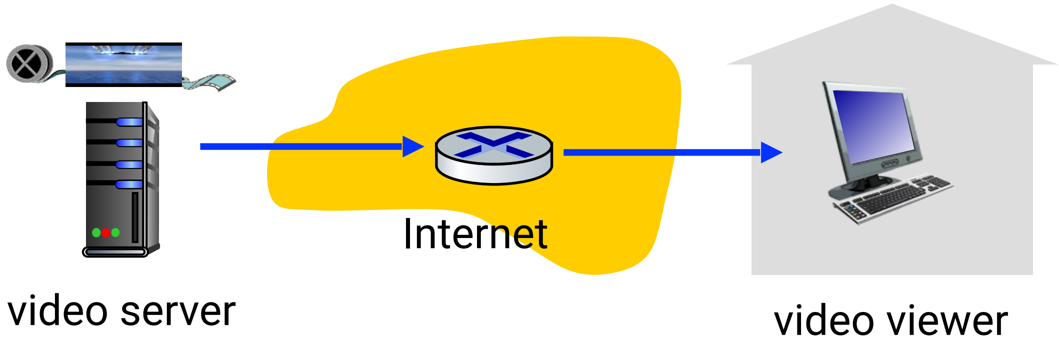
Content Distribution Networks

Challenges of Internet Streaming



Data Volume

Netflix, YouTube, Amazon Prime account for 80% of residential ISP traffic (2020)

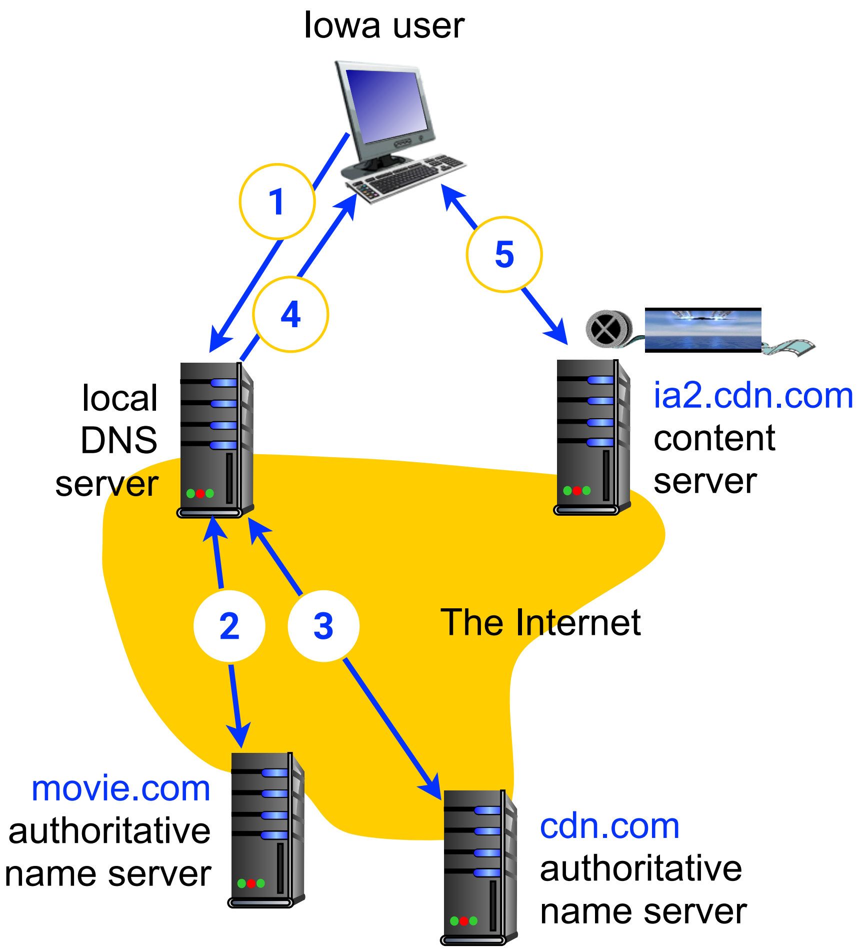
Scale

How to efficiently stream video to millions - billions of users worldwide?

Heterogeneity

Users have different device capabilities and bandwidths, both of which may vary with time

CDN operation



1. **DNS lookup**: User visits *movie.com*, and user's host sends a DNS query to the local DNS server
2. **CDN handover**: local DNS server relays the query to an authoritative name server for *movie.com*. Instead of an IP address, it returns the name of its CDN operator
3. **Content server selection**: local DNS server sends another query to *cdn.com*, to which the CDN responds with the IP address of the chosen/nearby content delivery server
4. **DNS response**: local DNS forwards the IP address of the content serving CDN node, *ia2.cdn.com*
5. **Content flow**: client establishes a *TCP* connection with the local content server, and sends *HTTP GET* request for the video

Real-world CDNs

	NETFLIX	YouTube
Operator	2007 - 2012: Akamai, Limelight, Level 3 2012 onwards: Self owned and operated	Google CDN (since 2006)
Architecture	Hybrid: ~200 IXP locations, ~100 ISPs. Offered free of cost to any ISP/IXP	Hybrid: several hundred IXP and ISP locations. Offered free of cost to any ISP/IXP
CDN content	Preprocessing on Amazon cloud; Netflix refreshes CDN content daily (i.e push caching)	Preprocessing on Google data centers; Employs DNS redirect and pull caching (lecture-7)
Streaming	DASH w/ manifest file	HTTP Streaming, choice of bit rate left to users

Ponder about: network **neutrality** | network **economics** | network **principles**

CS3640

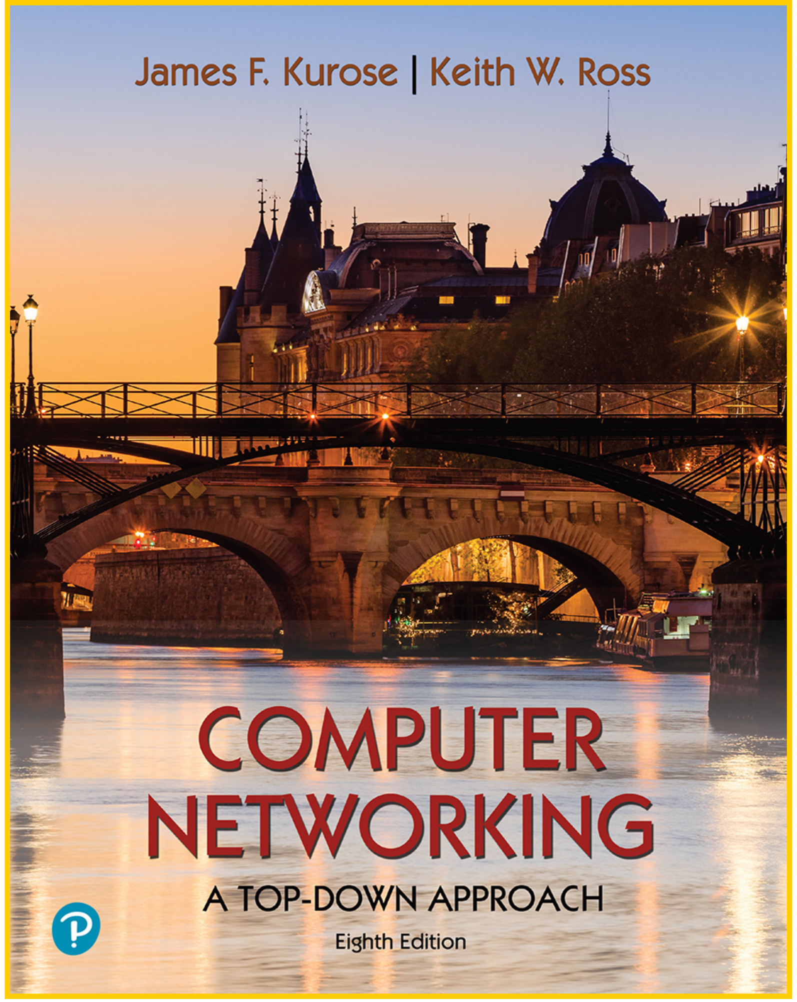
Application Layer (6): Socket Programming

Prof. Supreeth Shastri
Computer Science
The University of Iowa

Lecture goals

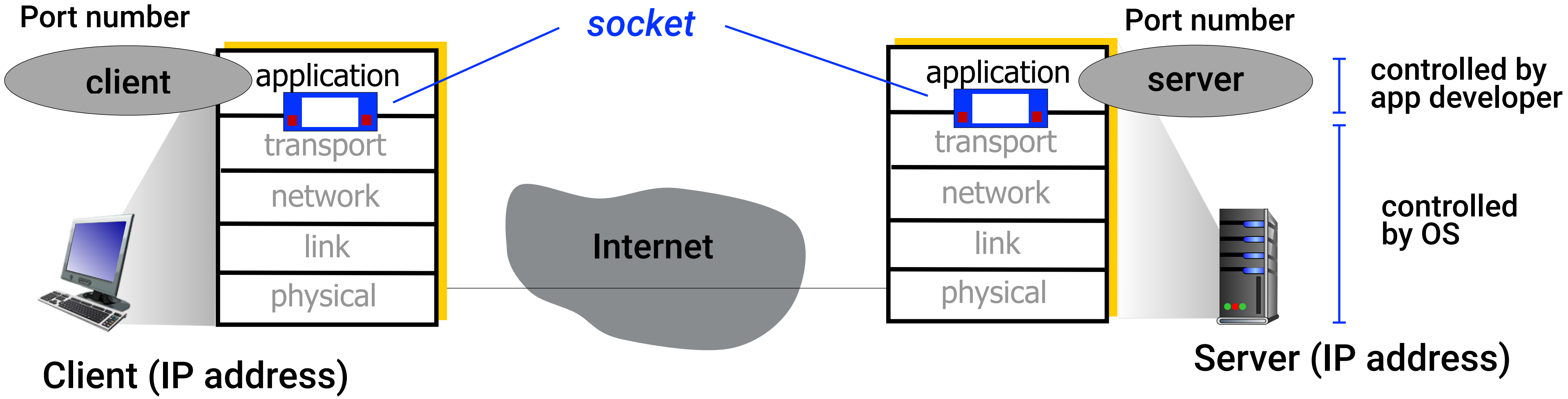
learn how to build client-server applications that communicate using sockets

- *Socket programming*
- *UDP sockets*
- *TCP sockets*



Chapter 2.7

Sockets: network programming API



Q: Do we have to use the socket APIs? Can't we create the whole packet and push it on to the Internet?

Socket fundamentals

Socket abstraction

- Originally defined in RFC 147 (in 1971) for ARPAnet
- First open implementation of sockets was by Berkeley in 1983
- After POSIX standardization, all operating systems have adapted Berkeley sockets as the de facto networking API

Socket types govern the exposed transport services

- **UDP sockets** for unreliable datagrams
- **TCP sockets** for reliable, flow- and congestion-controlled data streams
- **Raw sockets** for directly sending and receiving IP packets

Socket programming

Types of networking applications

- **Open**: conform to the rules laid out in the RFCs (or other standards). E.g., HTTP browsers and web servers that interoperate without being developed by same organization/developers
- **Proprietary**: applications whose behavior is not openly documented; may change over time without any notice. E.g., Skype and Zoom

A simple app for this lecture

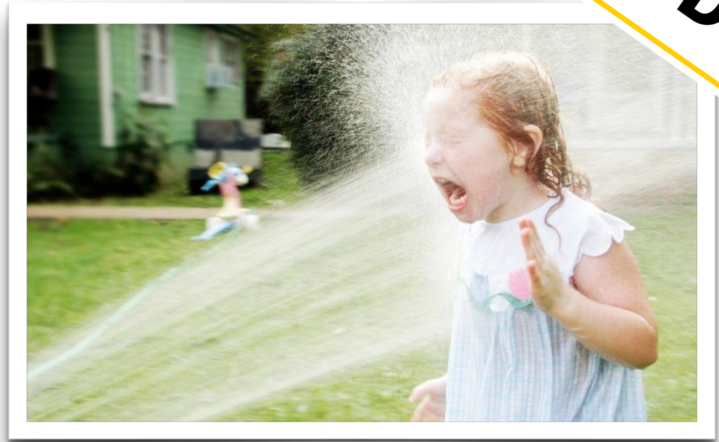
- client reads a line of characters (data) from its keyboard and sends data to server
- server receives the data and converts characters to UPPERCASE
- server sends modified data back to client
- client receives modified data and displays line on its screen

Q: does this make our application open or proprietary?

Hands-on Socket Programming

TCP

UDP



Mechanics of UDP

Application viewpoint

UDP provides unreliable transfer for a group of bytes (“datagrams”) between client and server processes

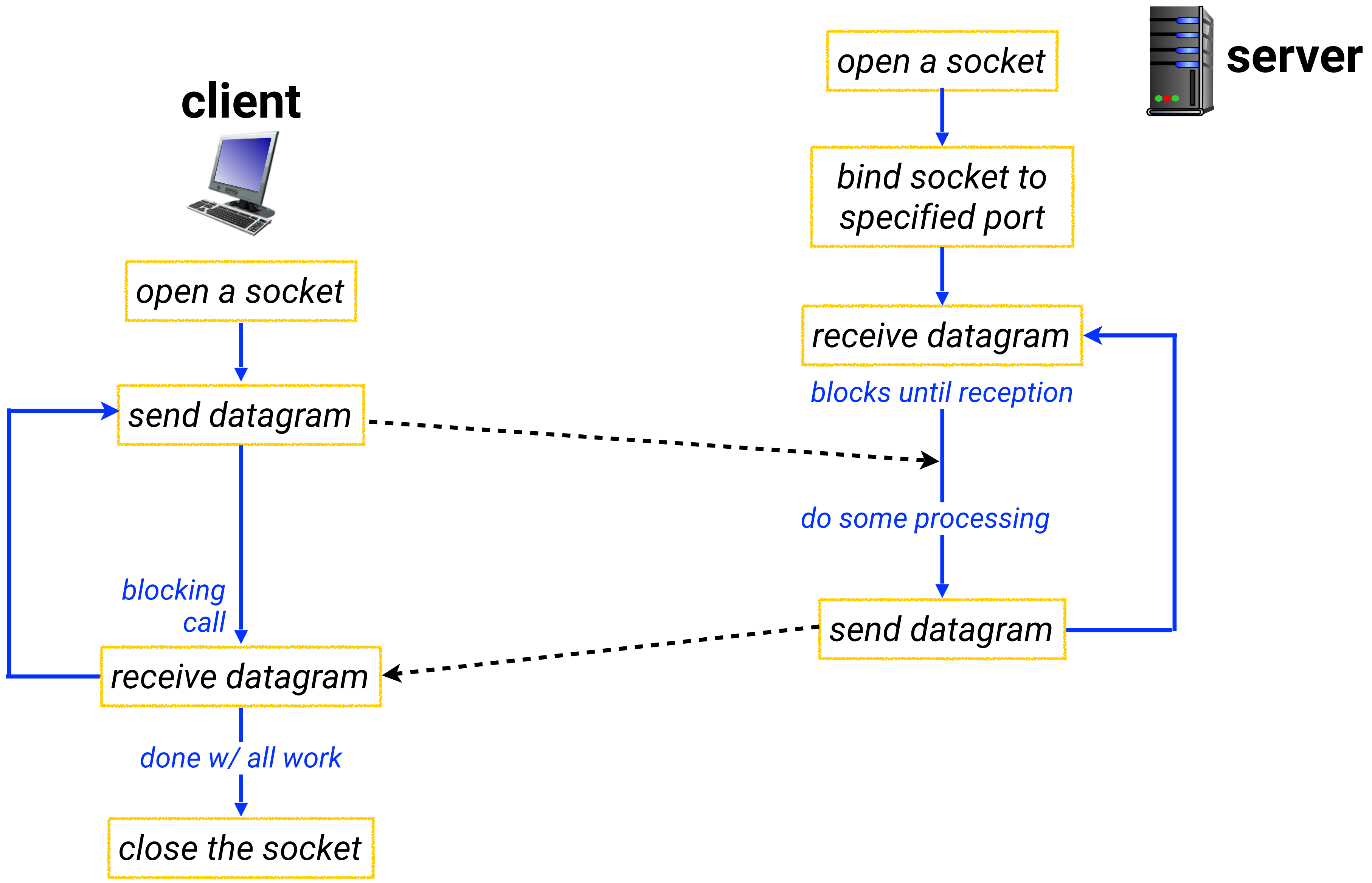
There is no “connection” between client and server

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

Minimal expectations

- transmitted data may be lost, may arrive out of order, and may overwhelm the receiver
- UDP does not monitor network congestion, nor does it have to respond to it

UDP Client Server Interaction



Coding up the UDP client

include Python's socket library →

create UDP socket for server →

get user keyboard input →

attach server name, port to message; send it 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()
```

Coding up the UDP server

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)
```

Demo: UDP client-server

Mechanics of TCP

Application viewpoint

TCP provides a reliable, in-order byte-stream (“pipe”) between client and server processes

First, server must be setup to accept connections

- server process must be continuously running
- server must have created the socket that welcomes client’s contact

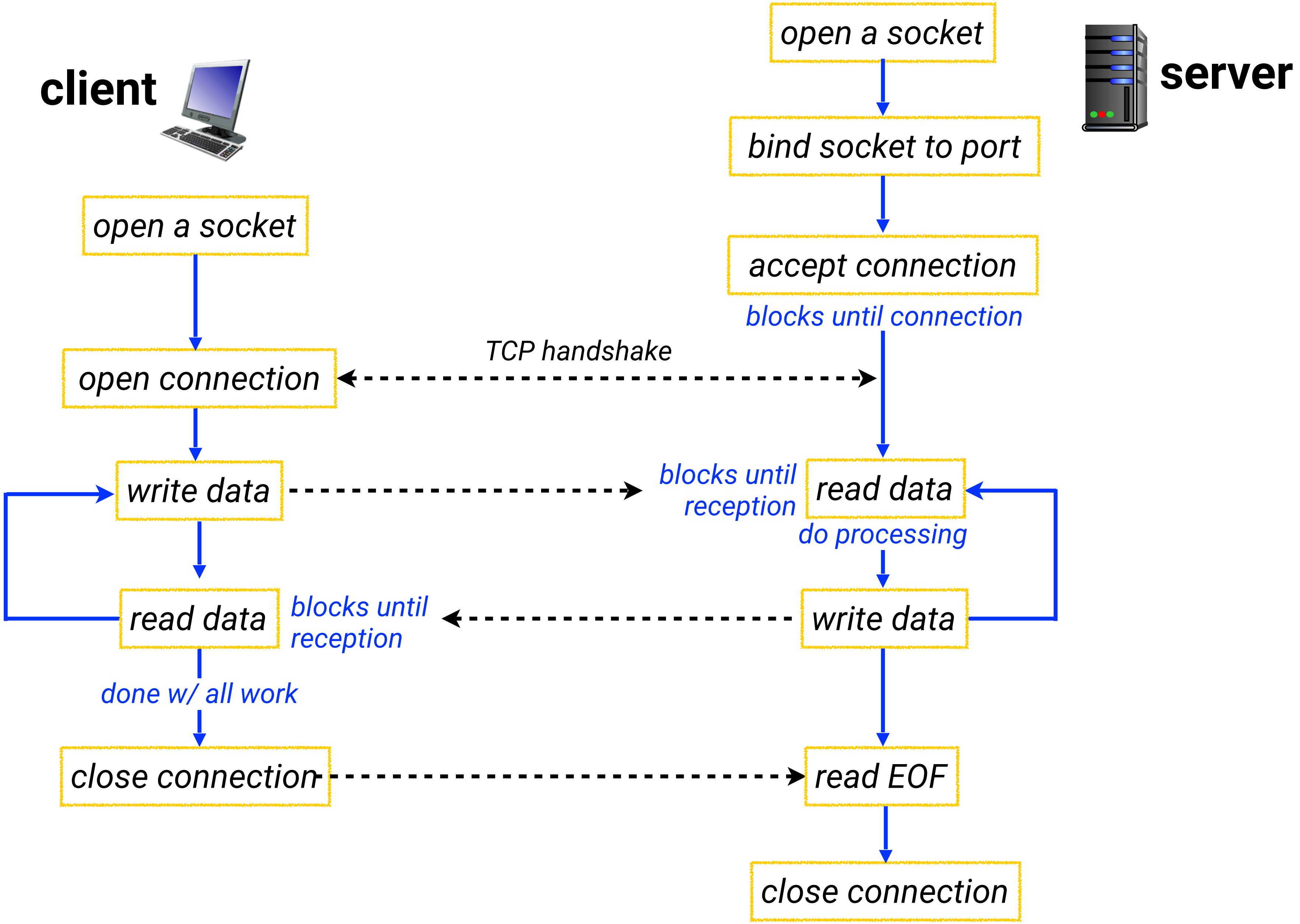
Then, a client can contact the server by

- creating a TCP socket and specifying server’s IP address and port number
- when client creates socket, client’s TCP establishes a connection to server TCP

On the server side

- when contacted by a client, server TCP spawns a new socket to manage communications with that client
- this feature allows the server to talk with multiple clients

TCP Client Server Interaction



Coding up the TCP client

create TCP socket for server

perform TCP handshake

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 = input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

Coding up the TCP server

create TCP welcoming socket

server begins listening for incoming TCP requests

server waits on incoming requests, new socket created on return

read and write bytes from/to socket (no IP addr/port as in UDP)

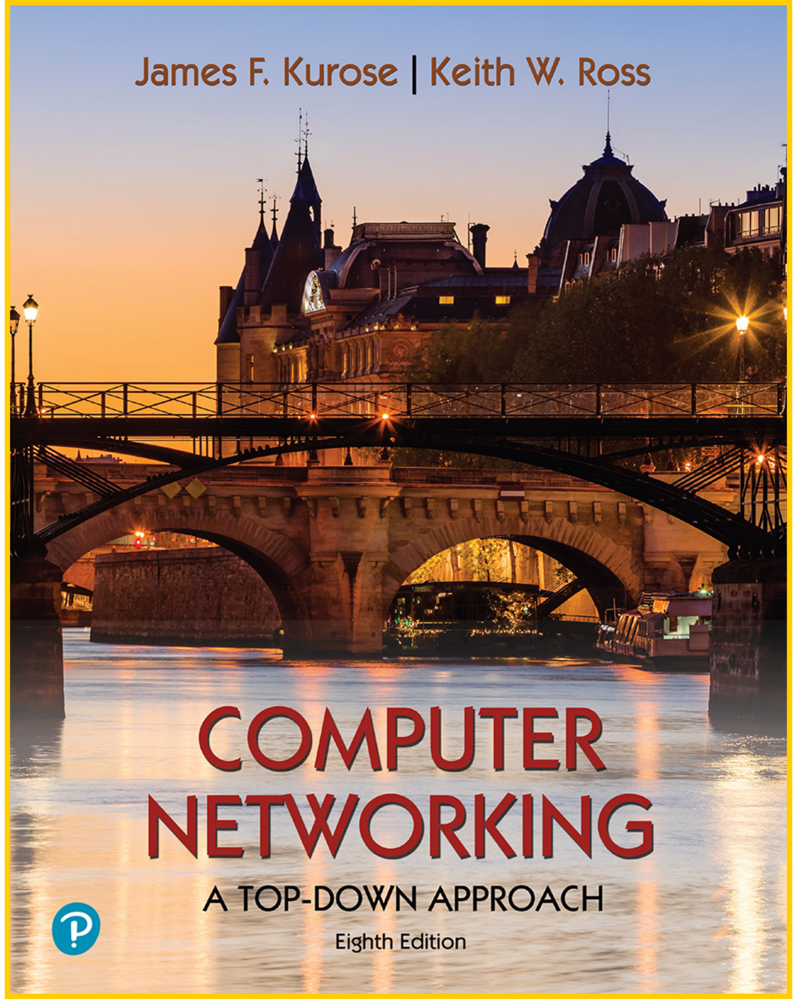
close connection to this client (but *not* welcoming socket)

```
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()
```

Next lecture

understand the principles and organization of the transport layer

- *Services*
- *Internet's transport protocols*
- *Multiplexing and demultiplexing*



Chapter 3.1 - 3.2

Spot Quiz (ICON)