

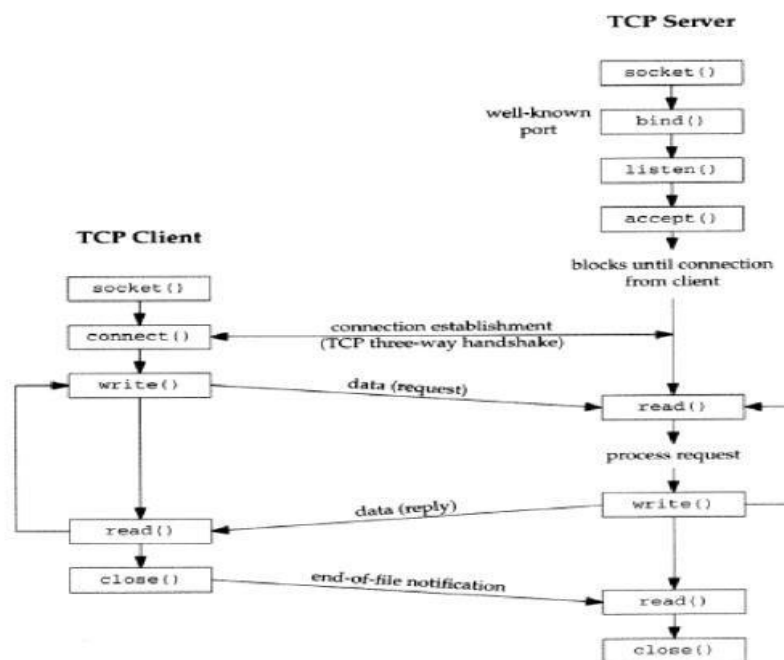
SOCKET PROGRAMMING

- ❑ Socket programming is a way of connecting two nodes on a network to communicate with each other.
- ❑ One socket (node) listens on a particular port at an IP, while the other socket reaches out to the other to form a connection.
- ❑ The server forms the listener socket while the client reaches out to the server. They are the real backbones behind web browsing. In simpler terms, there is a server and a client
- ❑ Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The socket library provides specific classes for handling the common transports as well as a generic interface for handling the rest.
- ❑ The term socket programming implies programmatically setting up sockets to be able to send and receive data.
- ❑ There are two types of communication protocols –
 - connection-oriented protocol
 - connection-less protocol
- ❑ TCP or Transmission Control Protocol is a connection-oriented protocol. The data is transmitted in packets by the server, and assembled in the same order of transmission by the receiver. Since the sockets at either end of the communication need to be set before starting, this method is more reliable.
- ❑ UDP or User Datagram Protocol is connectionless. The method is not reliable because the sockets does not require establishing any connection and termination process for transferring the data.
- ❑ Python The socket Module
 - This module includes Socket class. A socket object represents the pair of hostname and port number. The constructor method has the following signature –
- ❑ Parameters
 - family – AF_INET by default. Other values - AF_INET6 (eight groups of four hexadecimal digits), AF_UNIX, AF_CAN (Controller Area Network) or AF_RDS (Reliable Datagram Sockets).
 - socket_type – should be SOCK_STREAM (the default), SOCK_DGRAM, SOCK_RAW or perhaps one of the other SOCK_ constants.
 - protocol – number is usually zero and may be omitted.
- ❑ Return Type
- ❑ This method returns a socket object.
 - Once you have the socket object, then you can use the required methods to create your client or server program.

❑ A simple server-client program:

▪ Server:

- A server has a **bind() method** which binds it to a specific IP and port so that it can listen to incoming requests on that IP and port.
- A server has a **listen() method** which puts the server into listening mode. This allows the server to listen to incoming connections.
- And last a server has an **accept(), send () and close()** method. The accept method initiates a connection with the client and the close method closes the connection with the client.



▪ Client:

- Similar socket is set up on the client end. It mainly sends connection request to server socket listening at its IP address and port number
- The **connect() method** takes a two-item tuple object as argument. The two items are IP address and port number of the server.
- Once the connection is accepted by the server, both the socket objects can send and/or receive data.
- Through **send() method** server sends data to client by using the address it has intercepted.
- Client socket sends data to socket it has established connection with.
- **sendall() method**
 - similar to `send()`. However, unlike `send()`, this method continues to send data from bytes until either all data has been sent or an error occurs. None is returned on success.
- **sendto() method**
 - This method is to be used in case of UDP protocol only.
- **recv() method**

- This method is used to retrieve data sent to the client. In case of server, it uses the remote socket whose request has been accepted.
- **recvfrom() method**
- This method is used in case of UDP protocol.

TCP Server Client programming

❑ A simple server-client program:

▪ Server:

A server has a bind() method which binds it to a specific IP and port so that it can listen to incoming requests on that IP and port. A server has a listen() method which puts the server into listening mode. This allows the server to listen to incoming connections. And last a server has an accept() and close() method. The accept method initiates a connection with the client and the close method closes the connection with the client.

```
# first of all import the socket library
import socket

# next create a socket object
s = socket.socket()
print ("Socket successfully created")

# reserve a port on your computer in our
# case it is 12345 but it can be anything
port = 12345

# Next bind to the port
# we have not typed any ip in the ip field
# instead we have inputted an empty string
# this makes the server listen to requests
# coming from other computers on the network
s.bind(('', port))
print ("socket binded to %s" %(port))

# put the socket into listening mode
s.listen(5)
print ("socket is listening")

# a forever loop until we interrupt it or
# an error occurs
while True:

    # Establish connection with client.
    c, addr = s.accept()
    print ('Got connection from', addr )

    # send a thank you message to the client.
    c.send('Thank you for connecting'.encode())
```

```
# Close the connection with the client
c.close()
```

```
# Breaking once connection closed
Break
```

- First of all, we import socket which is necessary.
- Then we made a socket object and reserved a port on our pc.
- After that, we bound our server to the specified port. Passing an empty string means that the server can listen to incoming connections from other computers as well. If we would have passed 127.0.0.1 then it would have listened to only those calls made within the local computer.
- After that we put the server into listening mode. 5 here means that 5 connections are kept waiting if the server is busy and if a 6th socket tries to connect then the connection is refused.
- At last, we make a while loop and start to accept all incoming connections and close those connections after a thank you message to all connected sockets.

❑ Client Side

- **Now for the client-side:**

```
# Import socket module
import socket
```

```
# Create a socket object
s = socket.socket()
```

```
# Define the port on which you want to connect
port = 12345
```

```
# connect to the server on local computer
s.connect(('127.0.0.1', port))
```

```
# receive data from the server and decoding to get the string.
print (s.recv(1024).decode())
# close the connection
s.close()
```

- First of all, we make a socket object.
- Then we connect to localhost on port 12345 (the port on which our server runs) and lastly, we receive data from the server and close the connection.
- Now save this file as client.py and run it from the terminal after starting the server script.

❑ Output

start the server:

```
$ python server.py
Socket successfully created
socket binded to 12345
socket is listening
Got connection from ('127.0.0.1', 52617)
```

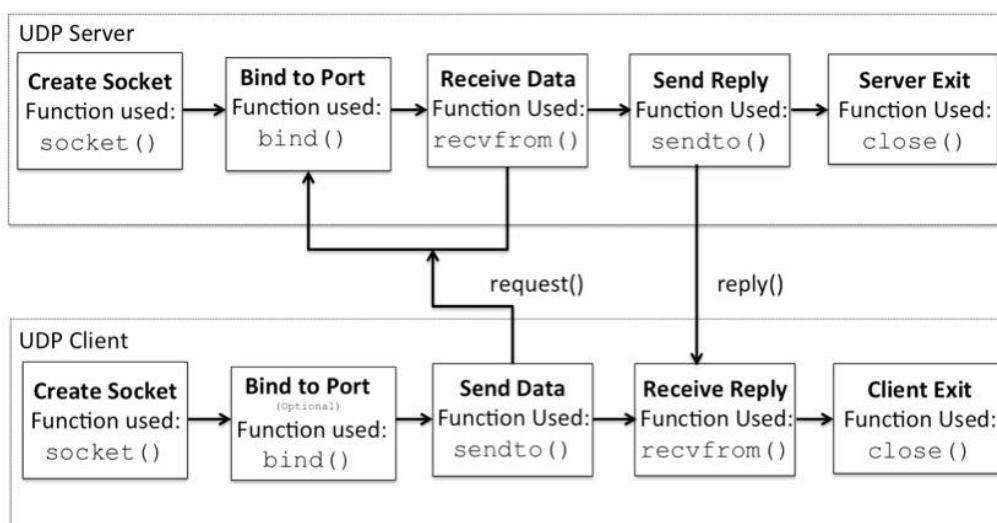
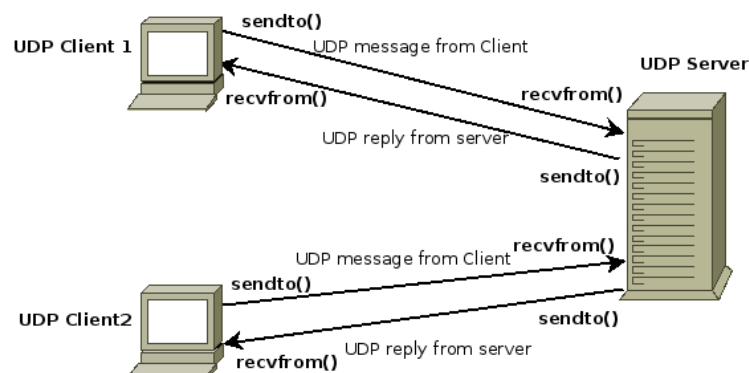
start the client:

```
$ python client.py
Thank you for connecting
```

UDP Server Client programming

❑ UDP Overview:

UDP is the abbreviation of User Datagram Protocol. UDP makes use of Internet Protocol of the TCP/IP suit. In communications using UDP, a client program sends a message packet to a destination server wherein the destination server also runs on UDP.




```
# Send to server using created UDP socket
UDPClientSocket.sendto(bytesToSend, serverAddressPort)

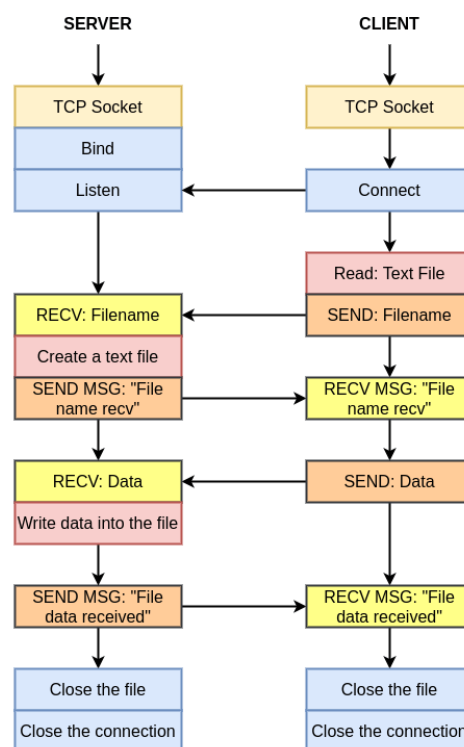
msgFromServer = UDPClientSocket.recvfrom(bufferSize)
msg = "Message from Server {}".format(msgFromServer[0])
print(msg)
```

❑ Output:

Message from Server "Hello UDP Client"

File Transfer using TCP Socket in Python

The overall procedure for the TCP file transfer is presented in the figure below.



❑ TCP-SERVER.py: Server-Side Implementation

We start by importing the socket library and making a simple socket. **AF_INET** refers to the address-family ipv4. The **SOCK_STREAM** means connection-oriented TCP protocol. After this, we bind the host address and port number to the server. The user enters the **maximum number of client connections required**. On the basis of the input, we determine the size of an array storing clients called **Connections**. The server then keeps listening for client connections and if found adds them to the Connections array. It does this until the Connections array reaches its maximum limit. Then, for every connection, processing starts,

- Data is received and then decoded.
- If data is not null, the server creates a file with the name "Output<a unique number>.txt" and writes it into it.
- This is continued until all data is written into the file

- The above happens for all clients and then connections are closed. Hence, we are able to receive a new file at the server's end with the exact same data.

```
import socket

if __name__ == '__main__':
    # Defining Socket
    host = '127.0.0.1'
    port = 8080
    totalclient = int(input('Enter number of clients: '))

    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind((host, port))
    sock.listen(totalclient)
    # Establishing Connections
    connections = []
    print('Initiating clients')
    for i in range(totalclient):
        conn = sock.accept()
        connections.append(conn)
        print('Connected with client', i+1)

    fileno = 0
    idx = 0
    for conn in connections:
        # Receiving File Data
        idx += 1
        data = conn[0].recv(1024).decode()

        if not data:
            continue
    # Creating a new file at server end and writing the data
    filename = 'output'+str(fileno)+''.txt'
    fileno = fileno+1
    fo = open(filename, "w")
    while data:
        if not data:
```



```

        break
    else:
        fo.write(data)
        data = conn[0].recv(1024).decode()

    print()
    print('Receiving file from client', idx)
    print()
    print('Received successfully! New filename is:', filename)
    fo.close()

# Closing all Connections
for conn in connections:
    conn[0].close()

```

❑ TCP-CLIENT.py: Client-Side Implementation

We first create a socket for a client and connect it to the server using the tuple containing the host address and port number. Then, the User enters the file name it wants to send to the server. After this, using the **open() function** of Python, the data of the file is read. Data is read line-by-line and then encoded into binary and sent to the server using **socket.send()**. When sending data is finished, the file is closed. It keeps on asking for a filename from the user until the client connection is terminated.

```

import socket
# Creating Client Socket
if __name__ == '__main__':
    host = '127.0.0.1'
    port = 8080

    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Connecting with Server
    sock.connect((host, port))

    while True:

        filename = input('Input filename you want to send: ')
        try:

```

```

# Reading file and sending data to server
fi = open(filename, "r")
data = fi.read()
if not data:
    break
while data:
    sock.send(str(data).encode())
    data = fi.read()
# File is closed after data is sent
fi.close()

except IOError:
    print('You entered an invalid filename!\
Please enter a valid name')

```

Chat Application using TCP Socket in Python

In this section, we'll explore the exciting world of socket programming by creating a simple chat server and client application in Python.

Socket programming allows two devices to communicate over a network using **sockets**, which act as endpoints for **sending** and **receiving** data. Our chat application will allow multiple clients to connect to the server and exchange messages in real-time.

Step 1: Setting up the Project

First, let's create a new directory for our project and navigate into it:

```

Mkdir chat_app
cd chat_app

```

Step 2: Implementing the Server

Create a new file called server.py and open it in your favourite text editor. We'll start by importing the necessary libraries:

```

import socket
import threading

```

Next, let's define the server code. We'll implement a function to handle each client's connection and communication:

```

def handle_client(client_socket):
    while True:
        data = client_socket.recv(1024)
        if not data:
            break
        message = data.decode('utf-8')
        print(f"Received message: {message}")
        response = "Server received your message: " + message

```

```
        client_socket.sendall(response.encode('utf-8'))
    client_socket.close()
```

In the `handle_client` function, we use a while loop to continuously receive data from the client. If the data received is empty, it means the client has disconnected, and we break out of the loop. Otherwise, we decode the received data and print it to the server's console. Then, we construct a response message and send it back to the client.

Now, let's create the main function to set up the server:

```
def main():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host = '127.0.0.1'
    port = 12345
    server_socket.bind((host, port))
    server_socket.listen(5)
    print(f"Server listening on {host}:{port}")

    while True:
        client_socket, client_address = server_socket.accept()
        print(f"Accepted connection from {client_address}")
        client_handler = threading.Thread(target=handle_client,
        args=(client_socket,))
        client_handler.start()

if __name__ == "__main__":
    main()
```

In the main function, we start by creating a socket object called `server_socket`. This socket acts as a communication endpoint for the server. The `socket` function from the `socket` module is used to create this socket object. We pass two arguments to this function:

1. `socket.AF_INET`: This argument specifies the address family for the socket, which, in this case, is the Internet Protocol version 4 (IPv4) addressing. It stands for "Address Family - Internet."
2. `socket.SOCK_STREAM`: This argument specifies the socket type, which is TCP (Transmission Control Protocol) in this case. TCP is a reliable, connection-oriented protocol used for data transmission. It stands for "Socket Type - Stream."

By combining these two arguments, we create a new socket object that will be used for TCP communication over IPv4. This socket will be the server's socket that listens for incoming connections from clients.

With this socket created and configured for TCP communication, the server can then proceed to bind it to a specific IP address (127.0.0.1) and port number (12345). After binding, the server is ready to listen for incoming connections from clients.

When a client connects, the server accepts the connection and creates a new thread to handle that client. The `handle_client` function is called in the new thread, allowing the server to handle multiple client connections concurrently.

Step 3: Implementing the Client

Next, create a new file called `client.py` and open it in your text editor. Similar to the server, start by importing the required library:

```
import socket
import threading
```

Next, implement the main function for the client:

```
def main():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host = '127.0.0.1'
    port = 12345
    client_socket.connect((host, port))

    while True:
        message = input("Enter your message: ")
        client_socket.sendall(message.encode('utf-8'))
        data = client_socket.recv(1024)
        response = data.decode('utf-8')
        print(f"Server response: {response}")

if __name__ == "__main__":
    main()
```

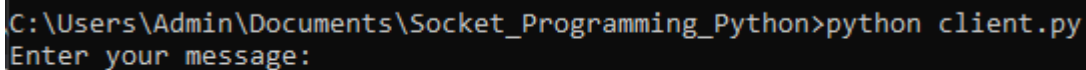
In the main function, we create a client socket and connect it to the server's IP address (127.0.0.1) and port number (12345). Then, we enter into a loop, allowing the user to input messages and send them to the server using the sendall method. We also receive the server's response and display it to the client.

Step 4: Testing the Chat Application

Now, let's test our chat application with some snapshots to see how it works.

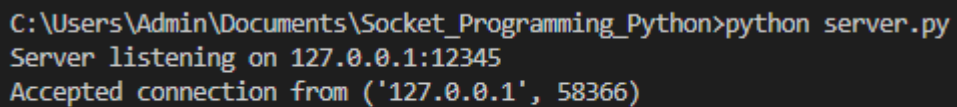
1. Start the server by running python server.py in the terminal:

Open a new terminal window and run the client script using python client.py:



```
C:\Users\Admin\Documents\Socket_Programming_Python>python client.py
Enter your message:
```

3. The server terminal will show a message indicating that it has accepted the client connection:



```
C:\Users\Admin\Documents\Socket_Programming_Python>python server.py
Server listening on 127.0.0.1:12345
Accepted connection from ('127.0.0.1', 58366)
```

4. To test with multiple clients, open more terminal windows and run client.py in each of them.
5. Once multiple clients are connected, the server terminal will display messages for each client connection:

```
C:\Users\Admin\Documents\Socket_Programming_Python>python server.py
Server listening on 127.0.0.1:12345
Accepted connection from ('127.0.0.1', 58366)
Accepted connection from ('127.0.0.1', 58374)
Accepted connection from ('127.0.0.1', 58385)
█
```

In the above examples, I have created 3 instances

6. Send a message from one of the clients:

```
C:\Users\Admin\Documents\Socket_Programming_Python>python client.py
Enter your message: Hi
Server response: Server received your message: Hi
Enter your message:
```

7. The server will receive the message and send a response back to the client.

```
C:\Users\Admin\Documents\Socket_Programming_Python>python server.py
Server listening on 127.0.0.1:12345
Accepted connection from ('127.0.0.1', 58402)
Received message: Hi
█
```

[NOTE —] The screenshots provided in the previous snapshot descriptions demonstrate how the terminal looks after performing the respective steps.

❑ Suggested further readings

- B. A. Forouzan – “Data Communications and Networking (5th Ed.) “– Chapter 25 – TMH.
- R. Nageswara Rao – “Core Python Programming” (2nd/3rd Edition) “– Chapter 23– Dreamtech Press.
- <https://www.geeksforgeeks.org/socket-programming-python/>
- https://www.tutorialspoint.com/python/python_socket_programming.htm
- <https://www.javatpoint.com/socket-programming-using-python>