" EMPOWERMENT THROUGH TECHNOLOGICAL EXCELLENCE "

# GENBA SOPANRAO MOZE COLLEGE OF ENGINEERING

Balewadi, Pune- 411 045.

Department of Electronics and Telecommunications

**Experiment No. –**

**Subject: - Mobile Computing**

**Name of the Student:_____Roll No._____**

**Date: _____ Marks &**

**Subject**

**Tittle:** File sharing by using TCP Protocol

**Problem Statement:**

To Perform File Transfer in Client & Server Using TCP/IP.

**Objectives:**

- What is a socket?
- The client-server model.
- Remote Communication

**Outcome:**

Develop Client-Server architectures and prototypes by the means of correct standards and technology.

**Software Requirements:**

Python, Open-source Linux operating system.

**THEORY:**

**The basics**

**What is mean by Socket**

Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else.

To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

**Types of Socket**

A Unix Socket is used in a client-server application framework. A server is a process that performs some functions on request from a client. Most of the application-level protocols like FTP, SMTP, and POP3 make use of sockets to establish connection between client and server and then for exchanging data.

**Socket Types**

There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used.

Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

**Stream Sockets** – Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order – "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.

**Datagram Sockets** – Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in Stream Sockets – you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).

**Raw Sockets** – These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.

**Sequenced Packet Sockets** – They are similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as a part of the Network Systems (NS) socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the Sequence Packet Protocol (SPP) or Internet Datagram Protocol (IDP) headers on a packet or a group of packets, either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets.

**The client-server model**

The client-server model is one of the most commonly used communication paradigms in networked systems. Clients normally communicate with one server at a time. From a server's perspective, at any point in time, it is not unusual for a server to be communicating with multiple clients. Client need to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established. The client and the server on the same local network (usually called LAN, Local Area Network), the client and the server may be in different LANs, with both LANs connected to a Wide Area Network (WAN) by means of *routers*

**Transmission Control Protocol (TCP)**

TCP provides a *connection oriented service*, since it is based on connections between clients and servers.

TCP provides reliability. When a TCP client sends data to the server, it requires an acknowledgement in return. If an acknowledgement is not received, TCP automatically retransmit the data and waits for a longer period of time for acknowledgement.

**TCP Socket API**

The sequence of function calls for the client and a server participating in a TCP connection is presented in following Figure

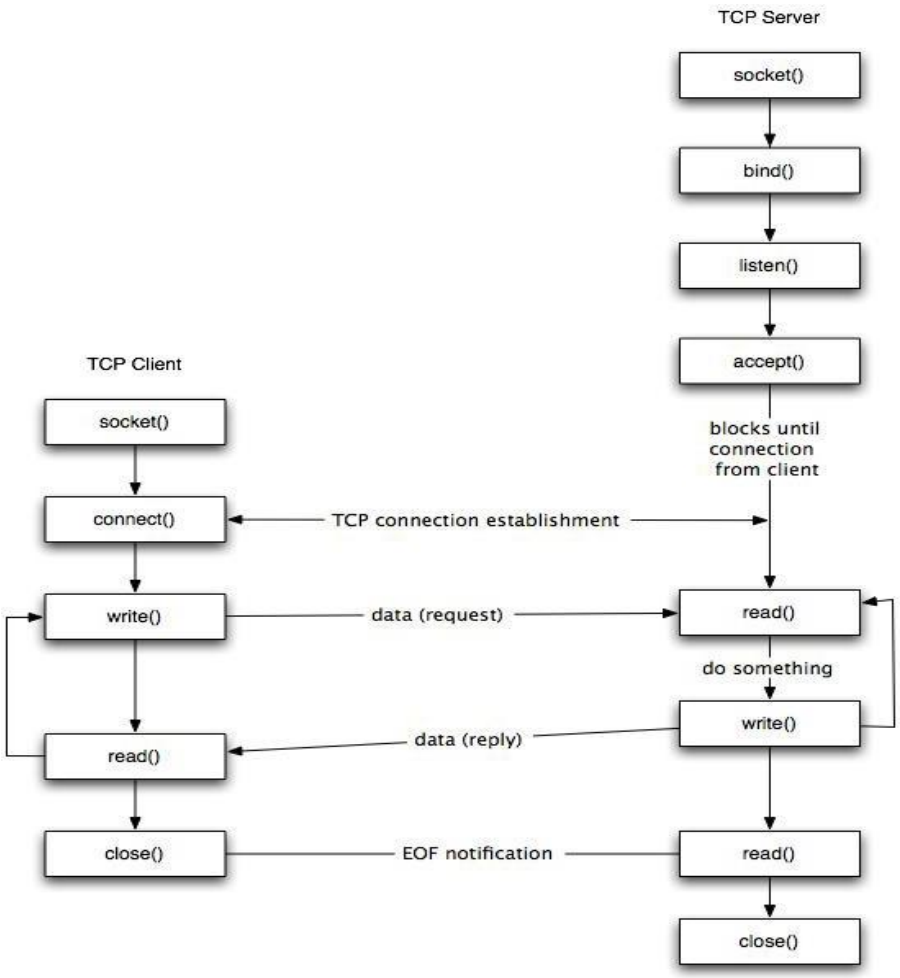



**Figure:** TCP client-server.

As shown in the figure, the steps for establishing a TCP socket on the client side are the following:

- Create a socket using the socket() function;
- Connect the socket to the address of the server using the connect() function;
- Send and receive data by means of the read() and write() functions.
- Close the connection by means of the close() function.
- The steps involved in establishing a TCP socket on the server side are as follows:
- Create a socket with the socket() function;
- Bind the socket to an address using the bind() function;
- Listen for connections with the listen() function;
- Accept a connection with the accept() function system call. This call typically blocks until a client connects with the server.
- Send and receive data by means of send() and receive().
- Close the connection by means of the close() function.

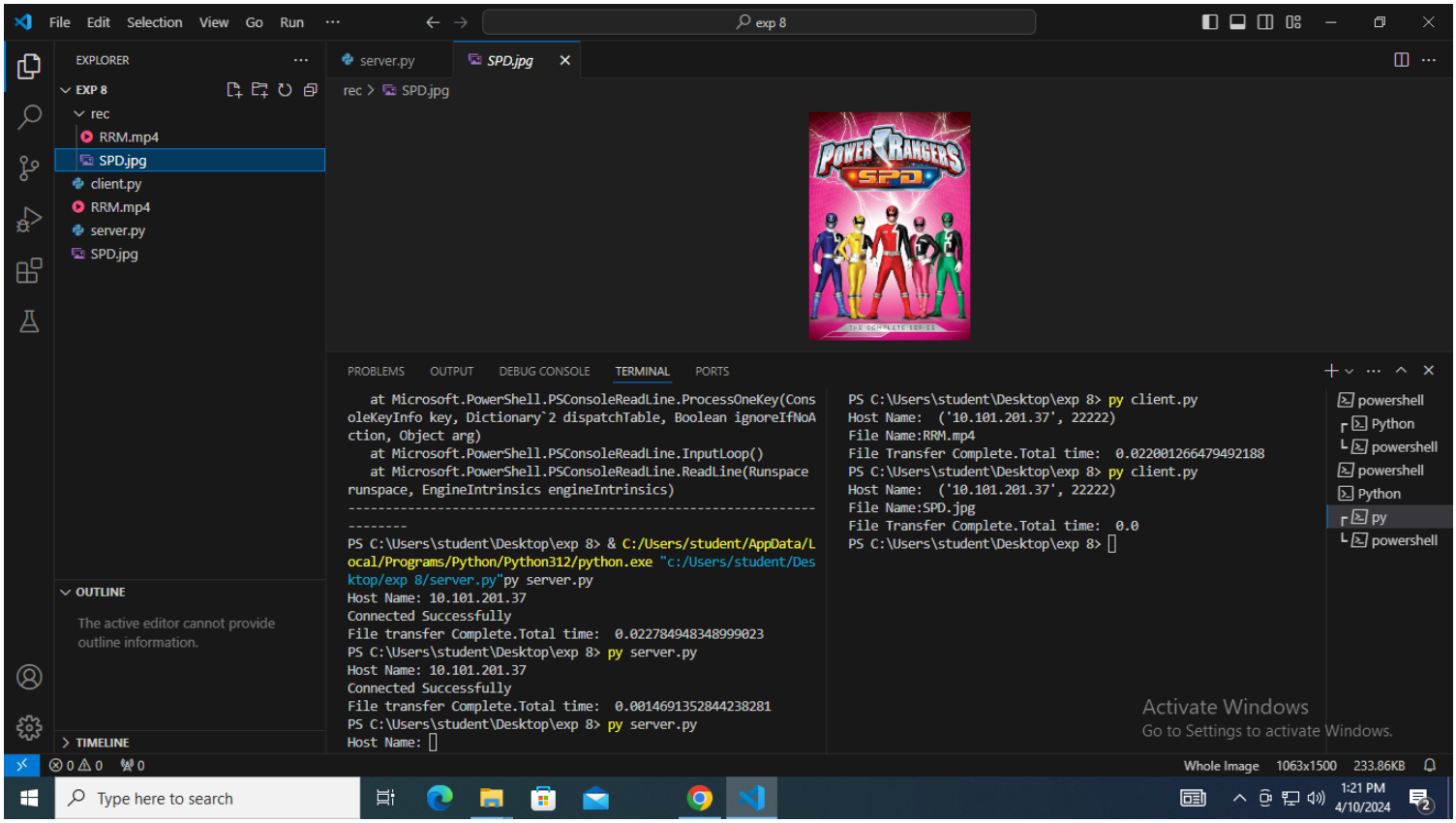
**Server code :-**

```
# This file is used for sending the file over socket
import os
import socket
import time
# Creating a socket.
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind((socket.gethostname(), 22222))
sock.listen(5)
print("Host Name: ", sock.getsockname())
# Accepting the connection.
client, addr = sock.accept()
# Getting file details.
file_name = input("File Name:")
file_size = os.path.getsize(file_name)
# Sending file_name and detail.
client.send(file_name.encode())
client.send(str(file_size).encode())
# Opening file and sending data.
with open(file_name, "rb") as file:
c = 0
# Starting the time capture.
start_time = time.time()
# Running loop while c != file_size.
while c <= file_size:
    data = file.read(1024)
    if not (data):
        break
    client.sendall(data)
    c += len(data)
# Ending the time capture.
end_time = time.time()
print("File Transfer Complete.Total time: ", end_time - start_time)
# Cloasing the socket.
sock.close()
```

## Client code :-

```
 # This file will be used for recieving files over socket connection.
import os
import socket
import time
host = input("Host Name: ")
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Trying to connect to socket.
try:
    sock.connect((host, 22222))
    print("Connected Successfully")
except:
    print("Unable to connect")
    exit(0)
# Send file details.
file_name = sock.recv(100).decode()
file_size = sock.recv(100).decode()
# Opening and reading file.
with open("./rec/" + file_name, "wb") as file:
    c = 0
    # Starting the time capture.
    start_time = time.time()
    # Running the loop while file is recieved.
    while c <= int(file_size):
        data = sock.recv(1024)
        if not (data):
            break
        file.write(data)
        c += len(data)
    # Ending the time capture.
    end_time = time.time()
print("File transfer Complete.Total time: ", end_time - start_time)
# Closing the socket.
sock.close()
```

## Result:




## Conclusion: