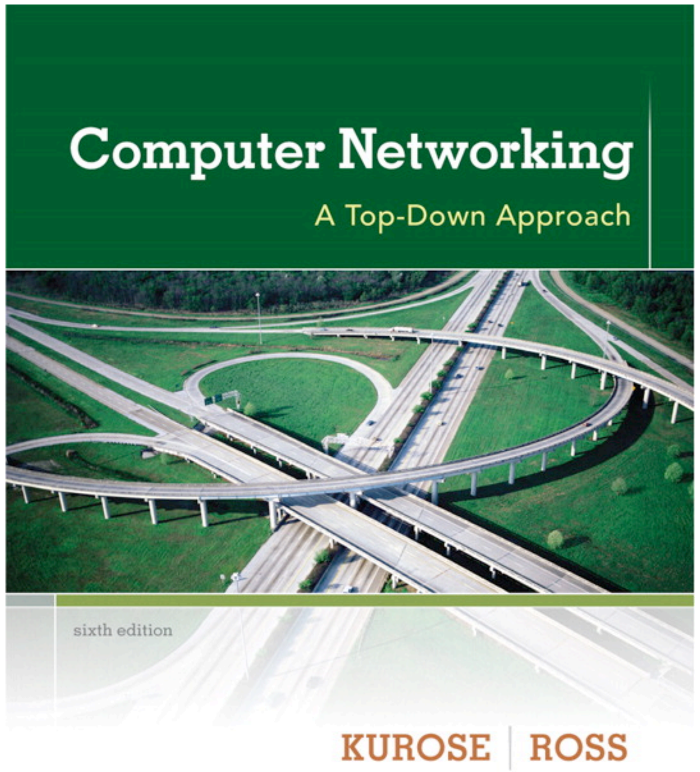
CN-Basic
L25

Socket Programming

Dr. Ram P Rustagi
rprustagi@ksit.edu.in
http://www.rprustagi.com
https://www.youtube.com/rprustagi

Chapter 2
Application Layer



A note on the use of these ppt 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.

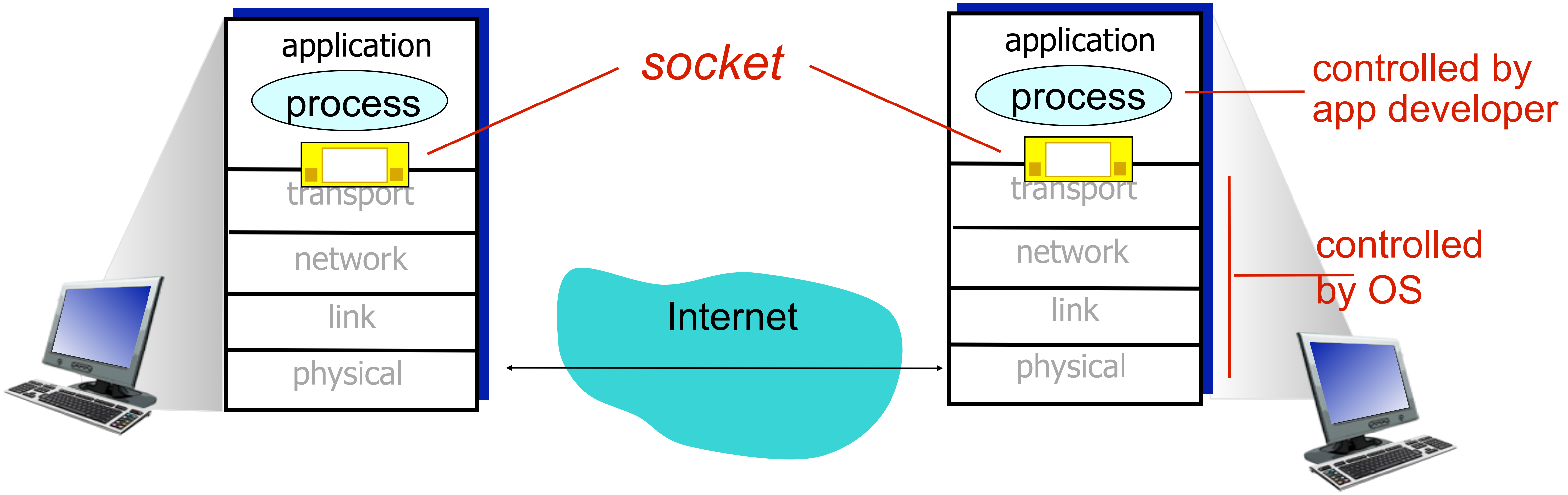
Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

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

- Sockets
  - APIs for applications to read/write data from TCP/IP
  - Provides file abstraction (open, read, write, close)
  - First introduced with TCP/IP
  - Now de-facto standard for TCP/IP programming
  - Uses IP Address and ports
- Ports
  - Server applications uses well known ports
    - E.g. 80/443 - web server, 22 - ssh server
  - Clients use dynamic ports
    - Generally 49152 to 65534

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 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.

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

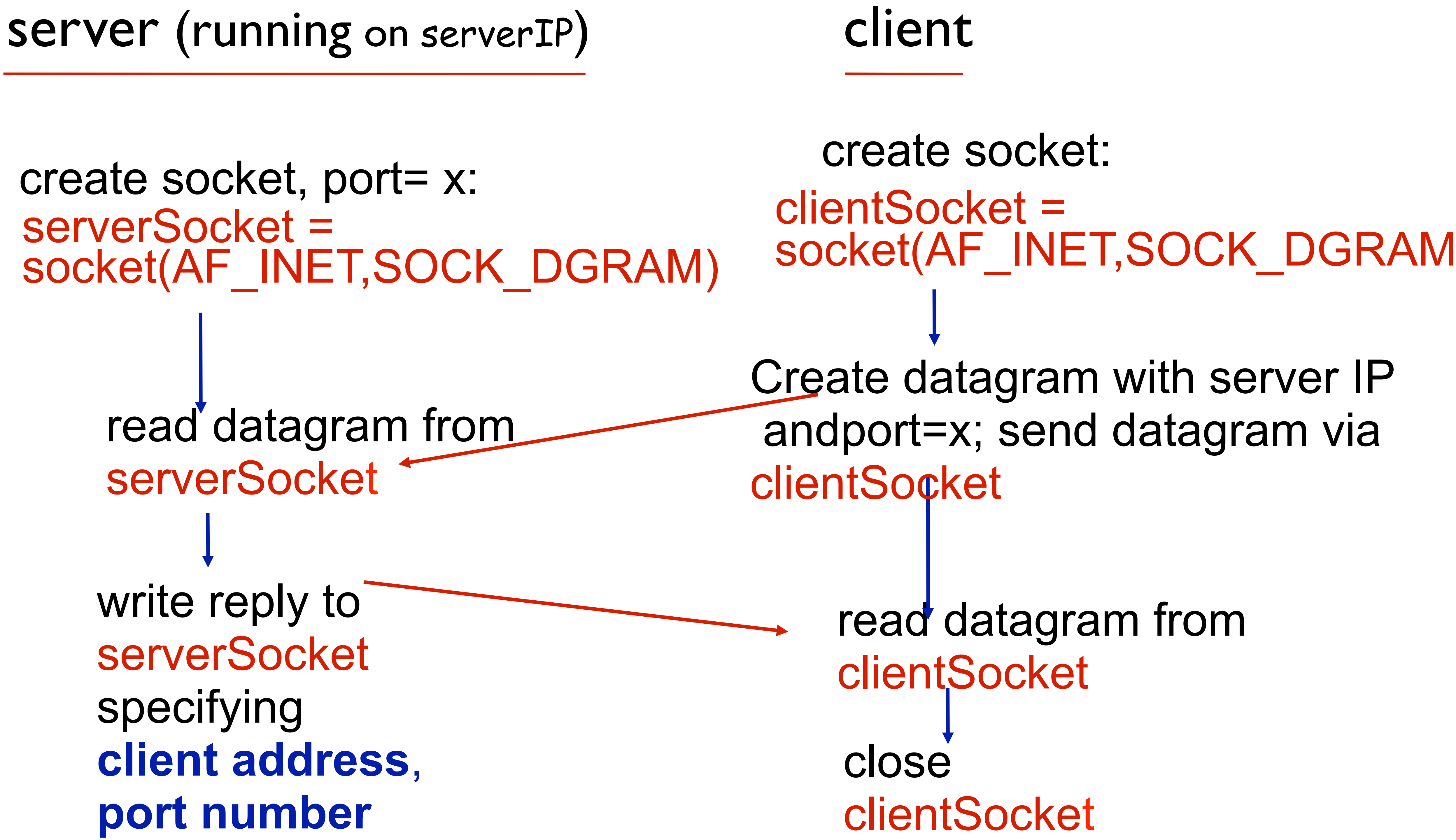
Application viewpoint:

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

Python Tutorial (Brief)

- Src:
  - `https://docs.python.org/3/howto/sockets.html`
    - A good tutorial/reference on python programming
  - `http://docs.python.org/library/socketserver.html`
    - A framework for network servers

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 server IP
andport=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

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

```
from socket import *
sName = ‘hostname’
sPort = 12000
sock = socket(AF_INET, SOCK_DGRAM)
msg = input(’lowercase text:’)
sock.sendto(msg.encode(‘ascii’),
                   (sName, sPort))
rmsg,saddr= sock.recvfrom(2048)
print(rmsg.decode(‘ascii’))
sock.close()
```

Example app: UDP server

Python UDPServer

```
from socket import *
port = 12000
sock = socket(AF_INET, SOCK_DGRAM)
sock.bind(('', serverPort))
print(”Ready to receive:”)
while True:
  msg,caddr= sock.recvfrom(2048)
  msg = msg.decode(‘ascii’)
  msg = msg.upper()

sock.sendto(msg.encode(‘ascii’,
          caddr)
```

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

General Observations

- Hardcoded values
  - Server name, server port
    - Using command line args is desirable, a bit sophisticated
  - Client has only one interaction
    - No continuous interaction
- Python programming characteristics in these programs
  - (`server,port`) are grouped together
  - Two variables on left when reading from socket
    - `msg, caddr = sock.recvfrom(2048)`

Python Programming...

- Modules
  - A a file containing definitions and statements
    - Filename is module name with the suffix .py
  - Can be imported into another or main module
  - Using modules
    - `import socket`
      - Does not enter the name of functions defined in `socket`
      - The functions needs to be accessed using `socket`
    - `from socket import *`
      - Import names directly into the symbol table
      - Does not need to be accessed using `socket`

Python Programming...

- Coding Styles
  - Use 4-space indentation, no tabs
  - Wrap lines so that don't exceed 79 chars
  - Use blank lines to separate functions, larger blocks
  - Preferably, put comments on a line of their own
  - Use spaces around operators after commas
  - Name variables, functions, classes consistently

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:

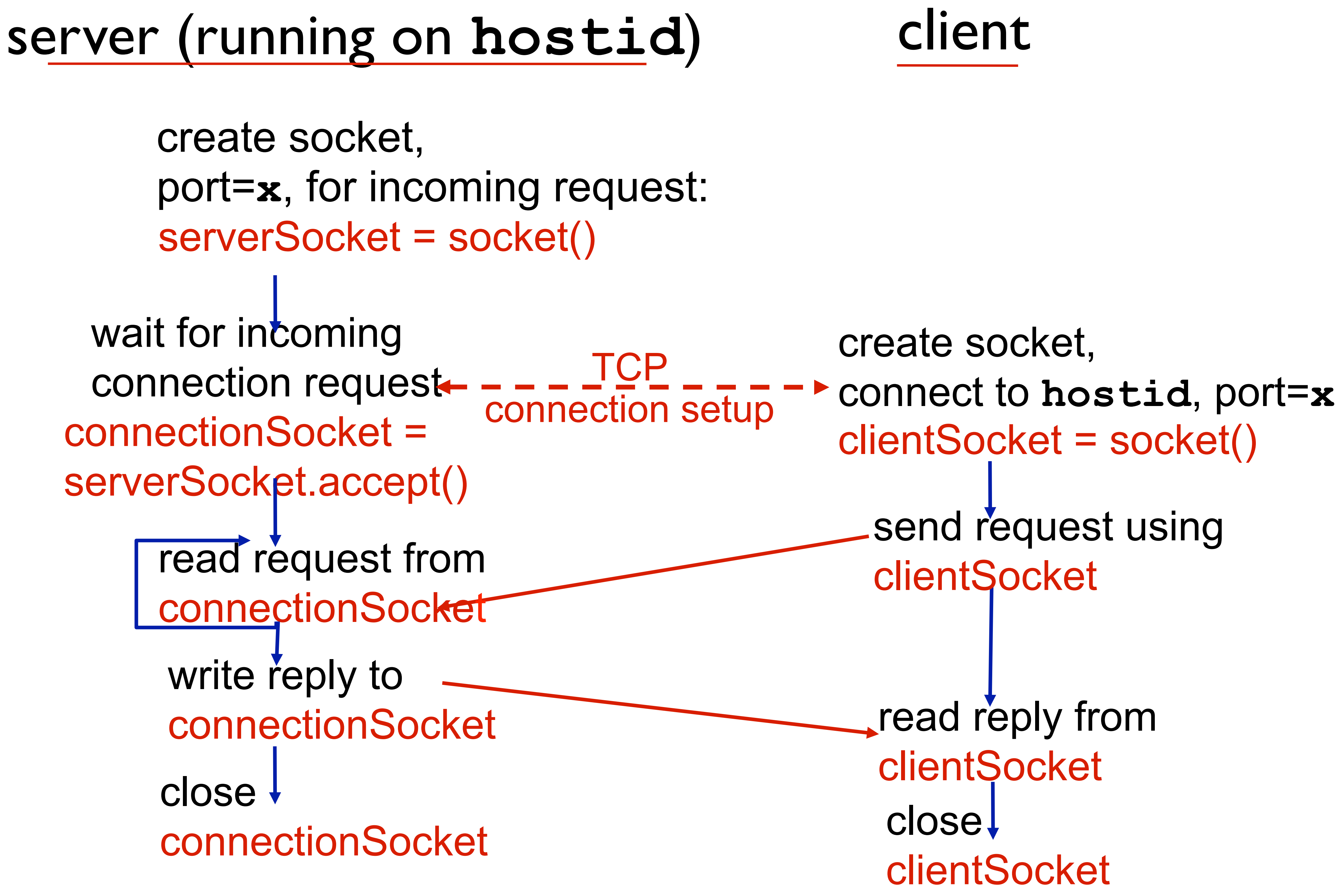
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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

application viewpoint:

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

Client/server socket interaction: TCP



server (running on **hostid**)

client

create socket,
port=**x**, for incoming request:
serverSocket = socket()

wait for incoming
connection request
connectionSocket =
serverSocket.accept()

TCP
connection setup

create socket,
connect to **hostid**, port=**x**
clientSocket = socket()

send request using
clientSocket

read request from
connectionSocket

write reply to
connectionSocket

read reply from
clientSocket

close
connectionSocket

close
clientSocket

Example app: TCP client *Python TCPClient*

```
from socket import *
args = parser.parse_args()
server = args.servername
port = args.serverport
args = parser.parse_args()
server = args.servername
port = args.serverport

sock = socket(AF_INET, SOCK_STREAM)
sock.connect((server, port))
sndmsg = input('Input in lower case sentence:')
sndmsg = sndmsg.encode('ascii')
sock.send(sndmsg)
rcvmsg = sock.recv(recvsize)
rcvmsg = rcvmsg.decode('ascii')
print("Received from server: ", rcvmsg)
sock.close()
```

create TCP socket for server, remote port

No need to attach server name, port

Example app: TCP server

TCPServer.py

```
from socket import *
args = parser.parse_args()
server = args.servername
port = args.serverport
ssock = socket(AF_INET, SOCK_STREAM)
ssock.bind((server, port))
ssock.listen(1)
print  (‘The server is ready to receive’)
while True:
     csock, caddr = ssock.accept()
    rcvmsg = csock.recv(recvsize)
    rcvmsg = rcvmsg.decode('ascii')
    print ("Received data: ", rcvmsg)
    sndmsg = rcvmsg.upper().encode('ascii')
    csock.send(sndmsg)
    csock.close()
```

create TCP welcoming socket

server begins listening for incoming TCP reqs

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 API calls - General

- Creating a socket
  - `socket(AF_INET, SOCK_STREAM)`
  - Creates a new socket object to use with TCP/IP
  - Returns a new socket object
- Setting options
  - `setsockopt(SOL_SOCKET,SO_REUSEADDR,1)`
  - Many options available
  - Releases the port immediately after socket is closed
  - Without this `bind()` call may fail for some time
    - When server application restarted immediately
  - Exercise: run server again after accepting the connection

Socket API calls - Server Side

- Associating to a particular port
    - `bind((hostname, port))`
    - Takes one parameter (as a tuple)
    - Prepares an application to receive connections
- Informing OS to allow connections
    - `listen(n)`
    - Number of connections that can be queued up
    - Different from number of concurrent connections
- Accepting a new connection
    - `accept()`
    - Accepts a new connection
    - It is a blocking call
    - Returns a tuple: allocated `socket` and client address
    - Usually passed to a thread/process

Socket API calls - Client Side

- Connecting to a server
  - `connect(hostname, portNumber)`
  - Connects to a server waiting for connections
  - Hostname should be resolvable by DNS
    - Can be in IP Address
  - It is a blocking call

Socket API calls - Established Connections

- Receiving data
  - `recv(N)`
  - Receives up to N bytes from sockets
  - Blocks until a message is received
  - Return type is string
    - Length 0 of received msg implies detection of disconnect
  - It is stream data, multiple small receive are ok
- Sending data
  - `send(msg)`
  - Sends messages on an established connection
  - Data may not immediately go to other side

Socket API calls - Established Connections

- Closing a connection
  - `close()`
  - Closes an existing established connection
  - If other end does `recv()`, will get 0 length data
- To discontinue usage of connection
  - `shutdown(n)`
    - 0 implies done receiving
    - 1 implies done sending
    - 2 implies done both (sending and receiving)

TCP Connection: Simple programs

- TCPServer.py
  - Loops for ever
  - Handles one connection at a time
  - Receives 1 msg, converts to uppercase, sends response
- TCPClient.py
  - Connects to server
  - Sends data to server
  - Receives response
- TCPServerLoop.py
  - Loops for ever
  - Handles one connection at a time
  - Communicates with client till client closes, then next client

Socket Programming Tutorial

- Exercise:
  - UDP Client sending more than one message
  - Take the sample code
    - Put a for/while loop in client.
    - Invoke multiple clients to talk to same server
    - Study the behavior
  - TCP client sending more than one message
  - Take the sample code
    - Put a for/while loop in client
    - How many clients can wait in queue i.e. Study `listen()`
    - Can multiple clients talk concurrently
      - –If no, spawn a thread to handle each client
    - Study the behavior

TCP Server Example Programs

- Concurrent TCP Server Programming
  - **TCPServerSelect.py**
    - Uses `select()` to determine communicating sockets
    - Allows a server to server concurrent requests
  - **TCPServerThread.py**
    - For each new connection, creates a new thread
    - Thread handles the child request completely
    - The main program just waits/listens for new connections
    - Once a thread is active ^C could have repercussions