Sanyam Agrawal SE21UCSE192 CSE3

Assignment 1: Introduction to Inter-Process Communication Using Sockets

Client2.py ->

Server2.py Client2.py

C: > Users > pc > Desktop > Dis_Sys > Lab1 > Client2.py > ...

```python
import socket

def start_client():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Friend's IP address
    host = '10.70.47.95'
    port = 9999

    # Connect to the server
    client_socket.connect((host, port))

    # Receive the message from the server
    message = client_socket.recv(1024)
    print(message.decode('ascii'))

    # Send "Hi" to the server
    client_socket.send("Hi".encode('ascii'))

    # Close the connection
    client_socket.close()

if __name__ == "__main__":
    start_client()

```

Server2.py ->

Server2.py | Client2.py

C: > Users > pc > Desktop > Dis_Sys > Lab1 > Server2.py > start_server

```python
import socket
def start_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Get local machine name
    host = socket.gethostname()
    port = 9999

    # Bind to the port
    server_socket.bind((host, port))

    # Queue up to 5 requests
    server_socket.listen(5)
    print(f"Server started! Listening on {host}:{port}")

    while True:
        # Establish a connection
        client_socket, addr = server_socket.accept()
        print(f"Got a connection from {addr}")

        # Send a thank you message to the client
        message = 'Message received. Thank you for connecting' + "\r\n"
        client_socket.send(message.encode('ascii'))

        # Close the connection
        client_socket.close()

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

Report: Design and Implementation of Client-Server Program Using Sockets

Design Overview

The program is designed to establish a simple client-server communication using Python's socket library. The server listens for incoming connections, accepts them, and sends a confirmation message back to the client. The client connects to the server, receives the message, sends a "Hi" message back, and then closes the connection.

Server Implementation

1. **Socket Creation:** The server program begins by creating a socket using socket.AF_INET (IPv4) and socket.SOCK_STREAM (TCP), which ensures a reliable, connection-based communication.
2. **Binding:** The server binds the socket to a specific host and port (9999). The host is retrieved using socket.gethostname(), which returns the hostname of the machine running the server.
3. **Listening:** The server then listens for incoming connections with a queue of up to 5 requests using server_socket.listen(5).
4. **Accepting Connections:** When a client attempts to connect, the server accepts the connection using server_socket.accept(), which returns a new socket for the connection and the address of the client.
5. **Communication:** The server sends a message ("Message received. Thank you for connecting") to the client.
6. **Connection Closure:** After sending the message, the server closes the connection to the client using client_socket.close(). The server remains in an infinite loop, continuously accepting and handling new connections.

Client Implementation

1. **Socket Creation:** Similar to the server, the client starts by creating a socket using socket.AF_INET and socket.SOCK_STREAM.
2. **Connecting to Server:** The client connects to the server using the server's IP address (host) and port (9999) via client_socket.connect((host, port)).
3. **Receiving Data:** The client receives the server's message using client_socket.recv(1024).
4. **Sending Response:** The client sends a "Hi" message back to the server.
5. **Connection Closure:** Finally, the client closes the connection.

Challenges and Solutions

1. **Networking Issues:** One common challenge in socket programming is ensuring that the correct IP address is used, especially when working in a networked environment with multiple devices. The solution is to use socket.gethostname() for local testing or manually input the correct IP address of the server when running the client.
2. **Port Conflicts:** If the chosen port (9999) is already in use by another application, binding the socket would fail. To overcome this, a different port number can be chosen, or the conflicting application can be stopped.
3. **Message Encoding/Decoding:** When sending and receiving messages, ensuring consistent encoding (e.g., ascii) is important to prevent issues with character interpretation. The encode('ascii') and decode('ascii') methods ensure that the data is properly handled across the connection.
4. **Connection Handling:** Properly managing the connections is crucial, especially closing the sockets after communication to avoid resource leaks. The program handles this by explicitly closing both client and server sockets after the communication is complete.
5. **Infinite Loop on Server:** The server runs an infinite loop to continuously accept client connections, which is typical in server design. However, this requires careful handling to ensure that the server can still be shut down properly when needed.