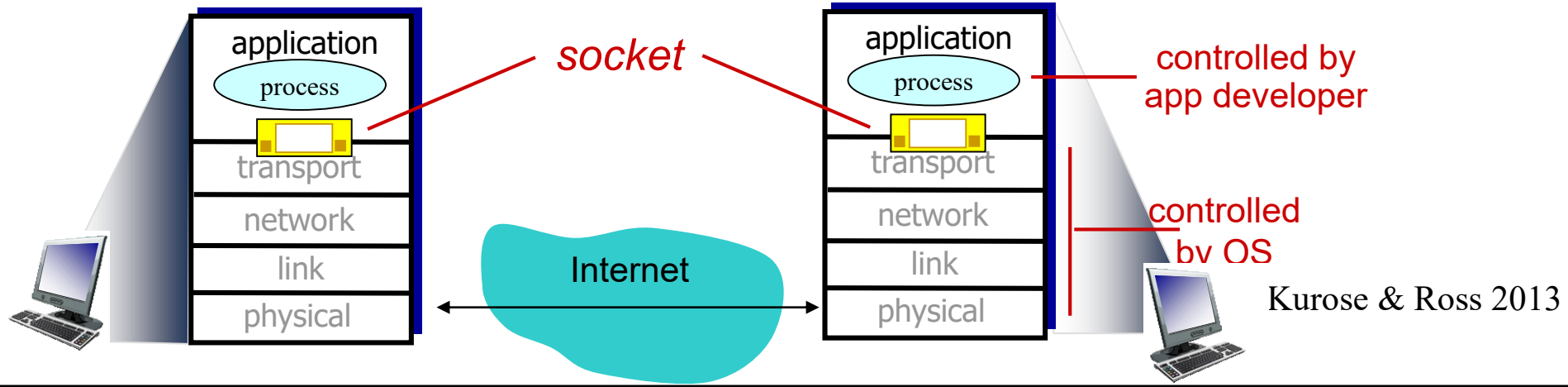
How processes communicate

- **Sockets** provide the application programmers' interface (API) between a **process** and the transport layer (sys/socket.h, java.net).
- User application code runs on end-systems - not network core
- The application programmer needs to specify
  - which transport protocol to use
  - what host to send messages to (e.g. IP address or hostname)
  - what process on the destination host to send messages to (port number)



Internet Transport Services

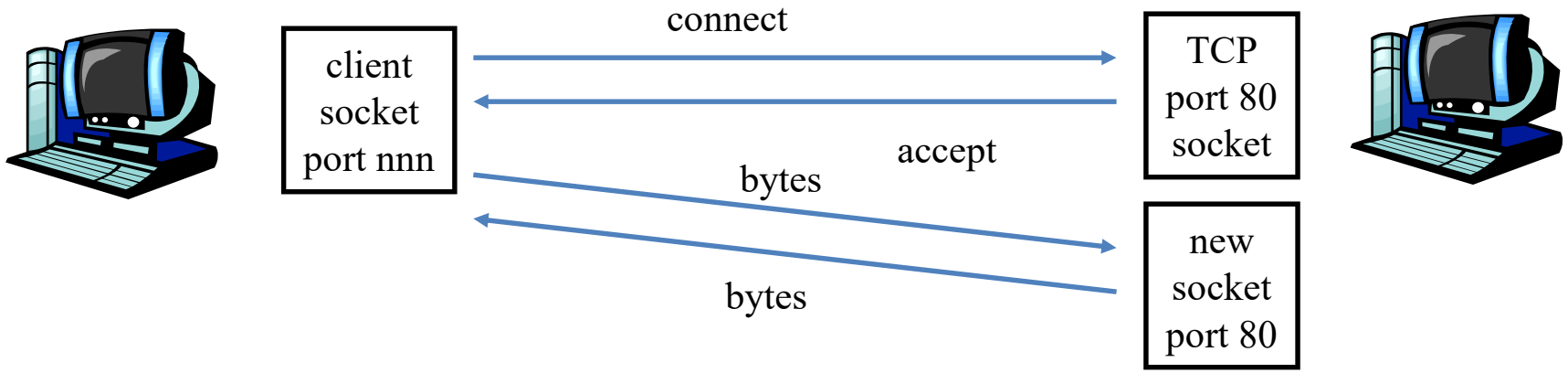
- What services do applications need?
  - Reliable data transfer, Minimum throughput guarantees, Bounded delays, Security
- What do the Internet protocols provide?
  - Reliable data transfer with transmission control protocol TCP
  - Minimal overhead, available bandwidth/delays, no delivery guarantee with user datagram protocol UDP
  - emerging protocols for providing timing and bandwidth guarantees
- Current choices in Internet are TCP or UDP. How does a network application designer decide?

Transport service requirements: common apps

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

Socket Programming with TCP

- Recall that TCP provides a reliable byte stream. All of our data will be going to the same host and port (ie to the same process).
- Assume we want to get a web page. We want to talk to www.foo.com on port 80. If we stay connected to the socket on port 80, how will www.foo.com service other requests?
- port 80 is used to establish a connection on a **second server socket**.

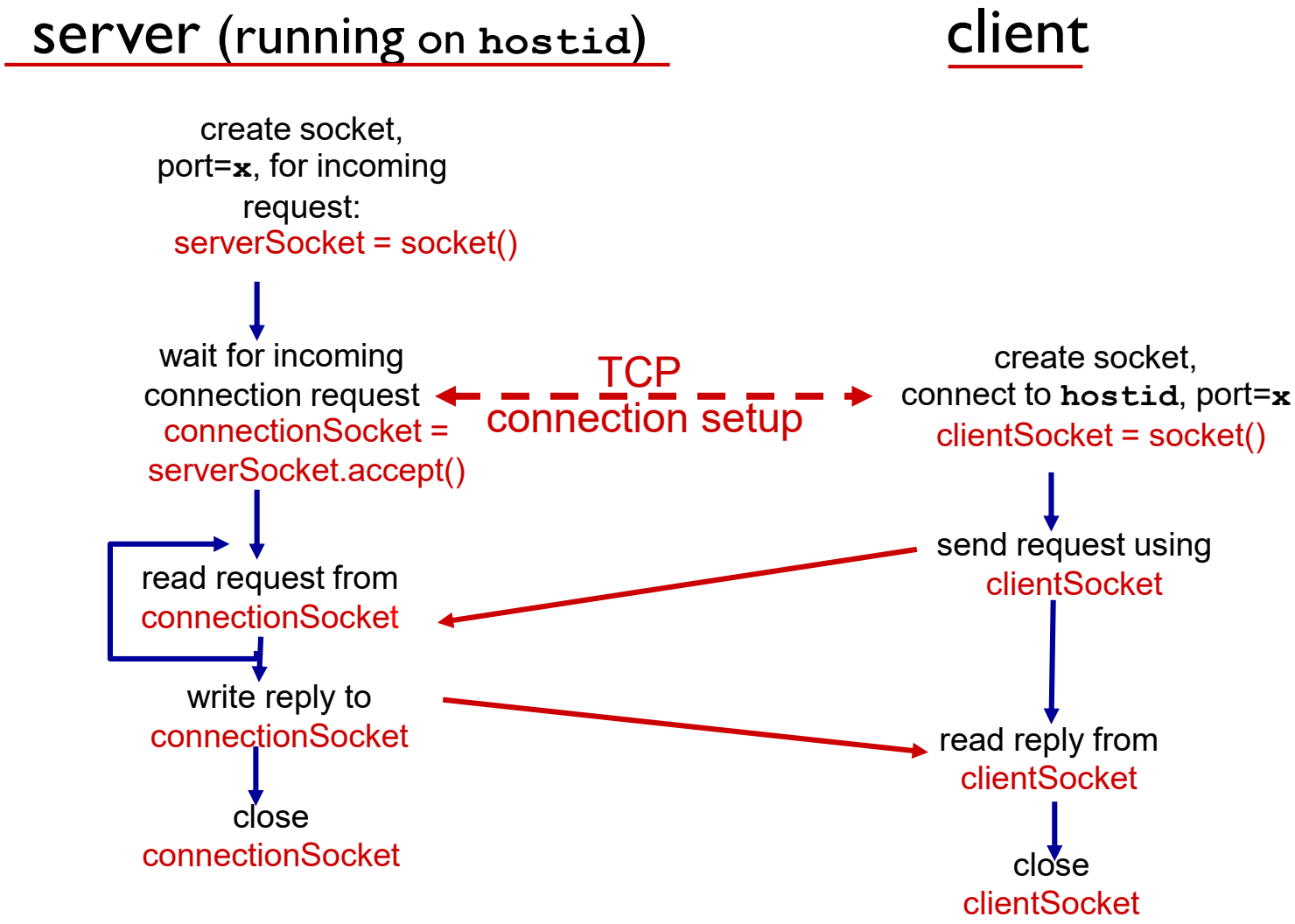


Socket programming with TCP

Application Example:

1. Client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

Client/server socket interaction: TCP



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)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

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 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    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)

Socket programming with UDP

UDP: no "connection" between client and server

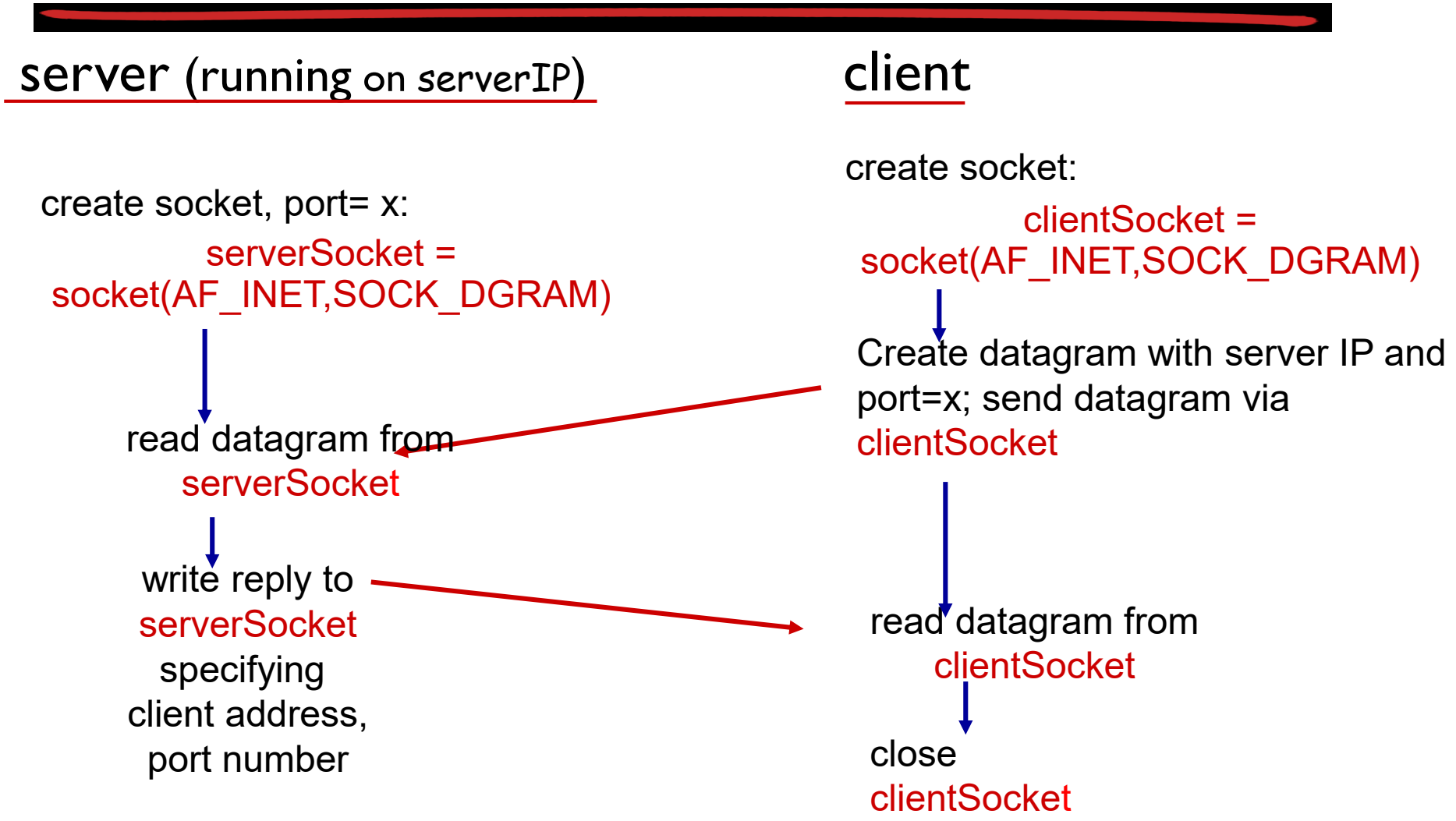
- no handshaking
- sender explicitly attaches IP address and port of destination
- server must extract IP address, port of sender from received datagram

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

application viewpoint

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

Client/server socket interaction: **UDP**



Python UDPClient

include Python's socket library

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(socket.AF_INET,
                      socket.SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message,(serverName, serverPort))
modifiedMessage, serverAddress =
                      clientSocket.recvfrom(2048)
print modifiedMessage
clientSocket.close()
```

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

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print "The server is ready to receive"
while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage, 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

Some hidden parts

- High level languages hide some details that must be dealt with in C.
  - byte ordering - hidden by high level languages,
  - getting IP address in TCP socket - Hidden by Java
- You don't need to know the details of these for this course, but be aware these issues exist.

- multi-threaded (multiple process) servers. Also called concurrent servers are more common than iterative server (this example).