Chapter 2
Application Layer

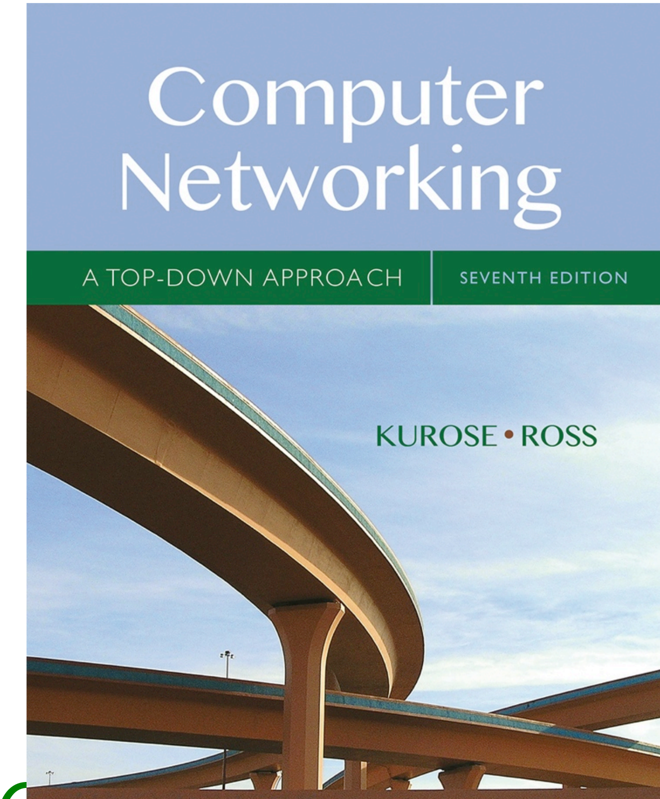
These slides have been adapted from the Pearson slides. S. Thomson 09/2017.

A note on the use of these Powerpoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR



Computer Networking: A Top Down Approach

7th edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016

Chapter 2: outline

The image part with relationship ID rId3 was not found in the file.

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

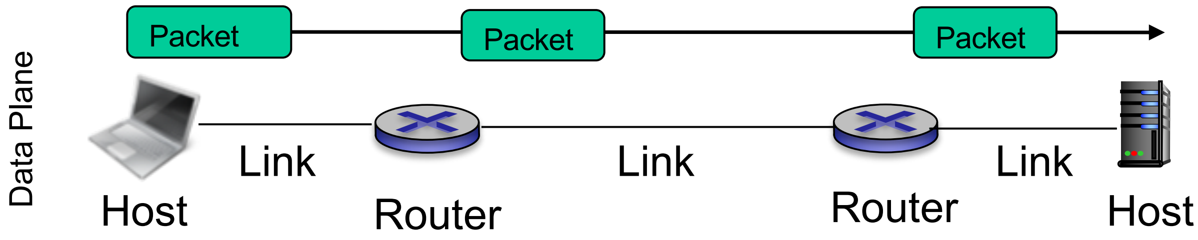
Content

- What transport services does the network provide to applications?
- How does the application developer make use of these services?
- How does the application developer decide what service to use?

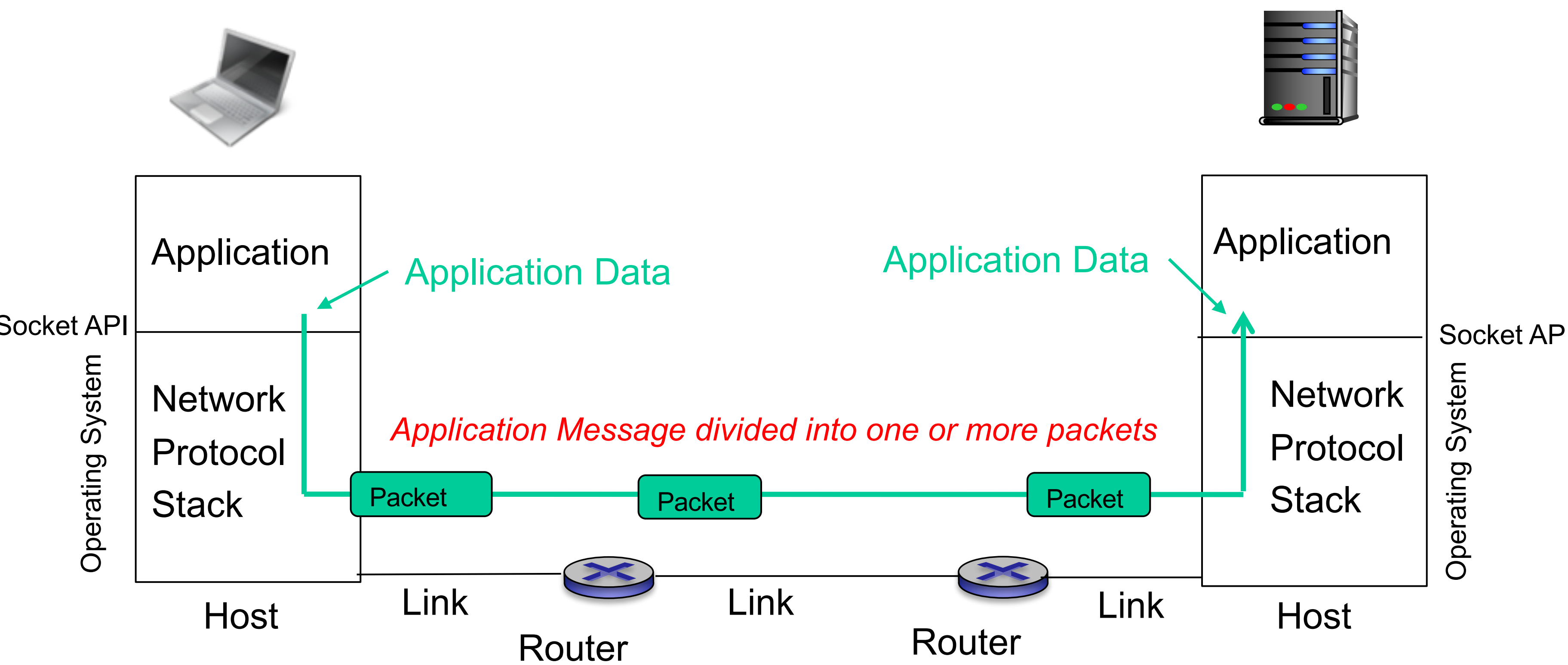
RECAP: What is the Internet Architecture? Packet Switching

- Components:
  - Hosts: Transmit and receive application packets of length L
  - Routers: Store and forward packets from one link to another
  - Links: Propagate packets within the link

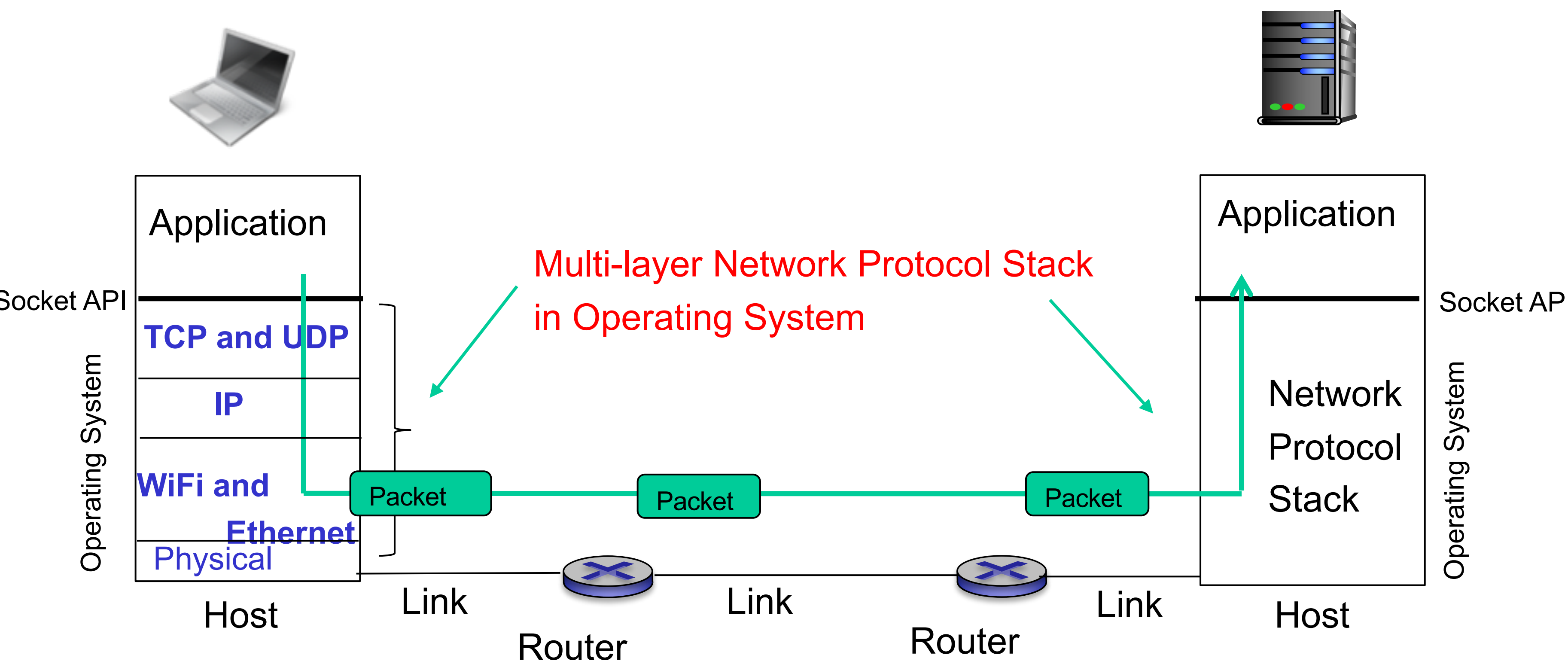
- Note:
  - A router may also act as a host e.g. management



RECAP: What is the Internet Architecture? Packet Switching

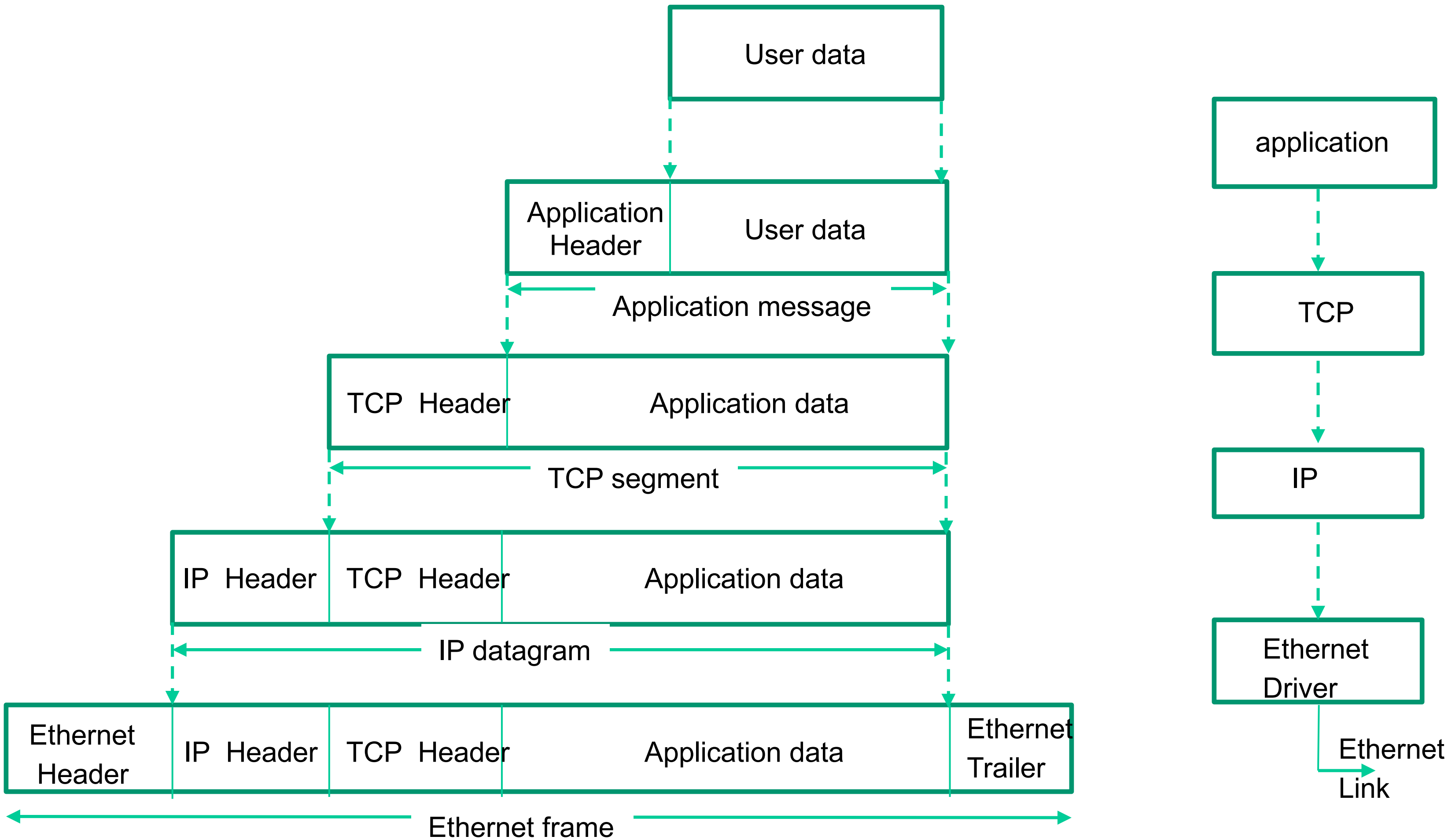


RECAP: What is the Internet Architecture? Protocol Stack



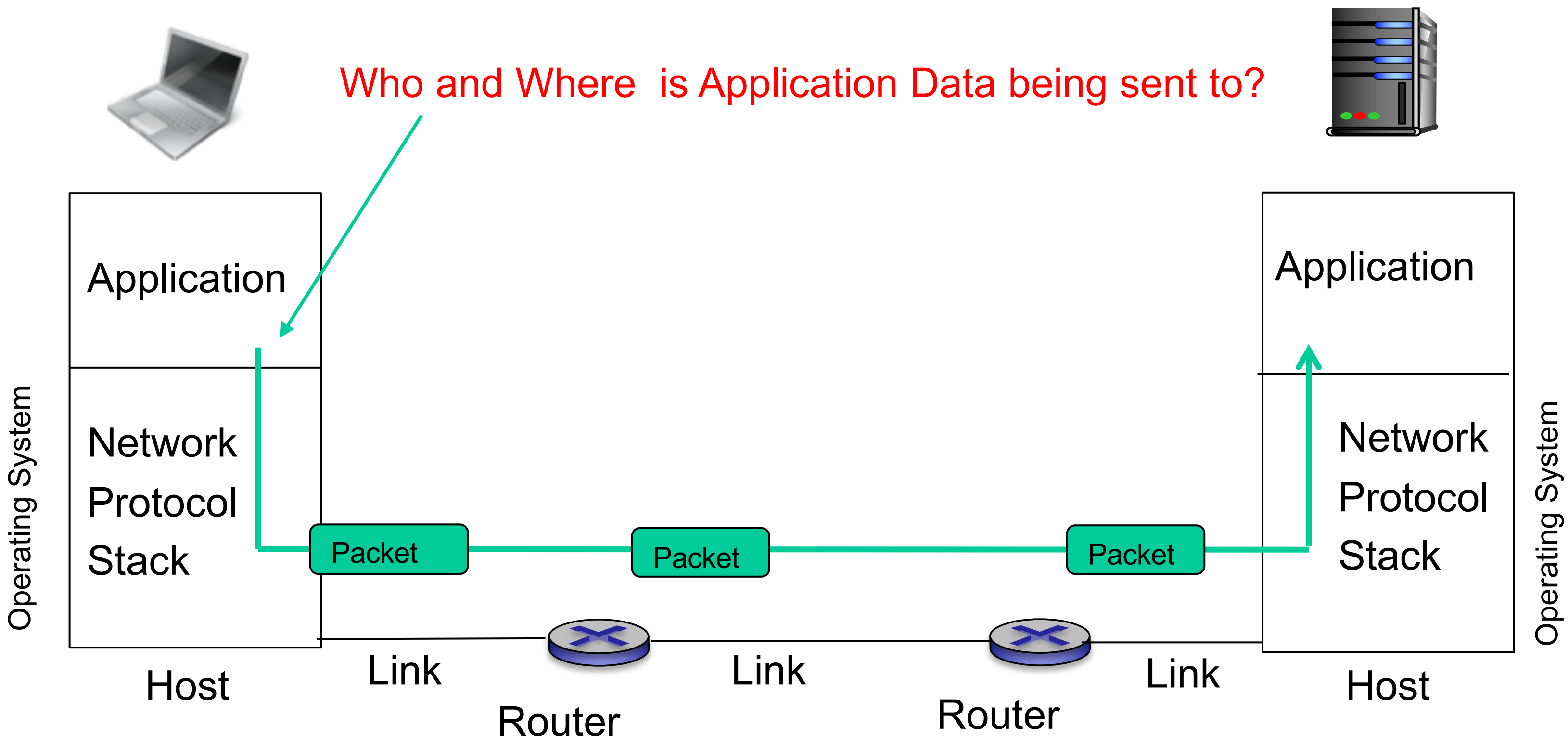
RECAP: What is the Internet Architecture?

Protocol Stack: Encapsulation



Reference: WR Stevens TCP Illustrated Volume 1, Addison-Wesley

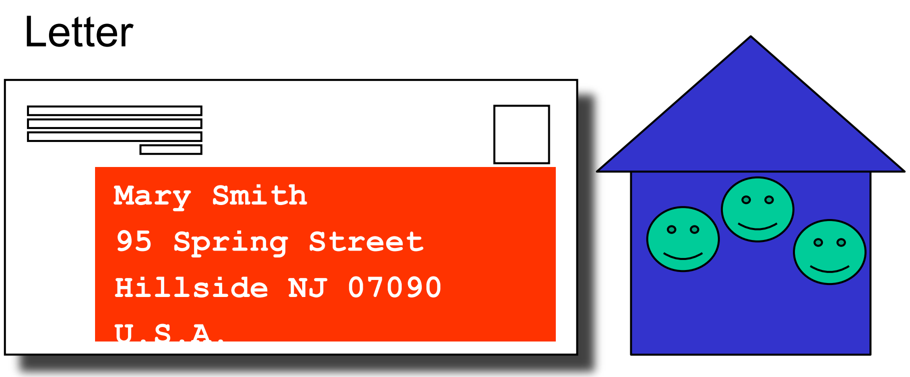
RECAP: What is the Internet Architecture? Packet Switching



Note: Uni-directional View

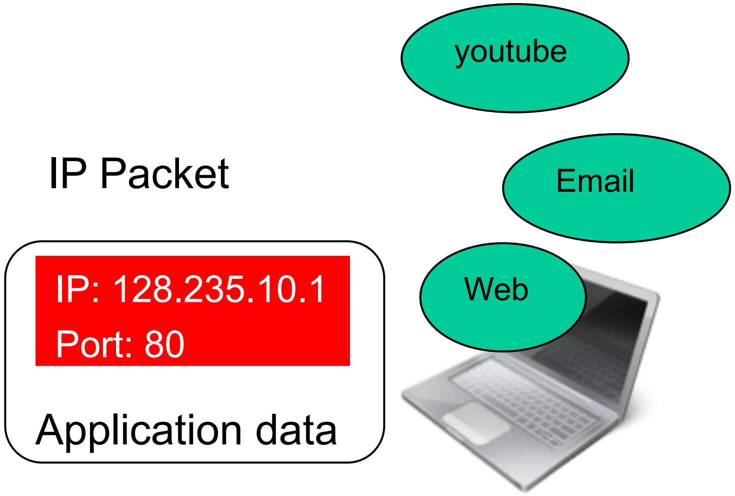
RECAP: What is the Internet Architecture? Packet Switching

- Naming and Addressing the Destination App (Process)
  - IP address indicates destination host
  - Port indicates application on host
- Use Post Office Analogy



Postal Address identifies and locates house

Name identifies person in house



IP Address identifies and locates host

Port identifies application in host

Host Addressing

- Hosts are addressed via IP addresses (32 bits)
  - Example: www.njit.edu = 128.235.251.25
  - IP addresses assigned by Regional Internet Registries (RIR)
  - To communicate with a destination, the application must determine the IP address of the host it is communicating with
    - May know a priori
    - May look up hostname->IP address mapping in DNS
    - May learn from incoming message (responses only)

Process Addressing

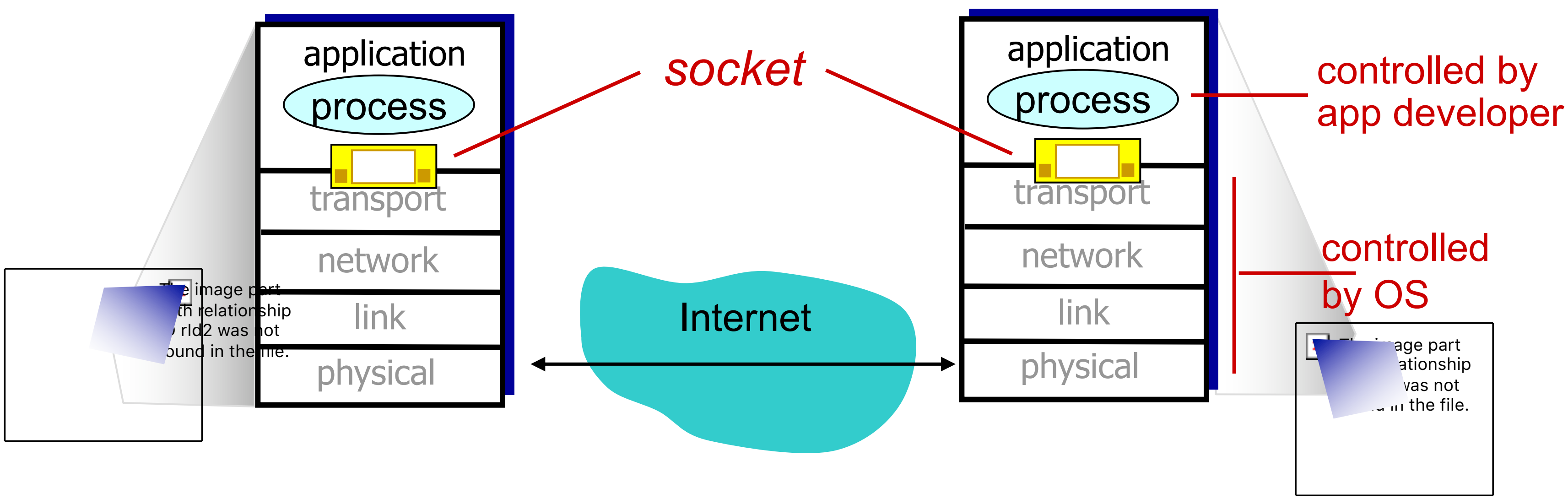
- Ports (16 bits)
  - Examples: 80 (HTTP), 53 (DNS)
  - Port numbers 0-1023 reserved for "well-known" services
  - Proprietary/Private apps: 1024 – 65535
  - Reserved Port Numbers assigned by IANA (Internet Assigned Numbers Authority) www.iana.org
  - Operating system can allocate unique port number to application dynamically, or application developer can specify port number to use

Socket API

The image part with relationship ID rId2 was not found in the file.

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Socket API

- Application developers need to send messages across the network

- The Socket API provides this:
  - Access to network controlled in OS
  - Other functions provided in system library

Socket API

- Design Goals
  - Communication between processes should not depend on whether they are on same or different machine

Socket API

The image part with relationship ID rId2 was not found in the file.

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example:

1. client reads arguments from command line and sends data to server
2. server receives the data and prints
3. server echoes received data back to client
4. client receives data and prints

Client/server socket interaction:

The image part with relationship ID rId2 was not found in the file.

server	client
	Send request to server
Receive request from client	
Process Request	
Send response to client	
	Receive response from server
	Process Response

time

Socket programming *with UDP*

The image part with relationship ID rId2 was not found in the file.

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

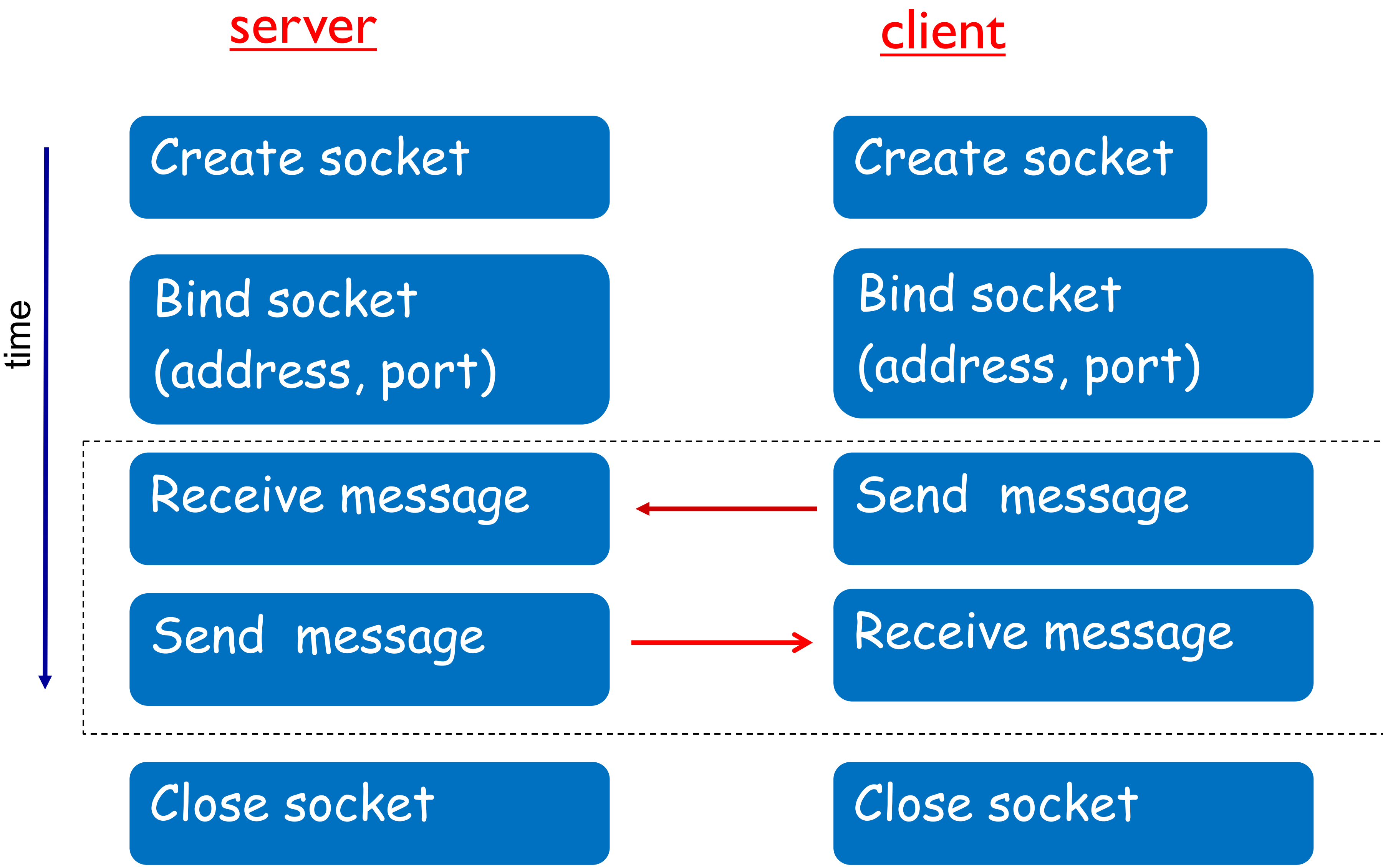
UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction (UDP):

The image part with relationship ID rId2 was not found in the file.



Step 1

Create a socket

int s = socket (int family, int type, int protocol)

AF_INET
AF_INET6

SOCK_DGRAM (UDP)
SOCK_STREAM (TCP)
SOCK_RAW (IP)

Default = 0

Python:

```
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
```

Step 2

Bind a socket

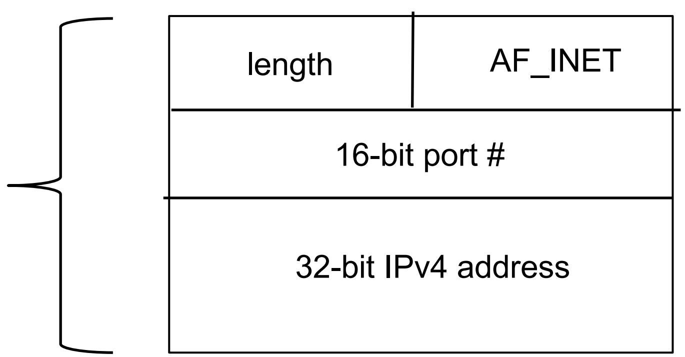
int error = bind (s, addr, addrlen)

socket # from *socket* call

Transport address (IP address, port)

Length of address

sockaddr_in



Python: serverSocket.bind(('', serverPort))

Step 3

- For TCP only, see later in slide deck
- Includes TCP connection establishment

Step 4 (exchange data)

```
Python:
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
receivedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
```

Connectionless calls (UDP):

```
int sendto(s, void *msg, int len, u_int flags,
     struct sockaddr *toaddr, addrlen)
int recvfrom (s, void *buf, int len, u_int flags,
     struct sockaddr *fromaddr, addrlen)
```

Step 5

Close socket

close(s)

Listening or connected socket #

Client/server socket interaction: UDP

The image part with relationship ID rId2 was not found in the file.

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

Attach server ip, port to message; send into socket

read reply characters from socket into string

print out received string and close socket

```
import sys, time
from socket import *

# Get the server hostname, port and data length as command line arguments
argv = sys.argv
host = argv[1]
port = argv[2]
count = argv[3]

# Command line argument is a string, change the port and count into integer
port = int(port)
count = int(count)
data = 'X' * count # Initialize data to be sent

# Create UDP client socket. Note the use of SOCK_DGRAM
clientsocket = socket(AF_INET, SOCK_DGRAM)

# Sending data to server
print("Sending data to   " + host + ", " + str(port) + ": " + data)
clientsocket.sendto(data.encode(),(host, port))

# Receive the server response
dataEcho, address = clientsocket.recvfrom(count)

# Display the server response as an output
print("Receive data from " + address[0] + ", " + str(address[1]) + ": " + dataEcho.decode())

#Close the client socket
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 string back to this client

```
import sys
from socket import *

serverIP = ''          # any local IP address
serverPort = 12000
dataLen = 1000000

# Create a UDP socket. Notice the use of SOCK_DGRAM for UDP packets
serverSocket = socket(AF_INET, SOCK_DGRAM)
# Assign IP address and port number to socket
serverSocket.bind((serverIP, serverPort))

print('The server is ready to receive on port:  ' + str(serverPort))

# loop forever listening for incoming datagram messages
while True:

    # Receive and print the client data from "data" socket
    data, address = serverSocket.recvfrom(dataLen)
    print("Receive data from client " + address[0] + ", " + str(address[1]) + ": " + data.decode())

    # Echo back to client
    print("Sending data to   client " + address[0] + ", " + str(address[1]) + ": " + data.decode())
    serverSocket.sendto(data,address)
```

Socket programming *with TCP*

The image part with relationship ID rId2 was not found in the file.

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

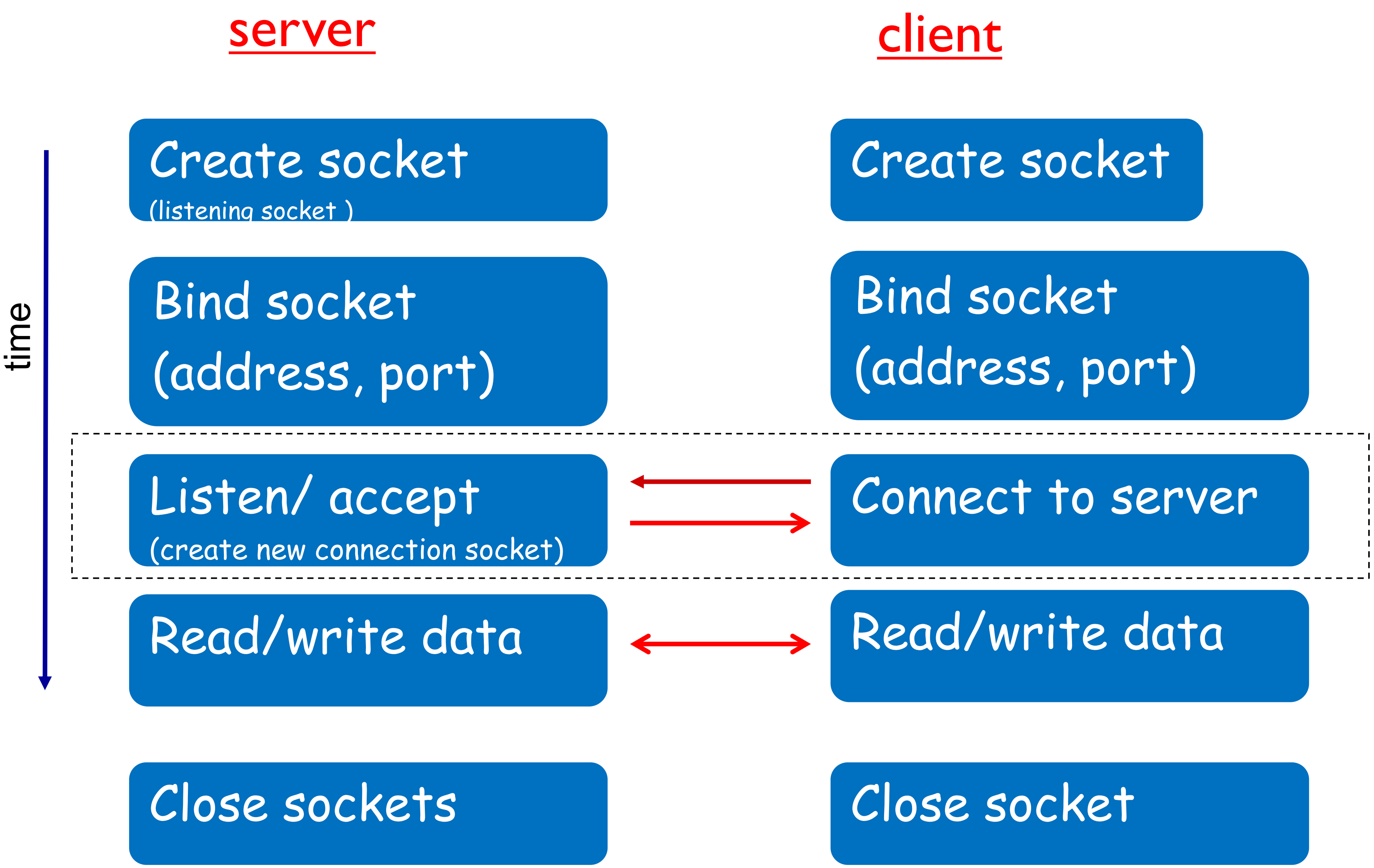
- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

application viewpoint:

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

Client/server socket interaction (TCP):

The image part with relationship ID rId2 was not found in the file.



Step 1

Create a socket

int s = socket (int family, int type, int protocol)

AF_INET
AF_INET6

SOCK_DGRAM (UDP)
SOCK_STREAM (TCP)
SOCK_RAW (IP)

Default = 0

Python:

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

Step 2

Bind a socket

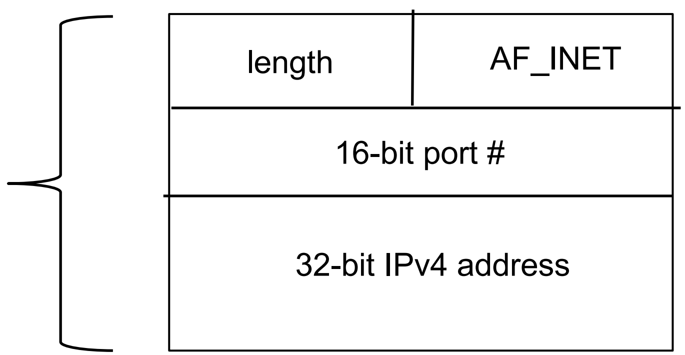
int error = bind (s, addr, addrlen)

socket # from *socket* call

Transport address (IP address, port)

Length of address

sockaddr_in



Python: serverSocket.bind(('', serverPort))

Step 3a (TCP server)

Listen on a socket

int error = listen (s, backlog)

socket # from
socket call

Queue Length
(number of connections
allowed between accept
system call)

Socket now configured to listen to new connections. Data flows on another socket.

Step 3b (TCP server)

Wait for new connection request from client

```
int sdata = accept (s, clntaddr, addrlen)
```

New socket #
(connected socket)

socket # from
socket call
(listening socket)

Transport address
of client

Transport address
length

New socket created for data flow.

Step 3 (TCP client)

Connect to server

int error = connect (s, svraddr, addrlen)

New socket #

socket # from socket call

Transport address of server

Transport address length

Client connection request to server

Step 4 (exchange data)

Connection-oriented calls (TCP):

read, write system calls

int send(s, void *msg, int len, u_int flags)

int recv(s, void *buf, int len, u_int flags)

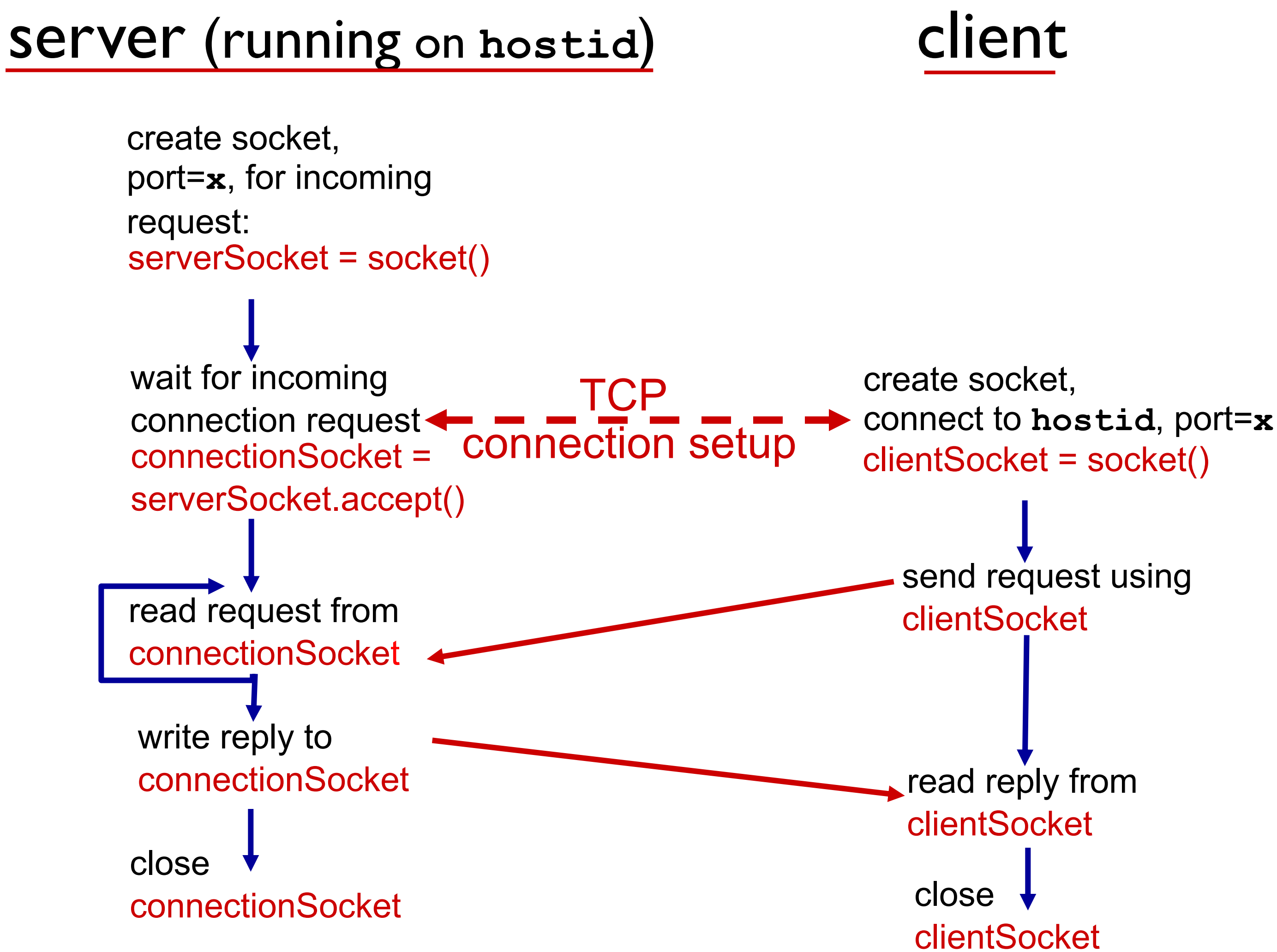
Connectionless calls (UDP):

int sendto(s, void *msg, int len, u_int flags, struct sockaddr *toaddr, addrlen)

int recvfrom (s, void *buf, int len, u_int flags, struct sockaddr *fromaddr, addrlen)

Client/server socket interaction: TCP

The image part with relationship ID rId2 was not found in the file.



Example app: TCP client

Python TCPClient

create TCP socket and connect to server ip, port

No need to attach server ip, port

```
import sys
from socket import *

# Get the server hostname, port and data length as command line arguments
argv = sys.argv
host = argv[1]
port = argv[2]
count = argv[3]

# Command line argument is a string, change the port and data length into integer
port = int(port)
count = int(count)

# Initialize and print data to be sent
data = 'X' * count

# Create TCP client socket. Note the use of SOCK_STREAM for TCP packet
clientSocket = socket(AF_INET, SOCK_STREAM)

# Create TCP connection to server
print("Connecting to " + host + ", " + str(port))
clientSocket.connect((host, port))

# Send data through TCP connection
print("Sending data to server:   " + data)
clientSocket.send(data.encode())

# Receive the server response
dataEcho = clientSocket.recv(count)
# Display the server response as an output
print("Receive data from server: " + dataEcho.decode())

# Close the client socket
clientSocket.close()
```

Example app: TCP server

Python TCPServer

```
from socket import *

serverIP = ''          # any local IP address
serverPort = 12000
dataLen = 1000000

# Create a TCP "welcoming" socket. Notice the use of SOCK_STREAM for TCP packets
serverSocket = socket(AF_INET, SOCK_STREAM)
# Assign IP address and port number to socket
serverSocket.bind((serverIP, serverPort))
# Listen for incoming connection requests
serverSocket.listen(1)

print('The server is ready to receive on port:  ' + str(serverPort))

# loop forever listening for incoming connection requests on "welcoming" soecket
while True:
    # Accept incoming connection requests, and allocate a new socket for data communication
    connectionSocket, address = serverSocket.accept()
    print("Socket created for client " + address[0] + ", " + str(address[1]))

    # Receive and print the client data in bytes from "data" socket
    data = connectionSocket.recv(dataLen).decode()
    print("Data from client: " + data)

    # Echo back to client
    connectionSocket.send(data.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)

POSIX system calls
Summary

System Call	Description
socket	Create a socket
bind	Associate address, port with socket (sockaddr)
listen	Set socket to listen to incoming connections
accept	Accepts next client connection
connect	Connect to socket on server
Read/write, sendto, recvfrom Sendmsg, recvmsg	Exchange data
Close/shutdown	Close connection

server

client

Connection-oriented only

Byte Ordering

- An issue when sending multi-byte values
- Not all computers store bytes in multi-byte values in the same order
  - Little endian: low-order byte in address A, high-order byte in address (A+1)
  - Big endian: high-order byte in address A, low-order byte in address (A+1)
- Example: 4-byte integer 0x00000001 (value 1)

A	A+1	A+2	A+3
01	00	00	00

Little endian

A	A+1	A+2	A+3
00	00	00	01

Big endian

Byte ordering: C

- Network protocols use big-endian ordering (*)
- Conversion utilities available for programmer convenience

uint_32 = htonl (uint32_t *hostint32)*
uint_16 = htons (uint16_t *hostint16)*
uint_32 = ntohl (uint32_t *netint32)*
uint_16 = ntohs (uint16_t *netint16)*

Reference: https://linux.die.net/man/3/byteorder

(*) Includes when assigning IP address and port to sockaddr_in data structure

Byte ordering: Python

- Python
  - struct.pack(format-string, v1, v2, …)
    - Return bytes object with values encoded according to format-string
    - Example: Encode unsigned integer i in network byte order in msg
      - msg = struct.pack ("!I", i)
  - struct.unpack(format-string, buffer)
    - Decode and return values packed in buffer according to format-string
    - Example:
      - i = struct.unpack("!I", msg)

Python: See https://docs.python.org/3/library/struct.html

Java: Byte buffers use big-endian by default

Internet transport protocols services

The image part with relationship ID rId3 was not found in the file.

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:* OS maintains state

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?

When to use UDP instead of TCP

- Multicast or broadcast
- Simple request-response type applications
  - Small amount of data
  - Infrequent communications
  - Timeliness
  - But must implement reliability in application
  - For bulk transfers, use TCP (do not want to reinvent the wheel at application layer)

What transport service does an app need?

The image part with relationship ID rId3 was not found in the file.

data integrity

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other 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: common apps

The image part with relationship ID rId3 was not found in the file.

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	
interactive games	loss-tolerant	few kbps up	yes, few secs
text messaging	no loss	elastic	yes, 100's msec
			yes and no

Internet apps: application, transport protocols

The image part with relationship ID rId3 was not found in the file.

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
Name Resolution	DNS	TCP or UDP
Host Configuration	DHCP	UDP

References

Network Programming and Socket API

- Python
  - https://docs.python.org/3/library/socket.html
  - https://docs.python.org/3/library/struct.html
- C
  - https://linux.die.net/man/7/socket
- Java
  - https://docs.oracle.com/javase/tutorial/networking/datagrams/index.html
  - https://docs.oracle.com/javase/tutorial/networking/sockets/index.html