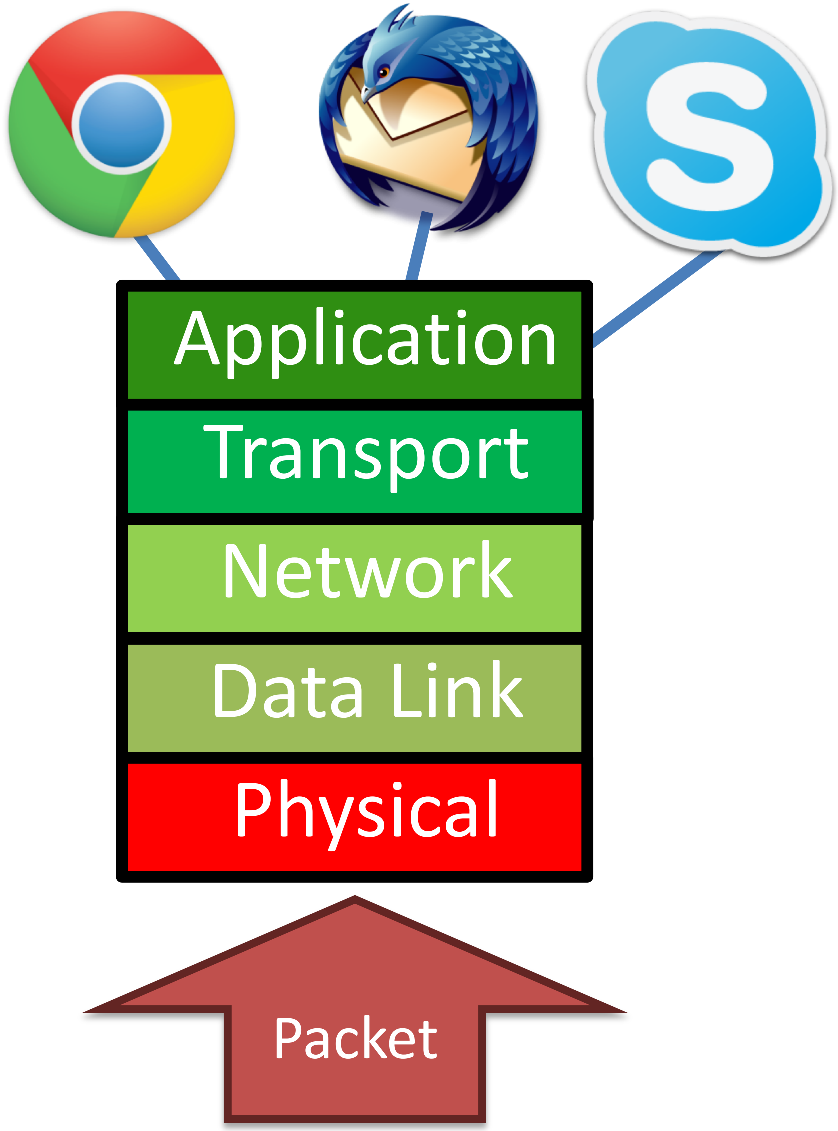
Transport Layer: Introduction to Sockets

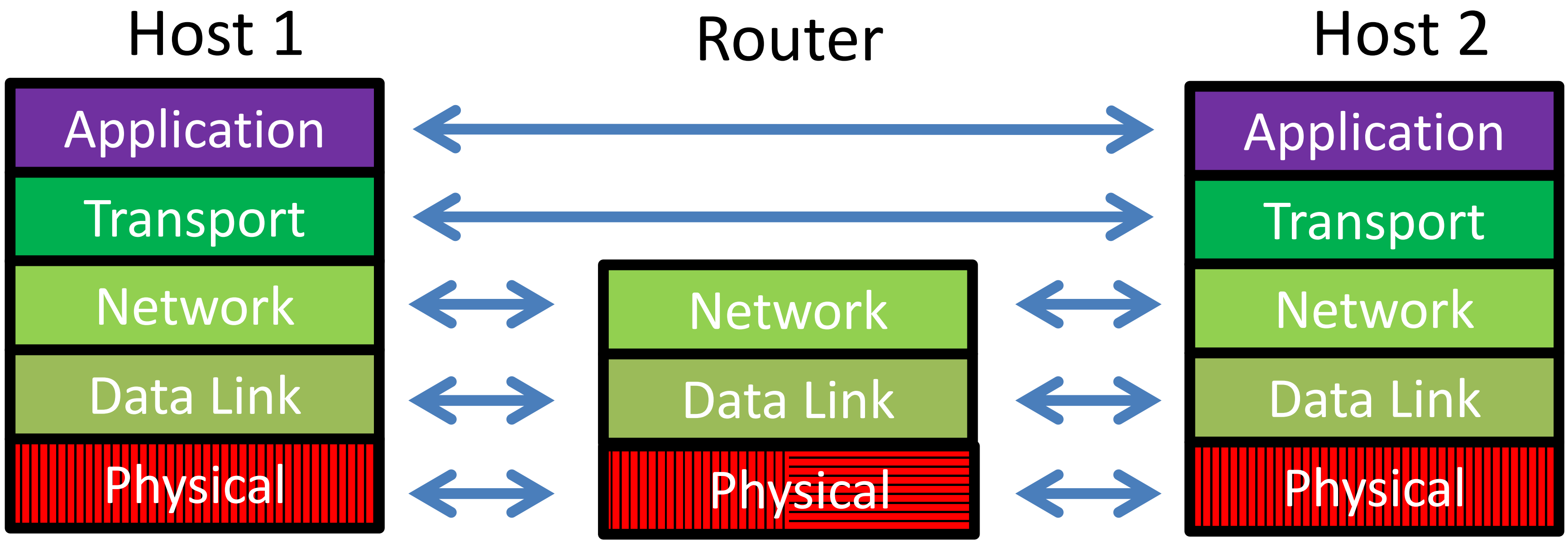
Re-look at the stack

- Headers are “peeled” as you go up the stack

- Headers are added as you go down the stack.



Layering, Revisited



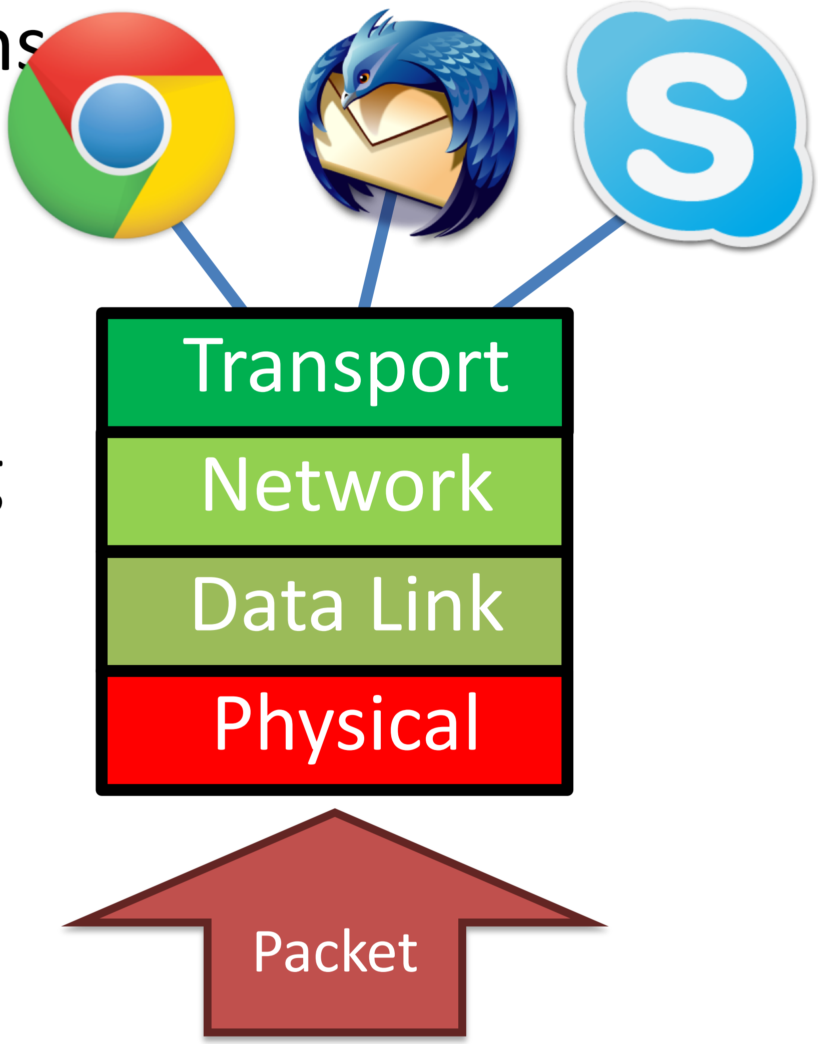
- Lowest level end-to-end protocol
  - Transport header only read by source and destination
  - Routers view transport header as payload
  - Each packet has a Maximum Segment Size (MSS)

Transport layer: TCP

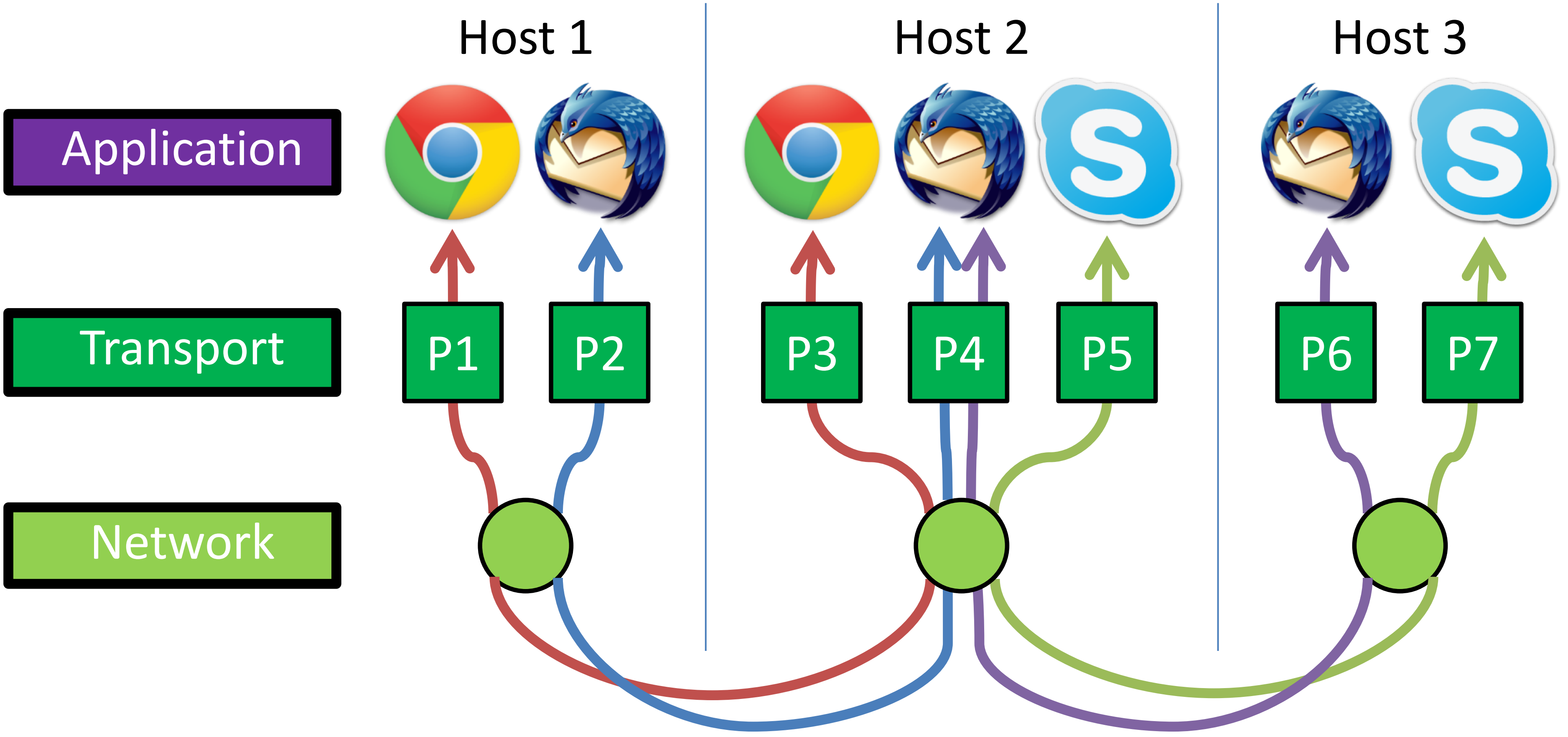
- Transport layer roles
  - “End-to-End” abstraction
  - De-multiplexing

What is de-multiplexing?

- Clients run many applications at the same time
  - Who to deliver packets to?
- Insert Transport Layer to handle demultiplexing using ports
- Rather than IP address, the end point has an IP address and a port.



Demultiplexing Traffic



Endpoints identified by *<src_ip, src_port, dest_ip, dest_port>*

Two types of Transport Protocol

- Transmission Control Protocol (TCP)
  - – Connection oriented
  - – Provides an “in-order delivery” abstraction
  - – Masks unreliability.

- User Datagram Protocol (UDP)
  - – Connection less
  - – No guarantees of in-order delivery.
  - – Does not mask unreliability.

A deep dive into Sockets

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:

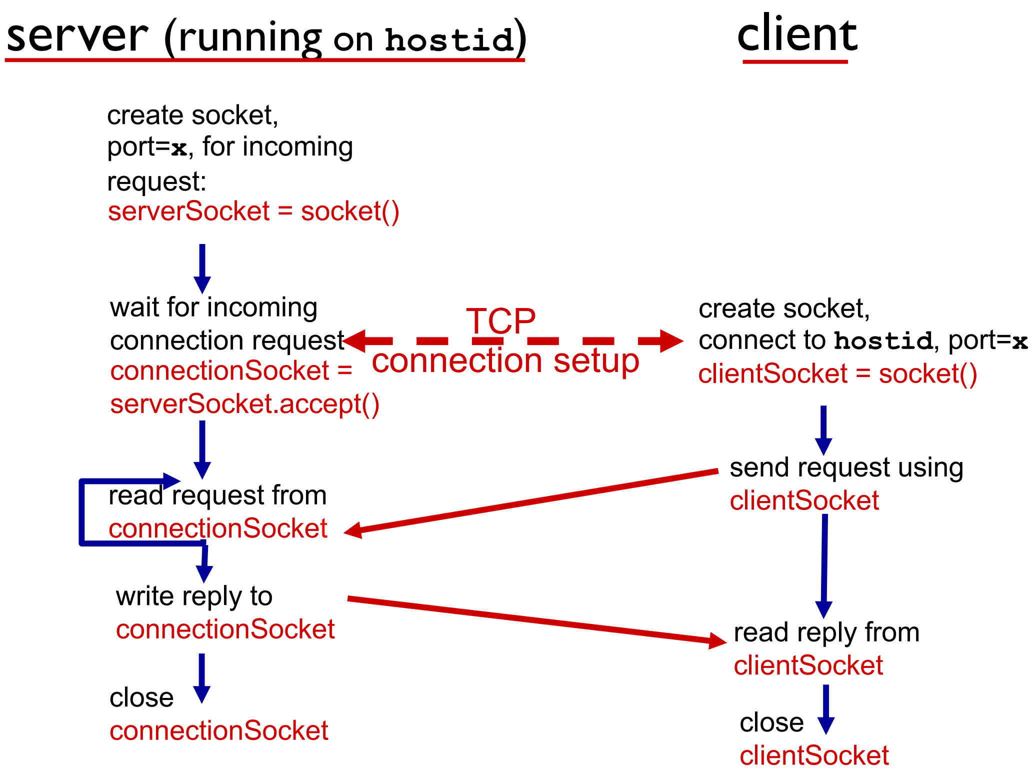
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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



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

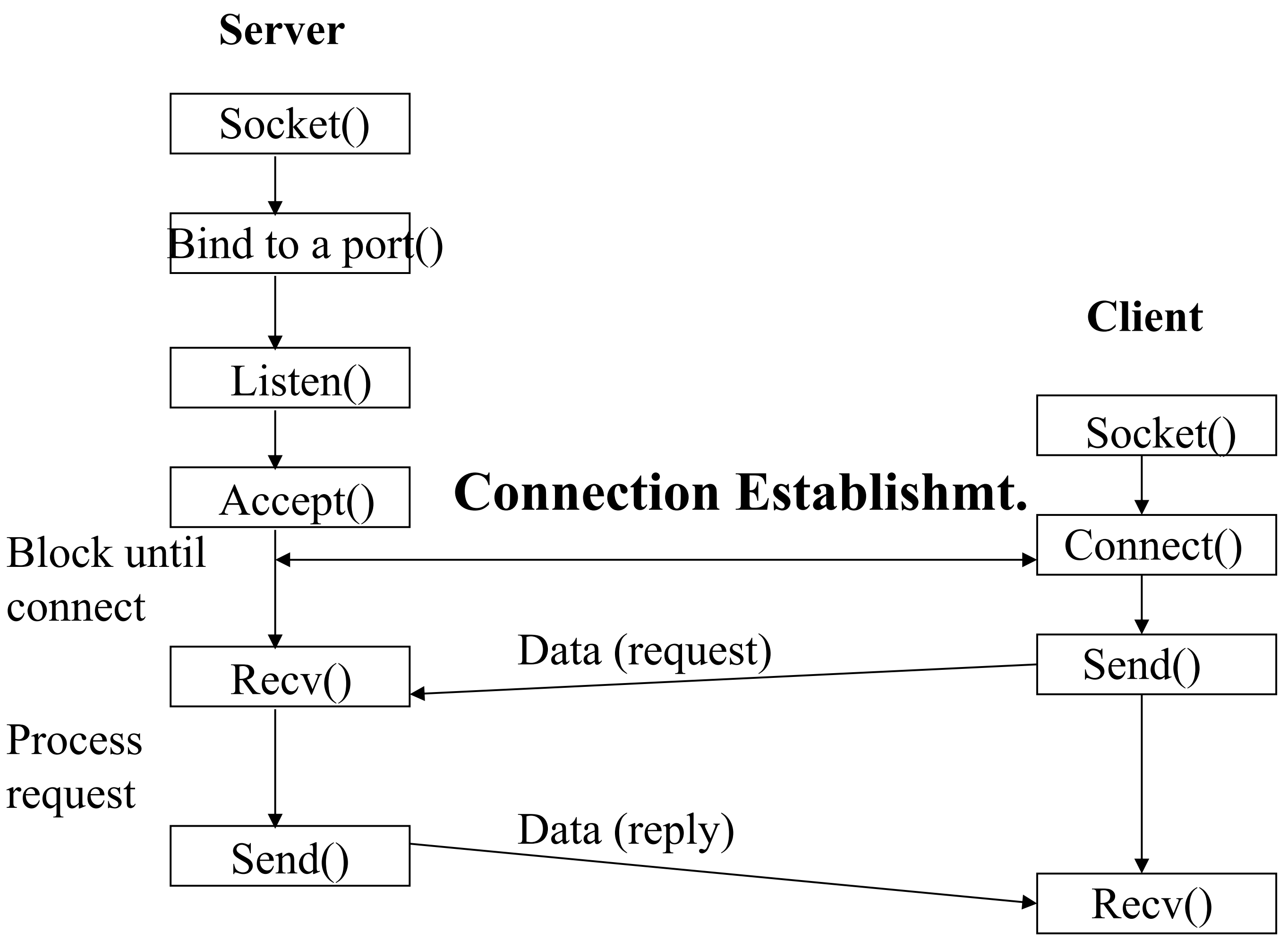
To make a TCP server

- Create a socket with the *socket()* system call.
- Bind the socket to an address using the *bind()* system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the *listen()* system call.
- Accept a connection with the *accept()* system call. This call typically blocks until a client connects with the server.
- Send and receive data using the *read()* and *write()* system calls.

To make a TCP client

- Creates a socket with a socket() call
- Bind the socket to a newly created port
- Assume you already know the IP address and port number of server
- Connect to the server using a connect() call
- Start sending and receiving data.

TCP connection



Server

Socket()

Bind to a port()

Listen()

Accept()

Block until connect

Recv()

Process request

Send()

Connection Establishmt.

Data (request)

Data (reply)

Client

Socket()

Connect()

Send()

Recv()

Handling multiple connection

- To allow the server to handle multiple simultaneous connections, we make the following changes in the above code
- Put the *accept* statement and the following code in an infinite loop.
- After a connection is established, call *fork()* to create a new process.
- Do your processing in the new thread, the accept statement can continue to accept messages

Example app:TCP client

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.encode())
modifiedSentence = clientSocket.recv(1024)
print (‘From Server:’, modifiedSentence.decode())
clientSocket.close()
```

Example app: TCP server

Python TCPServer

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)

```
 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()
```