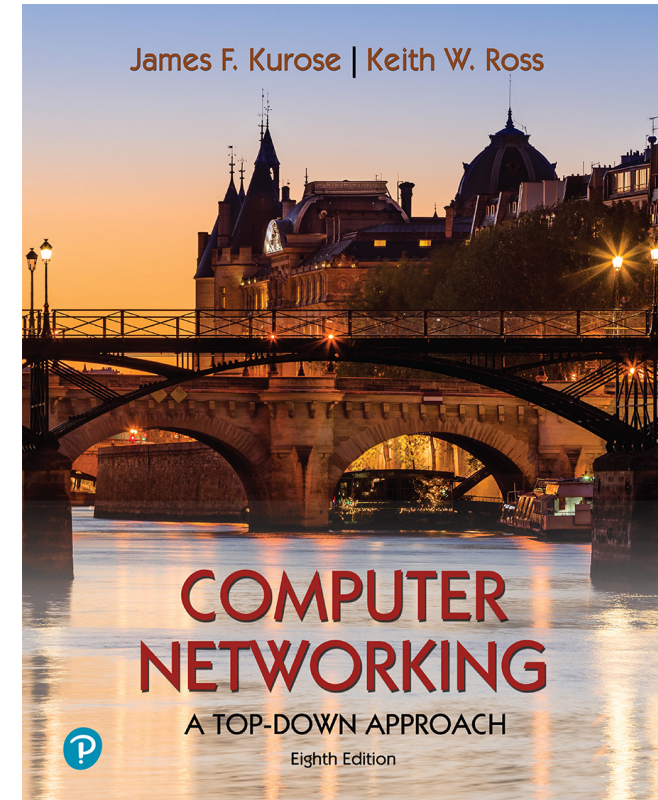


Chapter 2

Application Layer



Computer Networking: A Top-Down Approach

8th edition

Jim Kurose, Keith Ross
Pearson, 2020

Acknowledgement: Based on the textbook's website:
https://gaia.cs.umass.edu/kurose_ross/index.php

Application Layer: Overview

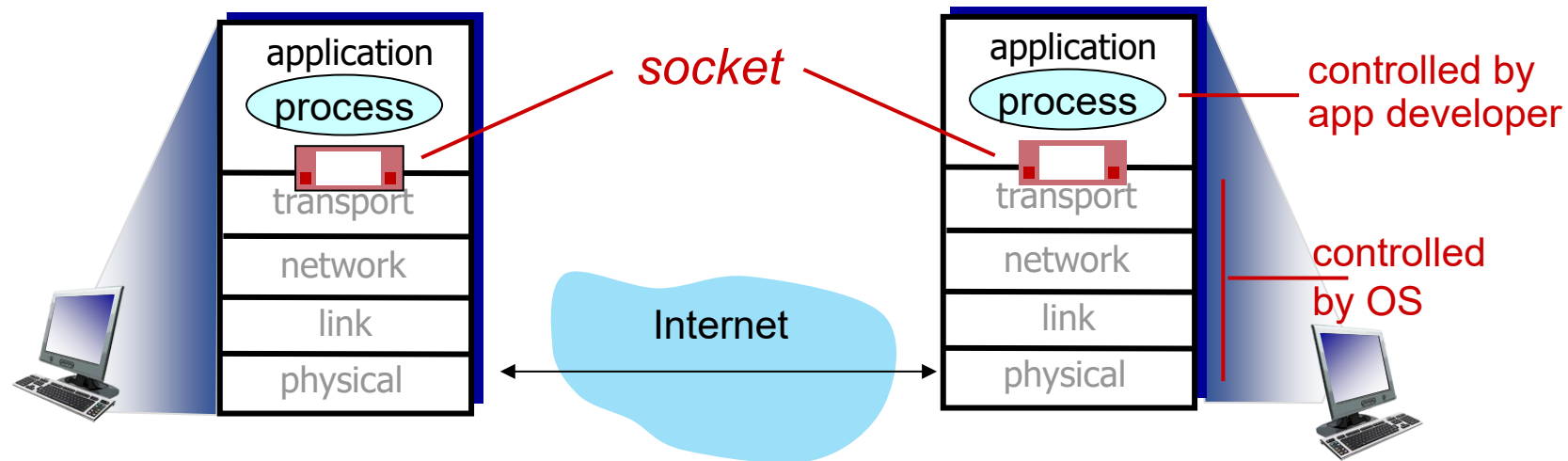
- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- **socket programming with UDP and TCP**



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

Socket programming with UDP

UDP: no “connection” between client and 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 processes

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 serverIP address
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 → clientSocket = socket(AF_INET,
                                           SOCK_DGRAM)
get user keyboard input → message = input('Input lowercase sentence:')
attach server name, port to message; send into socket → clientSocket.sendto(message.encode(),
                                                                    (serverName, serverPort))
read reply data (bytes) from socket → 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

create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)
bind socket to local port number 12000 → serverSocket.bind(('', serverPort))
print('The server is ready to receive')

loop forever → while True:
    Read from UDP socket into message, getting client's address (client IP and port) → message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    send upper case string back to this client → 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
 - client source port # and IP address used to distinguish clients (more in Chap 3)

Application viewpoint

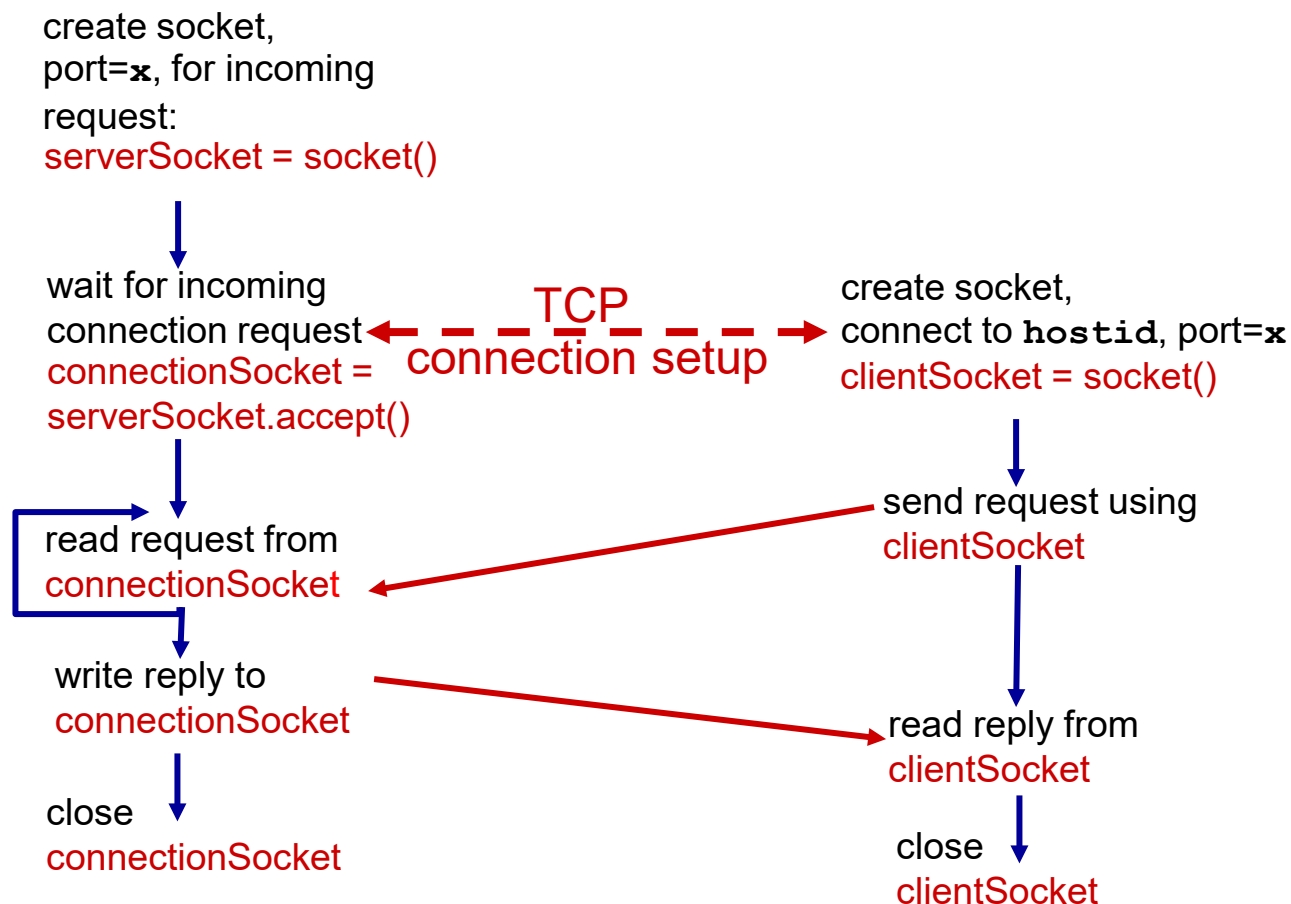
TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server processes

Client/server socket interaction: TCP



server (running on `hostid`)

client



Example app: TCP client

Python TCPClient

create TCP socket for server,
remote port 12000

```
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()
```

No need to attach server name, port

Example app: TCP server

Python TCPServer

	from socket import *
	serverPort = 12000
create TCP welcoming socket →	serverSocket = socket(AF_INET,SOCK_STREAM)
	serverSocket.bind(('',serverPort))
server begins listening for incoming TCP requests →	serverSocket.listen(1)
	print('The server is ready to receive')
loop forever →	while True:
server waits on accept() for incoming requests, new socket created on return →	connectionSocket , addr = serverSocket.accept()
read bytes from socket (but not address as in UDP) →	sentence = connectionSocket.recv(1024).decode()
	capitalizedSentence = sentence.upper()
	connectionSocket.send(capitalizedSentence.encode())
close connection to this client (but <i>not</i> welcoming socket) →	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”

Additional Chapter 2 slides

JFK note: the timeout slides are important IMHO if one is doing a programming assignment (especially an RDT programming assignment in Chapter 3), since students will need to use timers in their code, and the TRY/EXCEPT is really the easiest way to do this. I introduce this here in Chapter 2 with the socket programming assignment since it teaches something (how to handle exceptions/timeouts), and lets students learn/practice that before doing the RDT programming assignment, which is harder

Python try-except block

Execute a block of code, and handle “exceptions” that may occur when executing that block of code

try:

<do something>

{ Executing this **try code block** may cause exception(s) to catch. If an exception is raised, execution jumps from jumps directly into **except code block**

except <exception>:

<handle the exception>

{ this **except code block** is only executed *if an <exception> occurred* in the **try code block** (note: except block is *required* with a try block)