Experiment No. 7

Title: Socket Programming using Python

Batch: B-1 **Roll No: 16010422234** **Name: Chandana Ramesh Galgali**

Experiment No.:7

Aim: To demonstrate interprocess communication using socket programming in Python

---

Resources needed: Python IDE

Theory:

Inter process communication refers to two processes which will be communicating with each other. Most interprocess communication uses the *client server model*. One of the two processes in IPC, the *client*, connects to the other process, the *server*, typically to make a request for information. The client needs 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. Notice also that once a connection is established, both sides can send and receive information. The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a *socket*.
A socket is one end of an interprocess communication channel. Sockets are communication points on the same or different computers to exchange data. The two processes each establish their own socket.

The steps involved in establishing a socket on the *client* side are as follows:

1. Create a socket with the socket() system call
2. Connect the socket to the address of the server using the connect() system call
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.

The steps involved in establishing a socket on the *server* side are as follows:

1. Create a socket with the socket() system call
2. 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.
3. Listen for connections with the listen() system call
4. Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
5. Send and receive data

Socket Types

Two processes can communicate with each other only if their sockets are of the same type and in the same domain. There are two widely used socket types, *stream sockets*, and *datagram sockets*. Stream sockets treat communications as a continuous stream of characters, while datagram sockets have to read entire messages at once. Each uses its own

communications protocol. Stream sockets use TCP (Transmission Control Protocol), which is a reliable, stream oriented protocol, and datagram sockets use UDP (Unix Datagram Protocol), which is unreliable and message oriented.

Socket Programming in Python

Python provides a socket module for the implementation of socket objects. To create a socket use the socket.socket()

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

The arguments of socket() are:

- **socket_family:** Represents the address (and protocol) family. It can be either AF_UNIX or AF_INET. AF_INET refers to the address family ipv4. create an IPv6 socket by specifying the socket AF_INET6 argument.
- **socket_type:** Represents the socket type, and can be either SOCK_STREAM (means connection oriented TCP protocol)or SOCK_DGRAM(means connection oriented UDP protocol).
- **protocol:** This is an optional argument, and it usually defaults to 0.

Once we have created a socket object, use built-in methods of socket module given below to open a **connection**, **send** data, **receive** data, and finally **close** the connection.

socket.socket(): Create a new socket using the given address family, socket type and protocol number.

socket.bind(address): Bind the socket to **address**.

socket.listen(backlog): Listen for connections made to the socket. The **backlog** argument specifies the maximum number of queued connections and should be at least 0; the maximum value is system-dependent (usually 5), the minimum value is forced to 0.

socket.accept(): The return value is a pair **(conn, address)** where **conn** is a new socket object usable to send and receive data on the connection, and **address** is the address bound to the socket on the other end of the connection.
At **accept()**, a new socket is created that is distinct from the named socket. This new socket is used solely for communication with this particular client.
For TCP servers, the socket object used to receive connections is not the same socket used to perform subsequent communication with the client. In particular, the **accept()** system call returns

a new socket object that's actually used for the connection. This allows a server to manage connections from a large number of clients simultaneously.

socket.send(bytes[, flags]): Send data to the socket. The socket must be connected to a remote socket. Returns the number of **bytes** sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.

socket.close(): Mark the socket closed. all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected, but it is recommended to close() them explicitly.

Activities:

1. Write a python socket program to communicate with your adjacent machine (server) by passing a message. Server should read the message sent by client and response message should include the unique words of message to client and total length of the message. (Students are expected to sit in pair-neighboring PC's and should perform the communication)

Result: (script and output)

```
# echo-client.py

import socket

HOST = "172.17.17.56" # The server's hostname or IP address
PORT = 65432 # The port used by the server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
 s.connect((HOST, PORT))
 s.sendall(b"Hello, world")
 data = s.recv(1024)

print(f"Received {data!r}")
```

```
========== RESTART: C:/Users/EXAM/NEW.py ==========
Received b'Hello, world'
```

```
import socket

HOST = "172.17.17.56"  # Standard loopback interface address (localhost)
PORT = 65432  # Port to listen on (non-privileged ports are > 1023)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
```

```
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

Outcomes: Demonstrate handling database with python and to understand network programming with Python scapy.

Conclusion: (Conclusion to be based on the objectives and outcomes achieved)
the successful completion of the demonstration has showcased the power and versatility of socket programming in Python, highlighting its importance in various applications that require interprocess communication.

References:

1. https://docs.python.org/3.4/howto/sockets.html
2. Daniel Arbuckle, Learning Python Testing, Packt Publishing, 1st Edition, 2014
3. Wesly J Chun, Core Python Applications Programming, O'Reilly, 3rd Edition, 2015
4. Wes McKinney, Python for Data Analysis, O'Reilly, 1st Edition, 2017
5. Albert Lukaszewsk, MySQL for Python, Packt Publishing, 1st Edition, 2010
6. Eric Chou, Mastering Python Networking, Packt Publishing, 2nd Edition, 2017