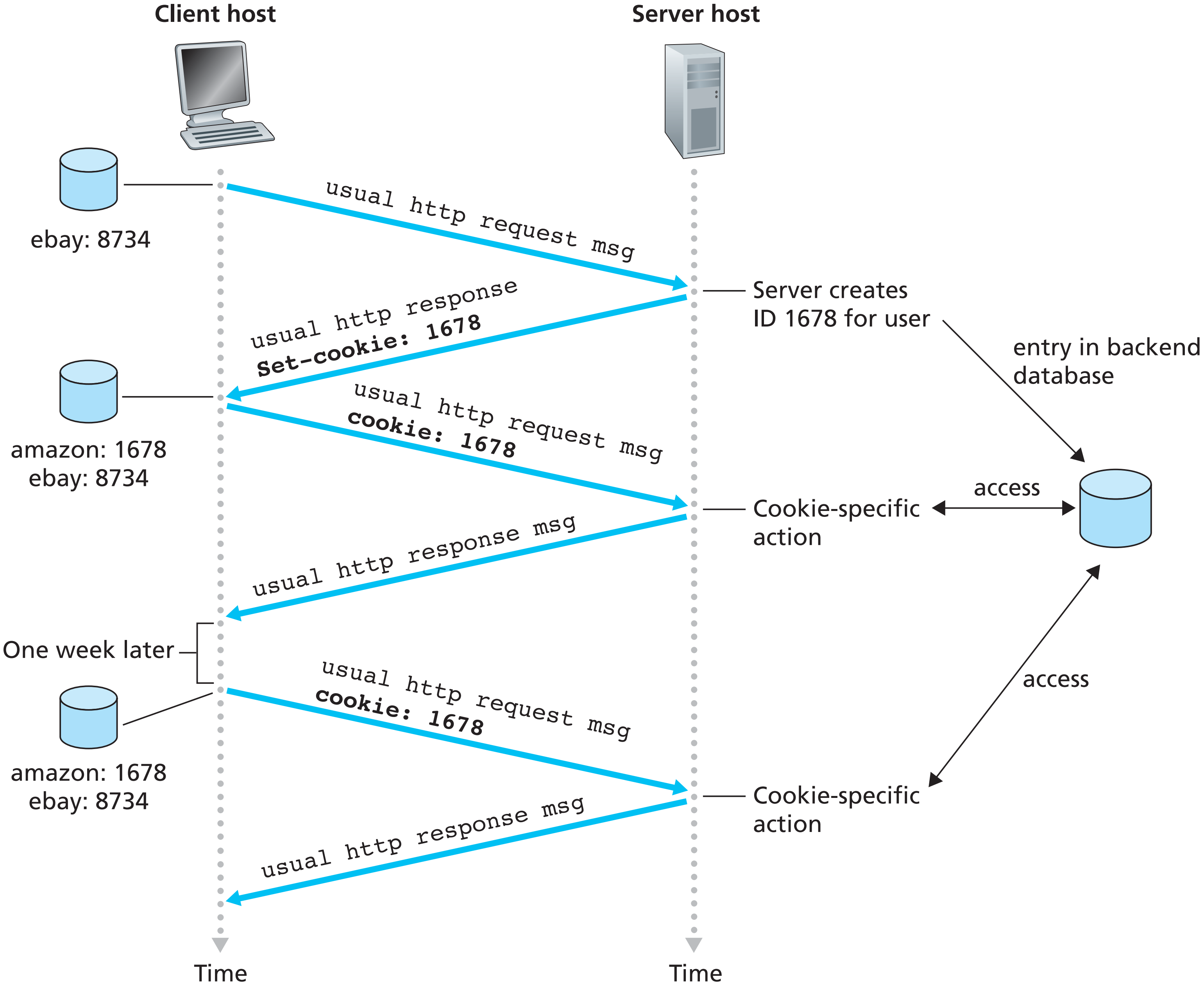
- multiple objects can be sent over single TCP connection between client and server
- the server closes a connection when it isn't used for a certain time
- Default mode

User-Server Interaction: Cookies

- HTTP is stateless, but what if the server wants to keep information, e.g., user ID?
  - Use **cookies**!
  - Track user activity (e.g., interests of pro

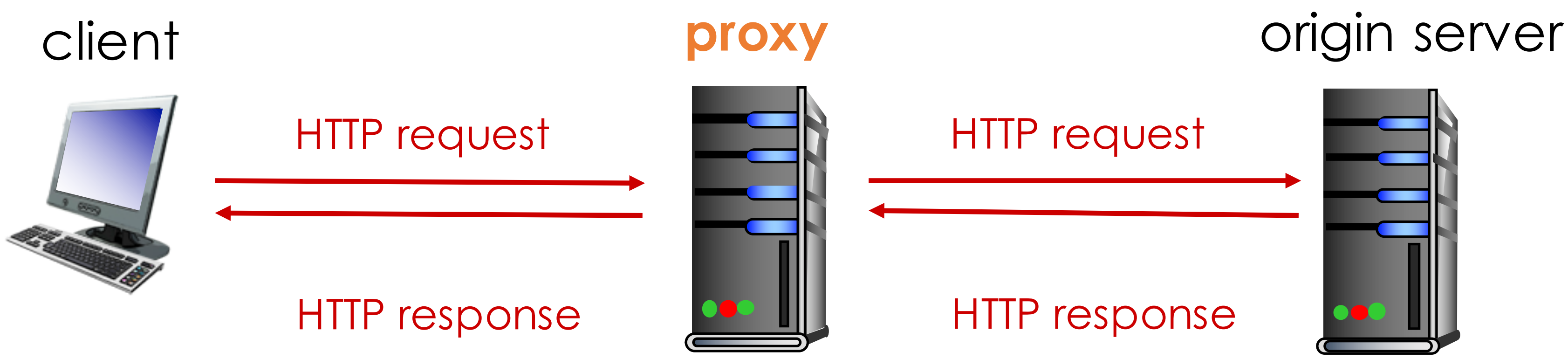
- Four components
  1. a header in the HTTP response message
  2. a header in the HTTP request message
  3. a cookie file kept on the users' browser
  4. a back-end database at the Web site

User-Server Interaction: Cookies



Web Caching

- Also called **proxy server**, a server keeping copies of recently requested objects
- If the browser configures a proxy server,
  1. the request will be first re-directed to the Web cache
  2. if hits, the proxy returns the objects
  3. if misses, the proxy creates a new connection to the origin server
  4. the proxy stores a copy, and forwards it to the client



Web Caching (cont.)

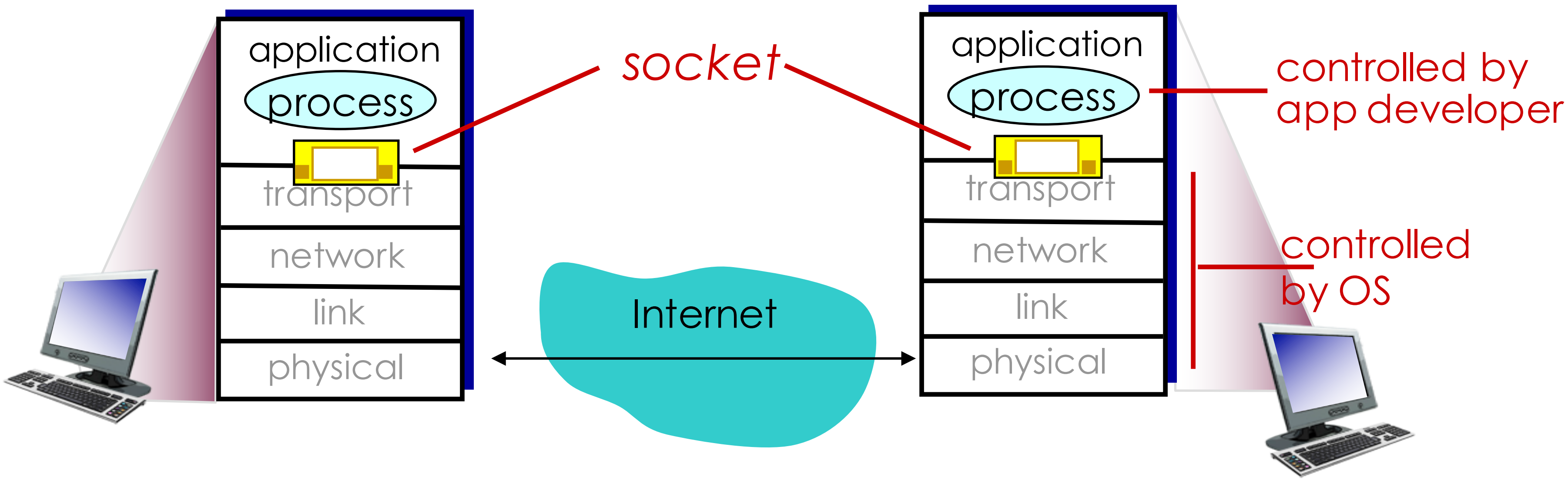
- Typically installed by an ISP
- Why needed?
  - closer, reducing the response time
  - sharing loading, reducing Web traffic
- **Q:** What is the disadvantage?
  - experience even longer latency if the proxy does not cache the objects
- **Q:** What is the practical challenge?
  - need to determine what objects should be kept if the storage is nearly full

Outline

- Principles of network applications
- Web and HTTP
- P2P Applications (later lecture)
- Video Streaming and CDN (later lecture)
- **Socket programing 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
- **Port Number**
  - open protocols (FTP, HTTP, …): follow RFC
  - Proprietary applications: avoid using well-known ports
- 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 number to each packet (typically down by OS)
  - receiver extracts sender IP address and port number 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 →

create UDP socket for server →

get user keyboard input →

Attach server name, port to message; send into socket →

read reply characters from socket into string →

print out received string and close socket →

```
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

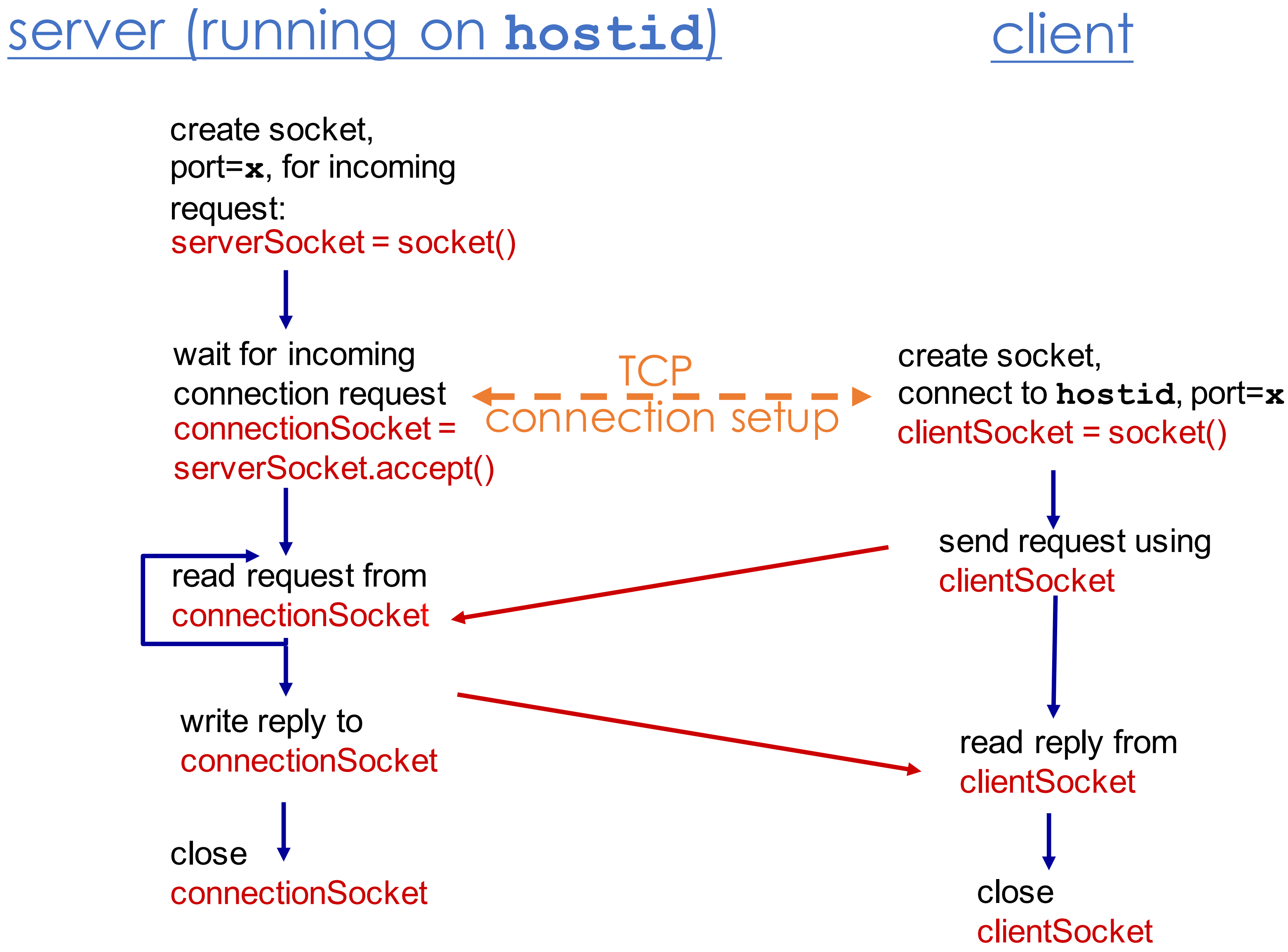
```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),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

Socket Programming with TCP

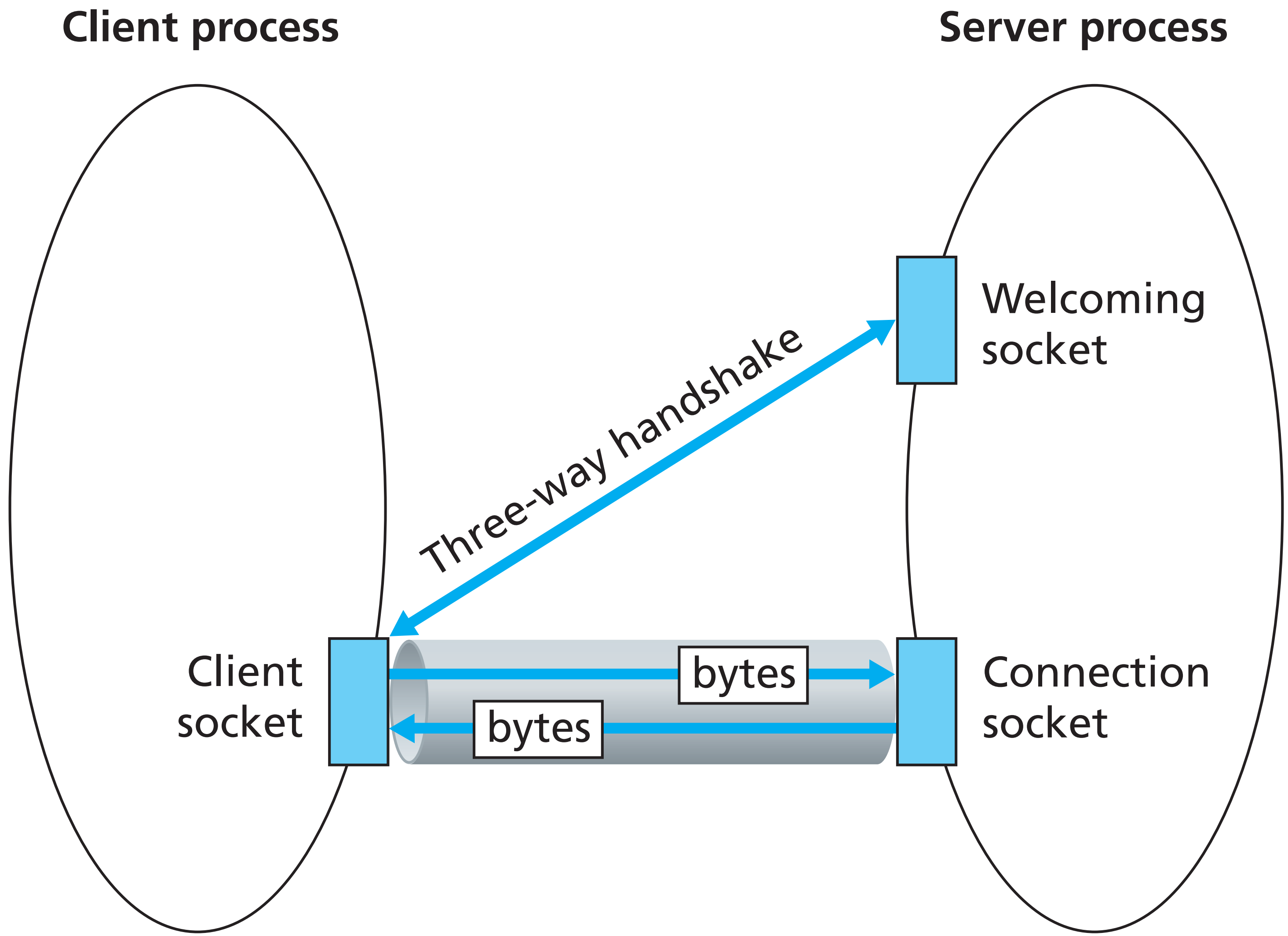
- **Client must contact server**
  - server process must first be running
  - server must have created socket (door) that welcomes client's contact
- Client contacts server by
  - creating TCP socket, **specifying IP address, port number** of server process
- When client creates socket, **client TCP 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 Ch. 3)

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

Client/Server Socket Interaction: TCP



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) // max # of clients, at least 1
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()
     // connectionSocket dedicated to a particular user
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                              encode())
    connectionSocket.close()
```

Summary

- Principles of network applications
- Web and HTTP
  - Will introduce HTTP streaming in later lectures
- Socket programing with UDP and TCP
  - Homework 1: Socket programming over UDP/TCP for audio delivery

Mini-Assignment

- Wireshark lab: HTTP
  - Trace log as connecting to: http://people.cs.nctu.edu.tw/~katelin/courses/mmnet17
  - Trace log as connecting to: https://www.youtube.com/watch?v=JqcmkLux0z8
- Instruction
  - https://wiki.wireshark.org/Hyper_Text_Transfer_Protocol
  - Tip of filtering: Find the IP and set "ip.dst == DST_ADDR"