Chapter 2
Application Layer

TA：YANG Xinyu
Email: xinyu.yang@connect.ust.hk

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks

2.7 socket programming with UDP and TCP

Some network apps

- 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
- …
- …

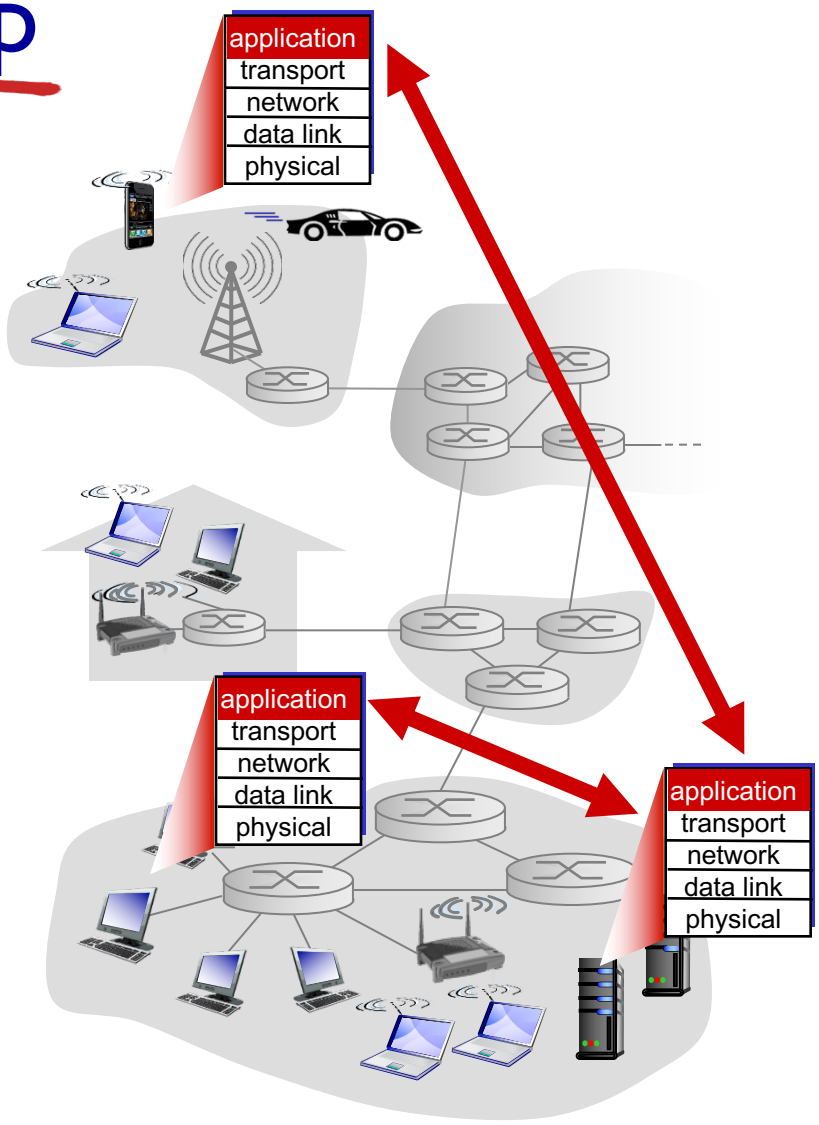
Creating a network app

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

Possible structure of applications

- Client-Server
- Peer-to-peer (P2P)

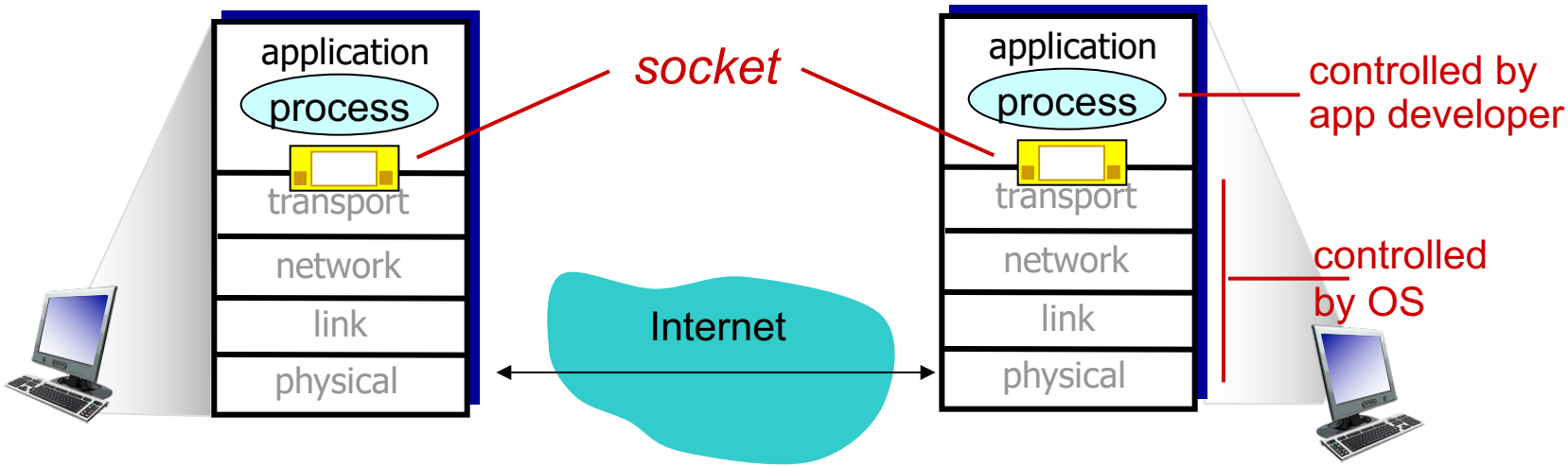
Software obtains communication services from the transport layer

- TCP service: reliable, in order delivery of a stream of bytes
- UDP service: unreliable datagram delivery

Service access point called: Socket

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out the door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



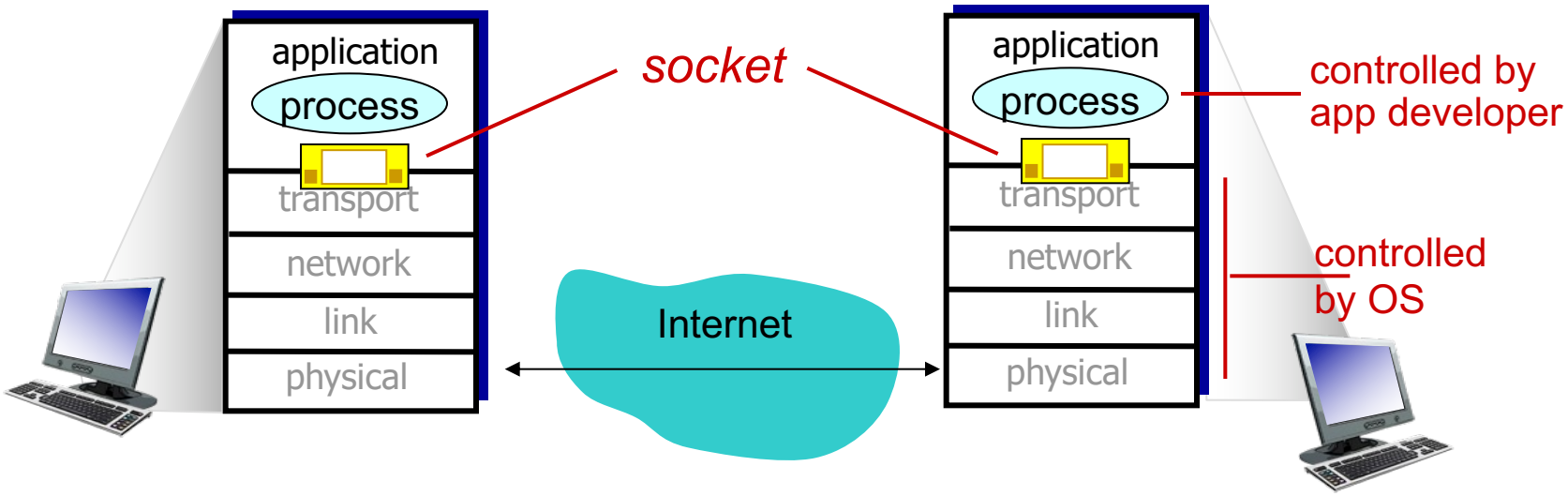
Sockets

- Socket is just an operating system object to allow Application software to talk to Transport layer:
- Associated with:
  - data structures and variables to identify and track the communication status through this socket
  - procedures that the data sent or received through this socket must go through before it is transmitted
  - identifiers to be able to dispatch received data to the appropriate application layer process
- For example, TCP must
  - keep track of the currently sent byte order
  - keep track of the currently acknowledged byte,
  - perform congestion control, flow control, check for errors, …

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

Socket programming *with UDP*

UDP: no "connection" between client & server

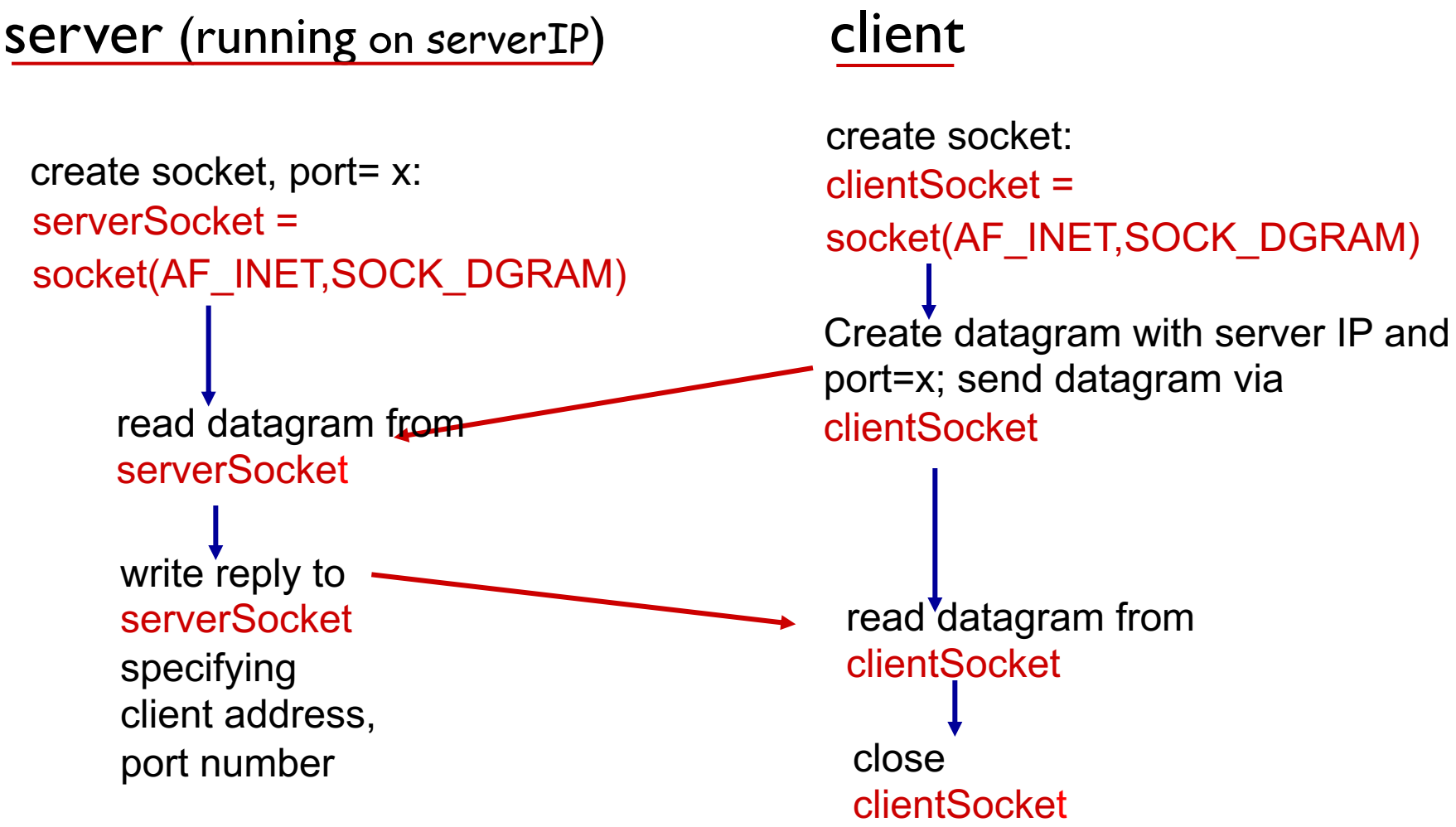
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

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



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

```
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)
```

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

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:

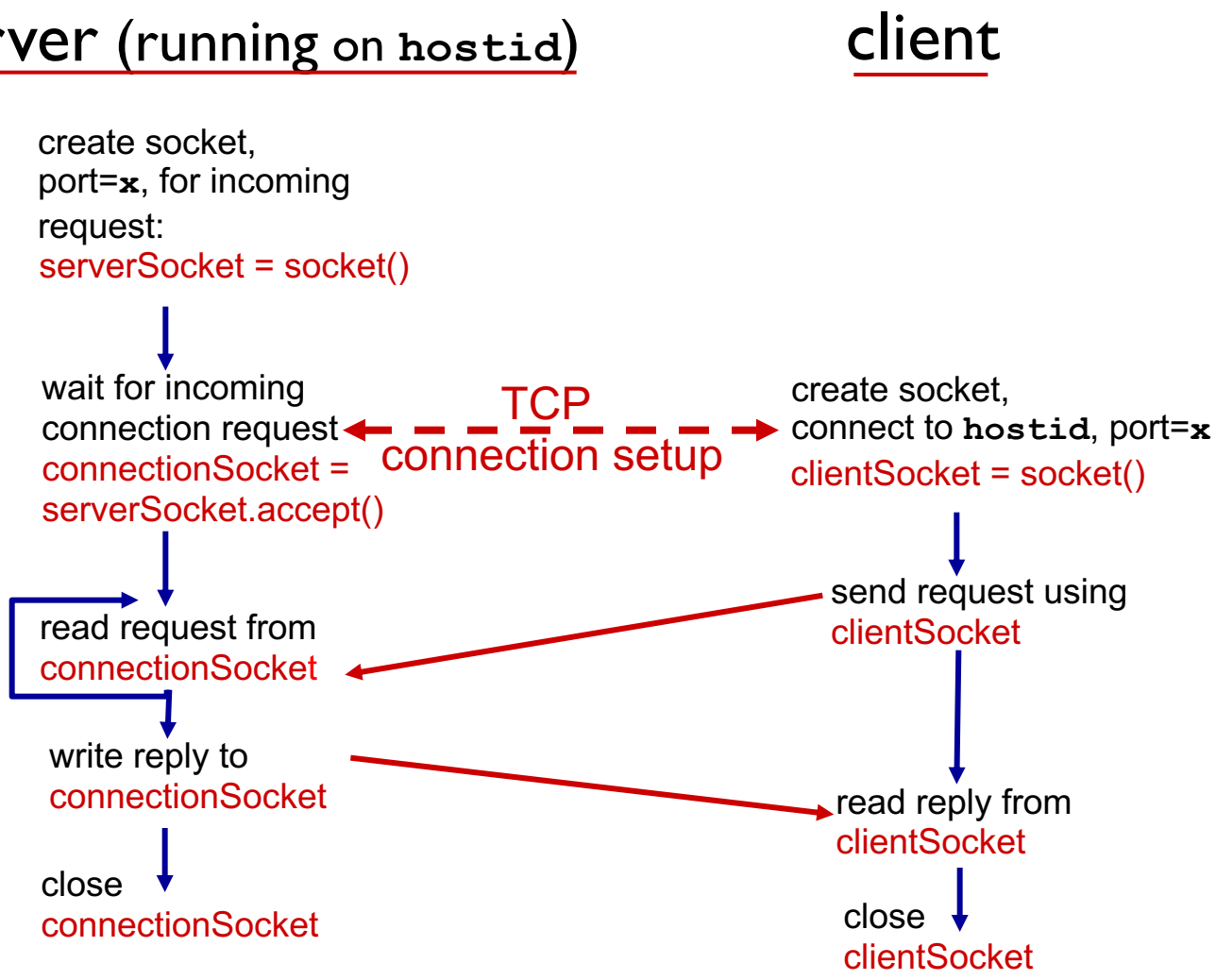
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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

```
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()
```

create TCP socket for server, remote port 12000

Different from UDP

No need to attach server name, port

Example app: TCP server

Python TCPServer

```
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()
```

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)

Chapter 2: summary

- In summary we have seen two types of sockets
  - Datagram sockets
  - Stream Sockets
- We have also learned how to create some basic clients and servers

Next:

- We will learn how this is done in C
- Then we will look at advanced techniques on how to build high performance client-server programs

Development Environment

Your C/Python code:

- *Must compile and run on lab2 machines accessible via ssh at csl2wkXX.cse.ust.hk, XX=01..52*
- *Don't run code overnight as the machines are reboot daily.*
- *Don't store large files on these machines.*

How to login:

- *Hostname: csl2wkXX.cse.ust.hk (XX=01..52)*
- *Port: 22*
- *Username: your itsc account (mine is xyangcp)*
- *Password: your itsc password*

For example: ssh xyangcp@csl2wk01.cse.ust.hk

Task

Play with the python code (available on Canvas) on CSLAB 2 machines and answer questions:

- Does it work if we replace "serverName" from "localhost" to "127.0.0.1" in tcp/udp client?
- What will happen if we change "serverPort" to a number less than "1024" like "22" in both tcp/udp client and server? Why?
- Change the messageSize to "1" and see what will happen? Why?
- Describe the what needs to be changed if we want to implement a web server that handles one request at a time in the TCP code.

Submit a pdf document on Canvas before 11:59pm tonight!