Unit IV

Network Programming Paradigm

Topics to be Covered

- Network Programming Paradigm
- Socket Programming: TCP & UDP
- Connection oriented, connectionless
- Sock_Stream, Sock_Dgram, socket(), bind(), recvfrom(), sendto(), listen() Server-Client; send(), recv(), connect(), accept(), read(), write(), close()
- Other languages: PowerShell, Bash, TCL
- Demo: Socket Programming in Python

Reference Book:

"Foundations of Python Network Programming" by Brandon Rhodes and John Goerzen

Network Programming Paradigm

- The Network paradigm involves thinking of computing in terms of a client, who is essentially in need of some type of information, and a server, who has lots of information and is just waiting to hand it out.
- Typically, a client will connect to a server and query for certain information. The server will go off and find the information and then return it to the client.
- In the context of the Internet, clients run on desktop or laptop computers attached to the Internet looking for information, whereas servers run on larger computers with certain types of information available for the clients to retrieve.
- The Web itself is made up of a bunch of computers that act as Web servers; they have vast amounts of HTML pages and related data available for people to retrieve and browse. Web clients are used by those of us who connect to the Web servers and browse through the Web pages.
- Network programming uses a particular type of network communication known as sockets.

Network Programming Paradigm…

- In network programming applications that use sockets are divided into two categories
  - the client
  - the server
- The Client/Server Paradigm
  - The client/server paradigm divides software into two categories, clients and servers.
  - A client is software that initiates a connection and sends requests, whereas a server is software that listens for connections and processes requests.
  - In the context of UDP programming, no actual connection is established, .

Role of Transport Layer

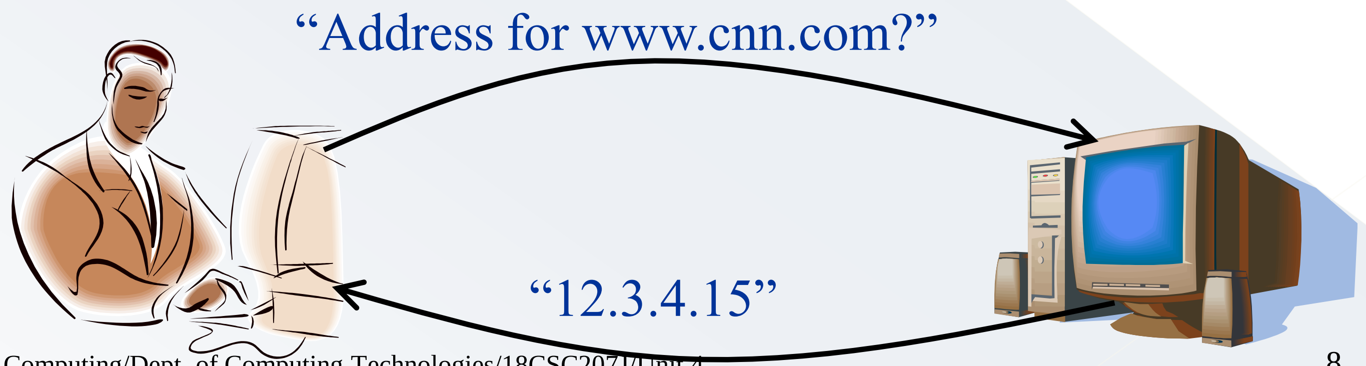
- Application layer
  - Communication for specific applications
  - E.g., HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP), Network News Transfer Protocol (NNTP)
- Transport layer
  - Communication between processes (e.g., socket)
  - Relies on network layer and serves the application layer
  - E.g., TCP and UDP
- Network layer
  - Logical communication between nodes
  - Hides details of the link technology
  - E.g., IP

Transport Protocols

- **Protocols:** These are the set of procedures or rules which govern the flow of data, format of data over the internet.
- Transport protocols Provide *logical communication* between application processes running on different hosts
- Run on end hosts
  - Sender: breaks application messages into segments, and passes to network layer
  - Receiver: reassembles segments into messages, passes to application layer
- Two major protocols over the internet:TCP and UDP

User Datagram Protocol (UDP)

- UDP sends independent packets of data, called datagrams, from one computer to another with no guarantee about delivery
- Example Applications
  - Clock Server
  - Ping
  - Multimedia streaming-E.g., telephone calls, video conferencing, gaming
  - Simple query protocols like Domain Name System



UDP Characteristics

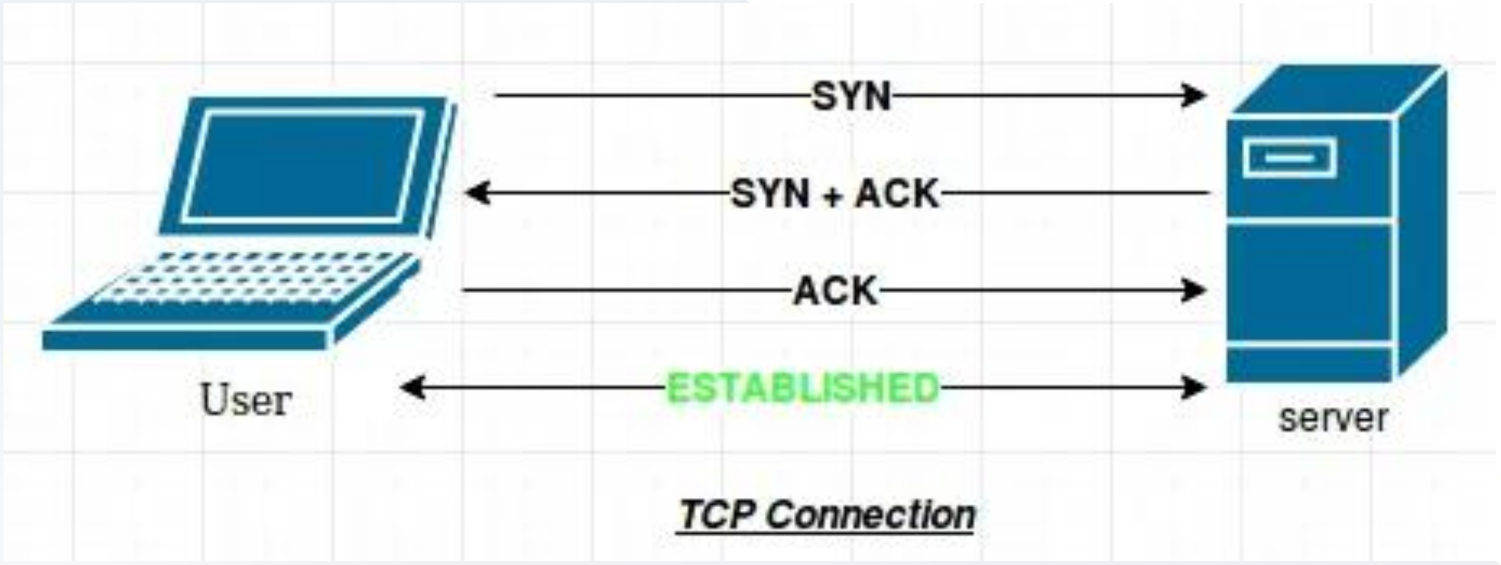
- Connectionless
- Unreliable
- Not ordered
- Datagram-based
- Applications are usually message-based
  - No transport-layer retries
  - Applications handle (or ignore) errors
- Processes identified by port number
- Services live at specific ports
  - Usually below 1024, requiring privilege

Transmission Control Protocol (TCP)

- TCP is a connection-oriented protocol that provides a reliable flow of data between two computers
- Reference Model: Telephone call
- Example Applications
  - File Transfer
  - Chat

Transmission Control Protocol (TCP)

- **Handshake**: It's a way to ensure that the connection has been established between interested hosts and therefore data transfer can be initiated.

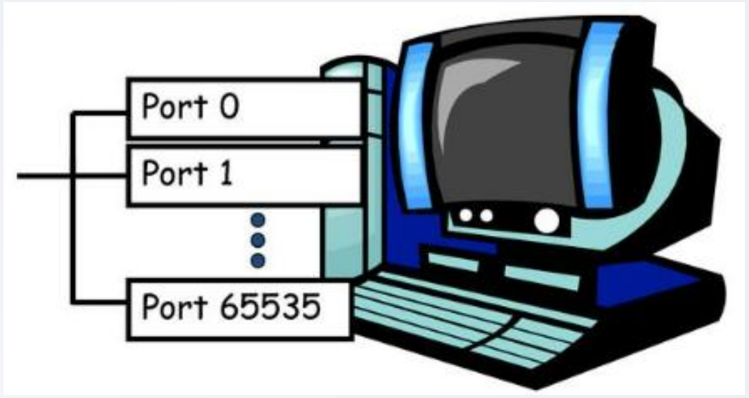


TCP Characteristics

- Connection-oriented
  - Two endpoints of a virtual circuit
- Reliable
  - Application needs no error checking
- Ordered:
  - The messages are delivered in a particular order in which they were meant to be.
- Stream-based
  - No predefined blocksize
- Processes identified by port numbers
- Services live at specific ports

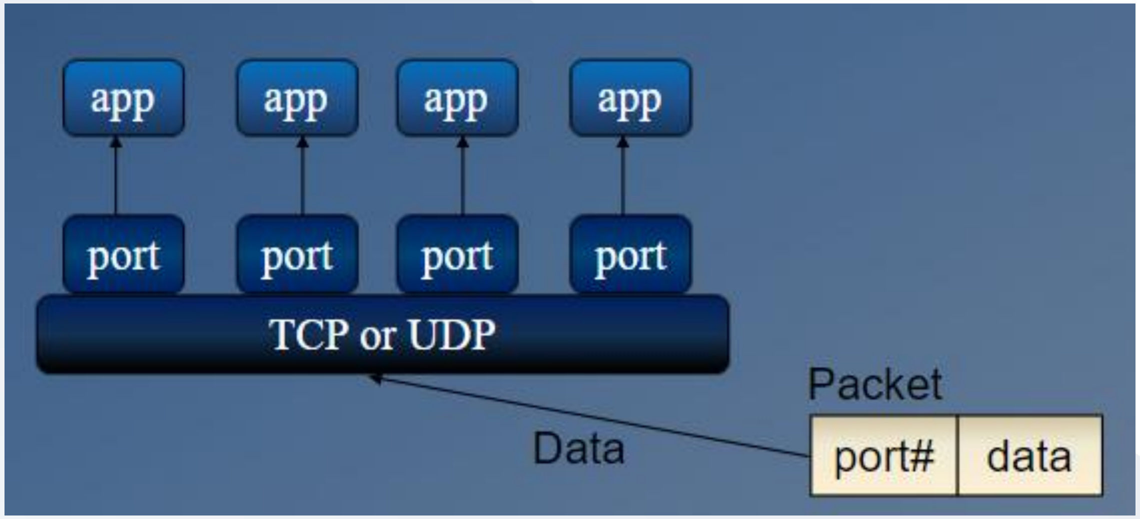
IP Addresses and Ports

- **IP addressess** are the addresses which helps to uniquely identify a device over the internet
- **Port** is an endpoint for the communication in an operating system.
- A system might be running thousands of services but to uniquely identify a service on a system the application requires a port number.
- There are total of **0 – 65535** ports on a system.



Understanding Ports

- The TCP and UDP protocols uses ports to map incoming data to a particular process running on a computer



Understanding Ports...,

- Port is represented by a positive(16-bit) integer value
- Each host has 65536 ports.
- **Well known ports**: Some ports are reserved to support common or well known services (0 t0 1023)
  - 21- FTP
  - 23 – TELNET
  - 25 –SMTP
  - 80 –HTTP
  - 53 - Domain Name System(DNS)
- **Ephemeral ports**: User level process/services >=1024

Socket Programming

- Sockets provides an interface for programming networks at the transport layer.
- *Two socket types for two transport services:*
  - *UDP:* unreliable datagram
  - *TCP:* reliable, byte stream-oriented
- A socket provides an interface to send data to/from the network through a port
- Socket API is the programming interface you need to use for transmitting and receiving messages.
- It is almost like a kind of door, a door to transport layer controlled by OS

Defining Socket

- A socket is a software abstraction for an input or output medium of communication.
- Sockets allow communication between processes that lie on the same machine, or on different machines working in diverse environment and even across different continents.
- Sockets are the end-point of a two-way communication channel.
- An endpoint is a combination of IP address and the port number.
- For Client-Server communication,
  - Sockets are to be configured at the two ends to initiate a connection
  - Listen for incoming messages
  - Send the responses at both ends
  - Establishing a bidirectional communication

Defining Socket...

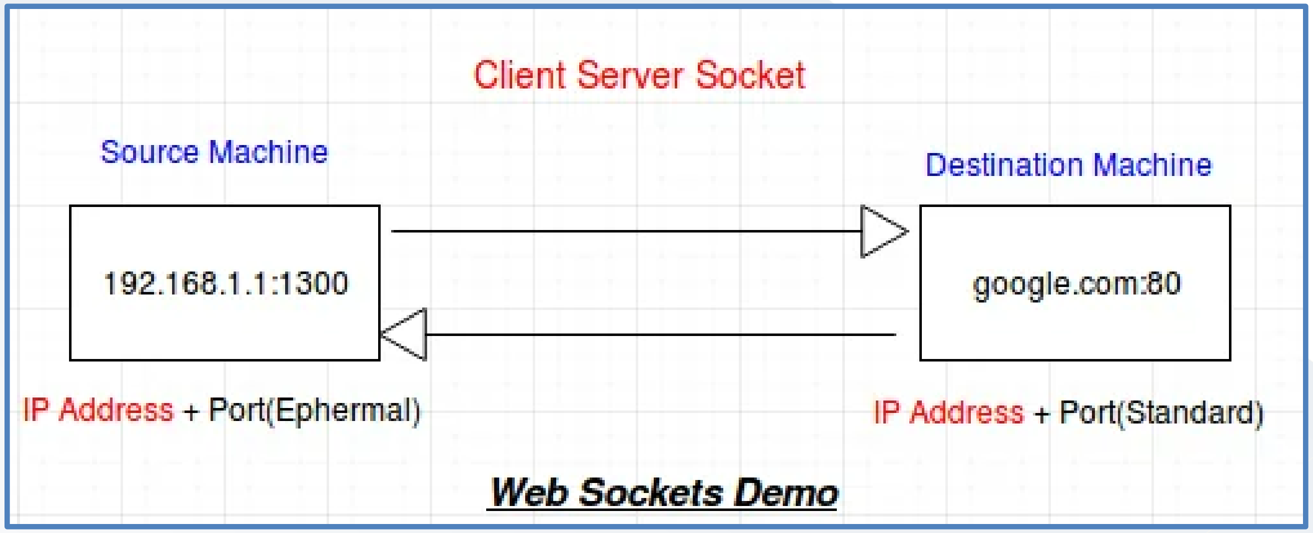
- Sockets were initially used in 1971 and later it turned into an API in the Berkeley Software Distribution (BSD) operating system which was introduced in the 1983 called Berkeley socket.
- The client-server applications are the most common type of the socket applications. In this type of application, one side is a server and waits for establish the connection to the client.
- Now, there is two endpoints **Client Socket** and a **Server Socket.**

Difference between Client socket and Server socket

- Client sockets closes after the request is complete, where the server sockets are not short lived.
- Client socket uses the Ephermal Port for connection, and server socket needs a standard or well defined port connection such as Port 80 for Normal HTTP Connection, Port 23 for Telnet etc.
- Client socket is an endpoint for communication, where server socket waits for request to come over the internet.

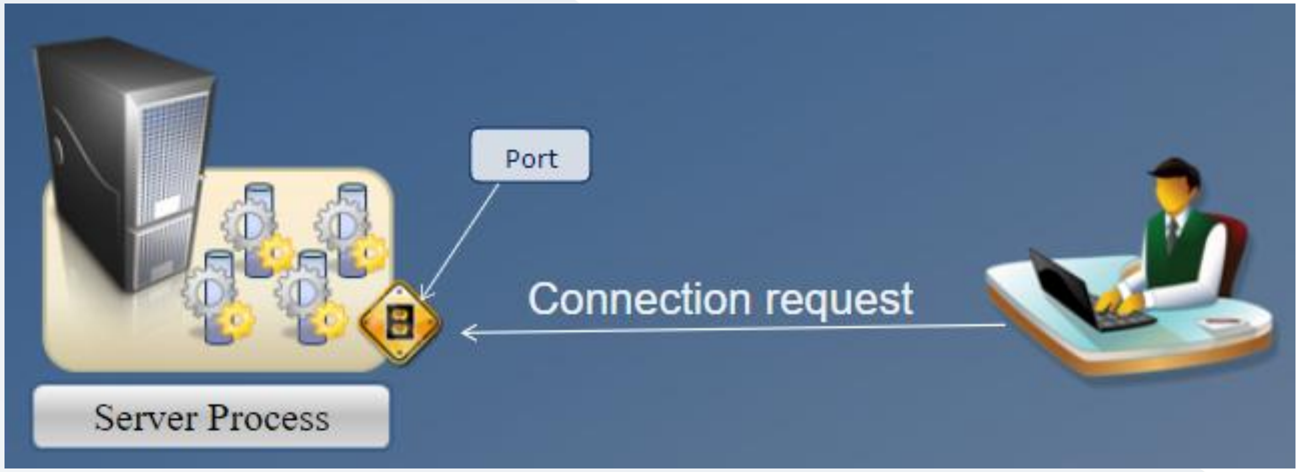
Socket Programming

- When the client tries to connect with the server, a random port is assigned by the OS for the connection. This random port is called **Ephermal Port**.
- In the figure, 1300 is an ephermal port on the source(client) machine.
- The client socket is short lived, i.e as soon as the data exchange ends it closes.



Server

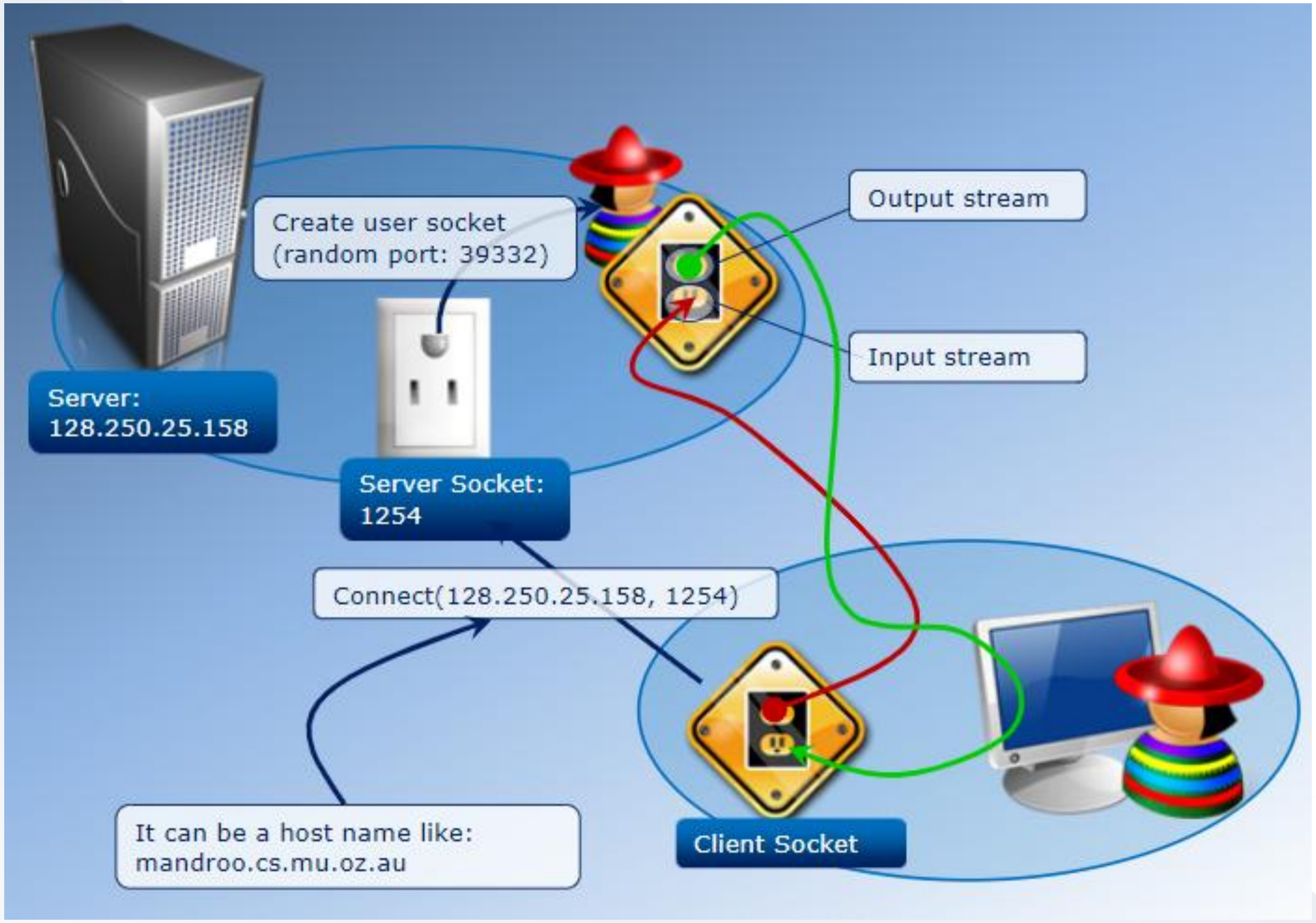
- A server(program) runs on a specific computer and has socket that is bound to a specific port.
- The server waits and listens to the socket for a client to make a connection request.



Connecting a server and a client socket

- If everything goes well, the server accepts the connection.
- Upon acceptance, the server gets a new socket bound to a different port.
- The server creates a new socket so that it can continue to listen to the original socket for connection requests while serving the connected client.

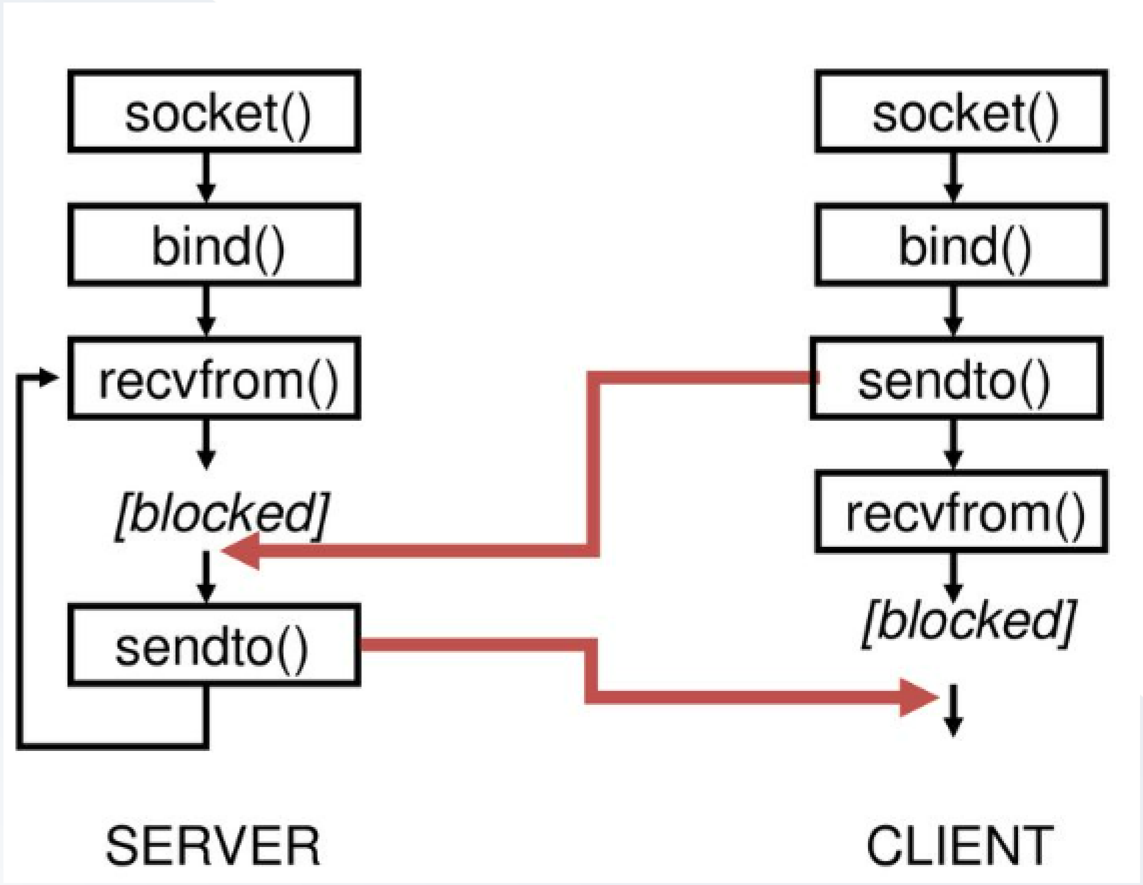
Connecting a server and a client socket…



Connection oriented, connectionless

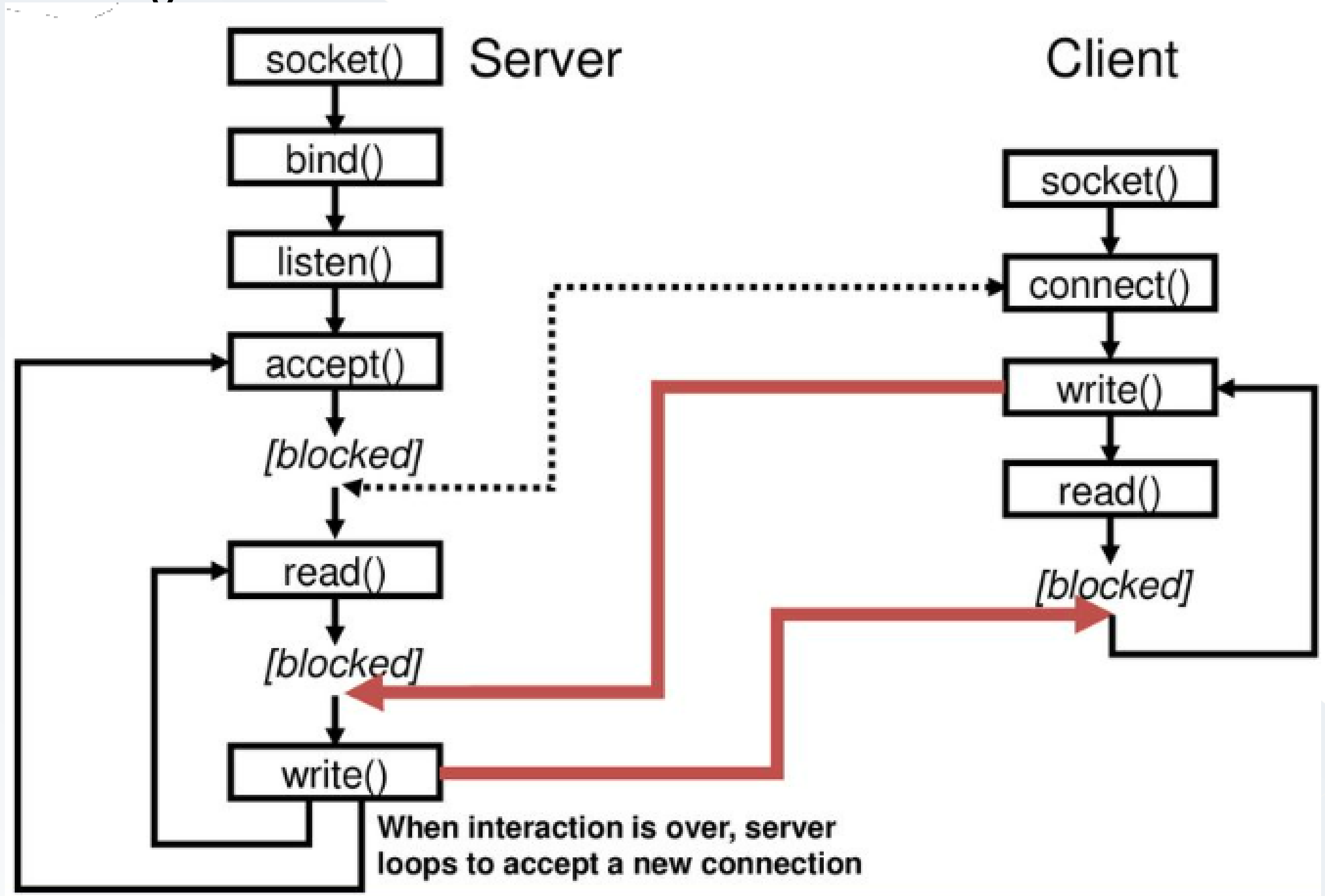
Connectionless Service

- bind(), recvfrom(), sendto(), listen()



Connection-oriented Services

- send(), recv(), connect(), accept(), read(), write(), close()



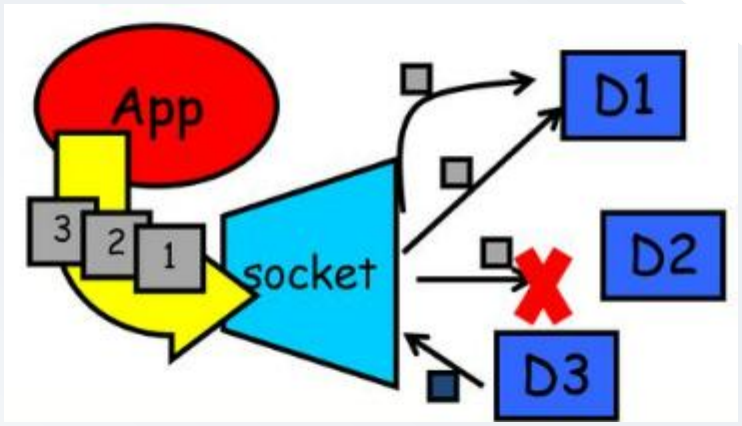
Types of Socket

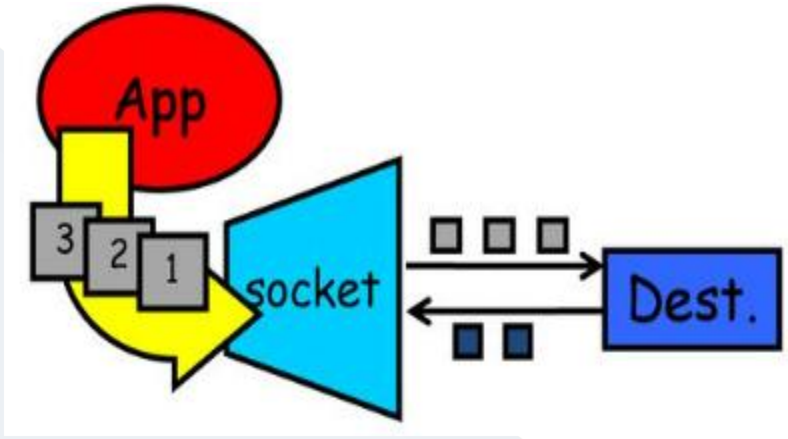
- Datagram Socket (SOCK_DGRAM)
  - A datagram is an independent, self-contained piece of information sent over a network whose arrival, arrival time, and content are not guaranteed.
  - A datagram socket uses User Datagram Protocol (UDP) to facilitate the sending of datagrams (self-contained pieces of information) in an unreliable manner.
  - Unreliable means that information sent via datagrams is not guaranteed to make it to its destination.
- Stream Socket (SOCK_STREAM)
  - A stream socket, or connected socket, is a socket through which data can be transmitted continuously.
  - A stream socket is more akin to a live network, in which the communication link is continuously active.
  - A stream socket is a "connected" socket through which data is transferred continuously

Types of Socket…

Stream socket

Datagram socket





Types of Socket…

SOCK_STREAM	SOCK_DGRAM
It is used for TCP protocols	It is used for UDP protocols
It has reliable delivery	It has unreliable delivery
It provides guaranteed correct ordering of packets.	There is no order guaranteed.
It is connection-oriented	There is no notion of connection
It is bidirectional	It is not Bidirectional (Can send or receive

Sockets…

- **Domain**
  - The family of protocols that is used as the transport mechanism. These values are constants such as AF_INET, PF_INET, PF_UNIX, PF_X25, and so on.
- **Type**
  - The type of communications between the two endpoints, typically SOCK_STREAM for connection-oriented protocols and SOCK_DGRAM for connectionless protocols.
- **protocol**
  - Typically zero, this may be used to identify a variant of a protocol within a domain and type.

Sockets…

- **hostname**
  - The identifier of a network interface –
  - A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation
  - A string "<broadcast>", which specifies an INADDR_BROADCAST address.
  - A zero-length string, which specifies INADDR_ANY, or
  - An Integer, interpreted as a binary address in host byte order.
- **port**
  - Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service.

Demo: Socket Programming in Python

Python Network Services

- Python provides two levels of access to network services.
  - At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.
  - Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

Python Socket Module

- Python socket module provides the socket() method that let programmers' set-up different types of socket virtually
- Syntax:

  s = socket.socket (socket_family, socket_type, protocol=value)

  - **socket_family** – Defines family of protocols used as transport mechanism. Either This is either AF_UNIX or AF_INET
  - **socket_type** – Defines the types of communication between the two end-points.
    - SOCK_STREAM (for connection-oriented protocols, e.g., TCP)
    - SOCK_DGRAM (for connectionless protocols e.g. UDP).
  - **protocol** – This is usually left out, or defaulting to 0.

Server Socket Methods

- Once you have *socket* object, then you can use required functions to create your client or server program.
- **s.bind()**
  - This method binds address (hostname, port number pair) to socket.
- **s.listen(backlog)**
  - This method sets up and start TCP listener.
  - The backlog is the maximum number of queued connections that must be listened before rejecting the connection.
- **s.accept()**
  - This passively accept TCP client connection, waiting until connection arrives (blocking)
  - The return value is a pair(conn, address)

Client Socket Methods

- **s.connect()**
  - This method actively initiates TCP server connection.
  - This function is used to set up a connect to a remote socket at an address. An address format contains the host and port pair which is used for **AF_INET** address family.

General Socket Methods

- **s.recv():** receives TCP message
- **s.send():** transmits TCP message
- **s.recvfrom():** receives UDP message
- **s.sendto():**transmits UDP message
- **s.close():**closes socket
- **socket.gethostname():**Returns the hostname.

Python Internet modules

Protocol	Common function	Port No	Python module
HTTP	Web pages	80	httplib, urllib, xmlrpclib
NNTP	Usenet news	119	nntplib
FTP	File transfers	20	ftplib, urllib
SMTP	Sending email	25	smtplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Telnet	Command lines	23	telnetlib
Gopher	Document transfers	70	gopherlib, urllib

Socket in Python...

- Example: #Socket client example in python

```
import socket
#create an AF_INET, STREAM socket (TCP)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print 'Socket Created'
```

Socket Creation

```
import socket import sys
try:
    #create an AF_INET, STREAM socket (TCP)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error, msg:
    print 'Failed to create socket.
    Error code: ' + str(msg[0]) + ' , Error message : ' + msg[1]
    sys.exit();
print 'Socket Created'
```

A Simple Server

```
import socket                                    # Import socket module

s = socket.socket()                              # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345                                     # Reserve a port for your
    service.
s.bind((host, port))                             # Bind to the port

s.listen(5)                    # Now wait for client connection.
while True:
  c, addr = s.accept()      # Establish connection with client.
  print 'Got connection from', addr
  c.send('Thank you for connecting')
  c.close()                                      # Close the connection
```

A Simple Client

```
import socket                    # Import socket module

s = socket.socket()         # Create a socket object
host = socket.gethostname() # Get local machine
    name
port = 12345                  # Reserve a port for your
    service.

s.connect((host, port))
print s.recv(1024)
s.close()                      # Close the socket when done
```

Client-Server Program…

- Now run this server.py in background and then run above client.py to see the result.
- Following would start a server in background.

```
$ python server.py
```

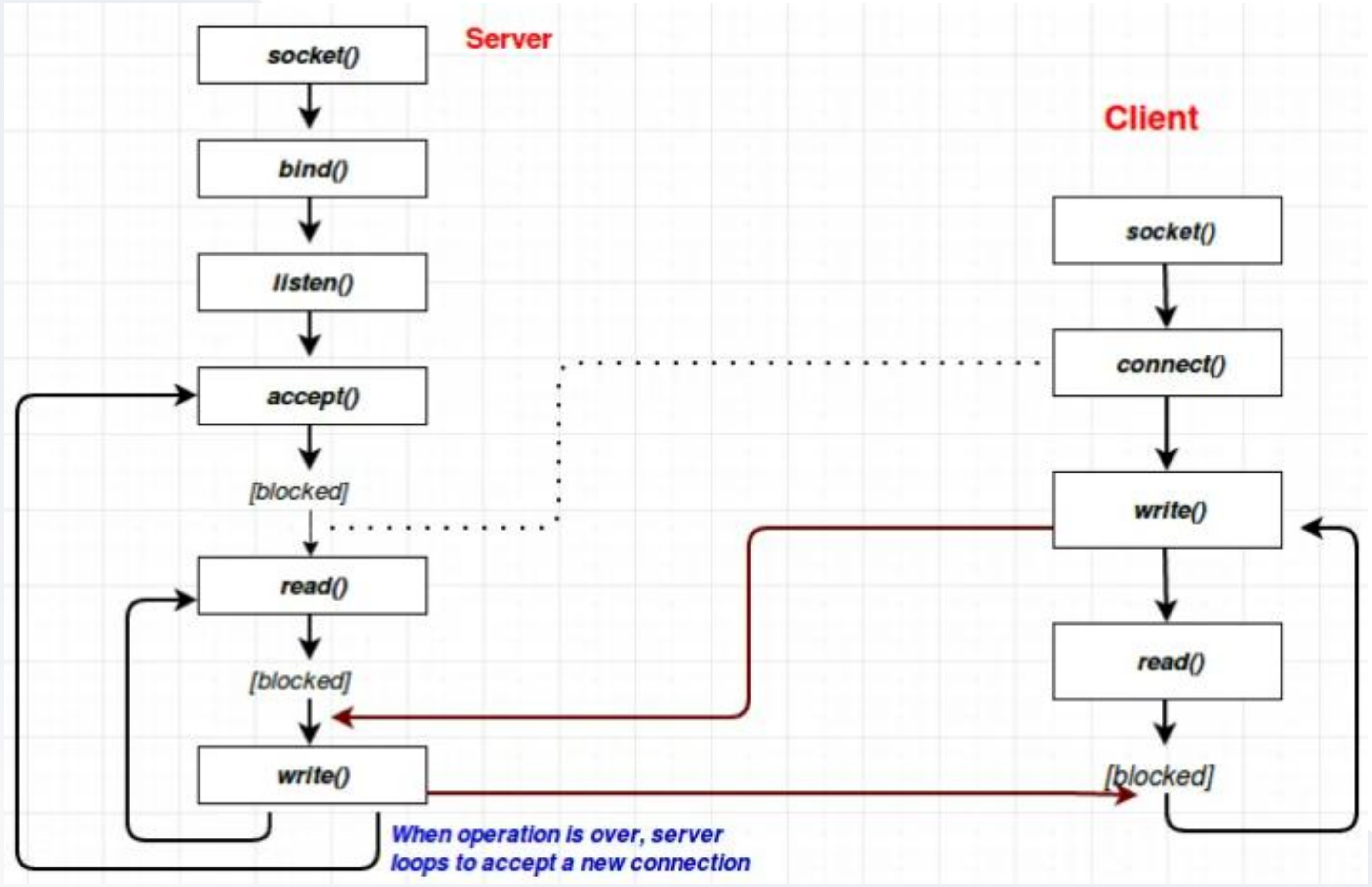
- Once server is started run client as follows:

```
$ python client.py
```

- This would produce following result –

```
Got connection from ('127.0.0.1', 48437)
Thank you for connecting
```

Flow Diagram of the Program



Working with UDP Sockets

- Syntax for UDP socket :

  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

Simple UDP Server program

```
import socket sock =
    socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
udp_host = socket.gethostname()                    # Host IP
udp_port = 12345
sock.bind((udp_host,udp_port))
while True:
    print "Waiting for client..." data,addr =
    sock.recvfrom(1024)               #receive data from client
print"Received Messages:",data," from",addr
```

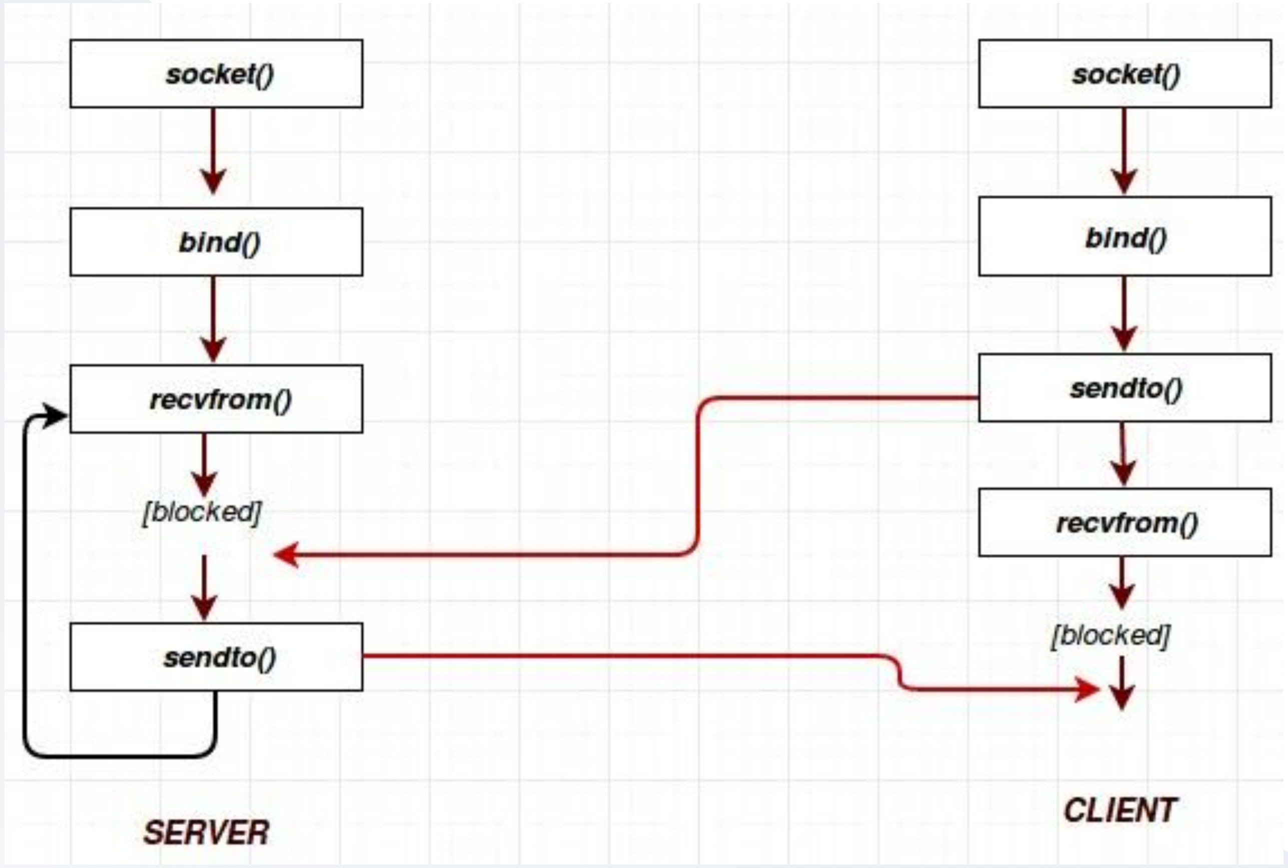
Output of the above script:

```
Waiting for client...
```

Simple UDP Client

```
import socket sock =
    socket.socket(socket.AF_INET,socket.SOCK_DGRA
    M)
udp_host = socket.gethostname()
udp_port = 12345 # specified port to connect
msg = "Hello Python!"
print "UDP target IP:", udp_host
print "UDP target Port:", udp_port
sock.sendto(msg,(udp_host,udp_port))
```

Flow Diagram of the Program



Example2

- Echo Client-Server Program
- Handling received data by adding them

Echo Server

```
import socket
host = socket.gethostname()
port = 12345
s = socket.socket()                                    # TCP socket object
s.bind((host,port))
s.listen(5)
print "Waiting for client..."
conn,addr = s.accept()      # Accept connection when client connects
print "Connected by ", addr
while True:
    data = conn.recv(1024)           # Receive client data
    if not data: break               # exit from loop if no data
    conn.sendall(data)  # Send the received data back to
client conn.close()
```

Echo Client

```
import socket
host = socket.gethostname()
port = 12345
s = socket.socket()                                    # TCP socket object
s.connect((host,port))
s.sendall('This will be sent to server') # Send This message to server
data = s.recv(1024)          # Receive the echoed  data from server
print data                                   # Print received(echoed) data
s.close()
```

Example 2:Handling received data by adding them

- Modify the previous program to send data to the server and the server will sum up the data and will send back to the clien
- Changes in Server Program

```
while True:
# Receive client data
    data = conn.recv(1024)
# Split the received string using , as separator and store in list 'd
    d=data.split(“,”)

#Add the content after converting into int
data_add=int(d[0])+ int d[1])

# Send the added data as string
conn.sendall(data_add)
```

Client Program

```
import socket
host = socket.gethostname()
port = 12345
a = str(raw_input('Enter first number: '))
b = str(raw_input('Enter second number: ')) =
c = a+','+b                                  # Generate a string from numbers
print "Sending string {0} to server" .format(c)
s = socket.socket()
s.connect((host,port))
s.sendall(c)                                 # Send string 'c' to server
data = s.recv(1024)                          # receive server response print
int(data)                                    # convert received dat to 'int'
s.close()
```

- run add_server.py first and after that run add_client.py
- <u>Output</u>

```
Enter First Number: 50
Enter Second Number: 40
Sending string 50,40 to Server
90
```

Exercise Program

Create a UDP Client Server Application

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.

Example app: TCP client

Python TCPClient

create TCP socket for server, remote port 12000

No need to attach server name, port

```
import socket
serverName = ‘servername’
serverPort = 12000
clientSocket = socket.socket(socket.AF_INET,
                             socket.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()
```

Example app: TCP server

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)

Exercise Programs

- File transfer
  a) send a file from a local server to a local client.
  b) server interaction with multiple clients
- Chat server and client
  - The server is like a middle man among clients. It can queue up to 10 clients. The server broadcasts any messages from a client to the other participants. So, the server provides a sort of chatting room.