

ALGORITHMS AND OPERATING SYSTEMS

Project Report

Implementation of Client-Server Model

Namratha Gopalabhatla - 20171017

Varun Balaji - 20171202

Sagrika Nagar - 20171204

Introduction

The client-server model is an application structure that partitions tasks between the providers of the resources or services, called servers and service requesters called clients. In the client-server architecture, when the client computer sends a request for data to the server through the internet, the server accepts the requested process and delivers the data packets requested back to the client.

In this project, we aim to understand the concepts of multithreading and sockets by creating a chat application which is based on the client-server model.

Chat Application

Some of the features of the chat application include:

- Multiple clients can connect to the server.
- Clients can post messages on the GUI.
- Messages posted by each client are shown to every other client connected to the server.

Threads and Multithreading

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its

current working variables, and a stack which contains the execution history. Threads provide a way to improve application performance through parallelism. The CPU switches rapidly back and forth among the threads giving the illusion that the threads are running in parallel.

Multithreading is similar to multitasking, but enables the processing of multiple threads at one time, rather than multiple processes. Since threads are smaller, more basic instructions than processes, multithreading may occur within processes.

By incorporating multithreading, programs can perform multiple operations at once.

Sockets

A network socket is a software structure within a network node of a computer network that serves as an endpoint for sending and receiving data across the network. The structure and properties of a socket are defined by an application programming interface (API) for the networking architecture. Sockets are created only during the lifetime of a process of an application running in the node.

Because of the standardization of the TCP/IP protocols in the development of the Internet, the term network socket is most commonly used in the context of the Internet Protocol Suite, and is therefore often also referred to as Internet socket. In this context, a socket is externally identified to other hosts by its socket address, which is the triad of transport protocol, IP address, and port number.

Implementation

For this application, we wrote two scripts: server script and client script. The server receives multiple connection requests from the clients which can be implemented through multithreading and sockets.

Server Script

- Since our application requires a connection to be established before any communication, we used TCP sockets (AF_INET and SOCK_STREAM flags).
- Then, we defined some constants (host, port etc) so that clients can connect to our server.

-
- Server tasks can be divided into three main functions:
 - Accepting new connections
 - Handling particular clients
 - Broadcasting messages
 - **Accepting connections:** This task is done by the function `accept_clients()`. This function contains a loop that waits forever for incoming connections and as soon as it gets one, it logs the connection (prints some of the connection details) and sends the connected client a welcome message and asks for the client's name. Then it stores the client's address in the addresses dictionary and later starts the handling thread for that client. The target function for the thread - `handle_client` is described later.
 - **Handling Clients:** In the `handle_client()` function, the first task we do is to save the name provided by the client and then send another message to the client giving further instructions (*'Hi there, %s! If you ever want to leave the chat, type QUIT or press the Quit button.'*). After this, if we receive further messages from the client and if a message doesn't contain instructions to quit, we simply broadcast the message to other connected clients using the broadcast function which is described after this. If we do encounter a message with exit instructions (i.e., the client sends a {quit}), we echo back the same message to the client (it triggers close action on the client side) and then we close the connection socket for it. We then do some cleanup by deleting the entry for the client, and finally broadcast to other connected clients that this particular client has left the conversation.
 - **Broadcasting messages:** This is defined by the function `broadcast_message()`. it simply sends messages to all the connected clients, and prepends an optional prefix if necessary. We do pass a prefix to `broadcast()` in the `handle_client()` function so that people can see exactly who is the sender of a particular message.
 - The main challenge faced here was connecting multiple clients to the server. This also includes them being able to send/receive messages at the same time. For that we used multithreading. In the main function, we start multiple threads as the clients join the server using `accept_clients()` as the target function. It initiates the infinite loop that waits for clients to connect. The `accept_thread.join()` line waits for all the clients to exit the server and then close the server.

Client Script

- In the client script we include the Python module Tkinter which is used for building the messaging application.
- Since our application requires a connection to be established before any communication, we used TCP sockets (AF_INET and SOCK_STREAM flags).
- We declare a few constants such as the host name, port number and the buffer size to connect the client to the server.
- The client has three main functions it needs to perform:
 - Receive messages
 - Send messages
 - Creating the chatbox
- We have three other small functions which are available just to enhance the functionality of the chat application and are dependent on the three main functions mentioned above
 - Closing the chat window
 - Creating the smile emoji
 - Creating the frown emoji
- **Receive messages:** This task is done by *recieve_message()*. We use an infinite loop here because the client will be receiving messages non-deterministically and is not dependent on when messages are sent. The message is received using *sock.recv()*, and the loop stops the execution until the message is received. The received message is appended to the list of messages, so that they can be viewed on the application.
- **Sending messages:** This task is done by *send_message()*. The message of the client is taken as input in the application, and we set that as the message that needs to be sent, say, *msg*. The message of the client is set to an empty string so that the next message can be taken as input and so on. The message, *msg*, can be a regular message or can be the message "QUIT". If the message of the client, that is *msg* is a regular message then it is sent using *sock.send()*. If not, then the socket connection is closed and the chat window for the client is destroyed.

-
- **Creating the chat window:** This task is done by *create_chatbox()*. We use Tkinter for building the GUI, so we first create the application window and give it a title. Then we create the message frame with a scrollbar, where we can see messages from various clients present in the chatroom. The chatbox starts out with a welcome message and prompts the client to enter his/her name and then allows the person to continue chatting with the other clients. We have 1 entry field and 4 different buttons on the GUI:
 - We create an entry field where the person can enter his message which he wants to send and press “enter” on his keyboard to send the message
 - The first button is the “Send Message” button which allows the person to send the message he enters in the entry field when the *send_message()* function is called within the button function.
 - The second button is the “Smile” button which sends the smile emoji as the message when the *smileEmoji_button()* function is called within the button function (discussed below).
 - The third button is the “Frown” button which sends the frowning emoji as the message when the *frownEmoji_button()* function is called within the button function (discussed below).
 - The fourth button is the “Leave Chat” button which closes the socket connection of the client to the server and destroys the chat window of the client when the *close_window()* function is called within the button function (discussed below).

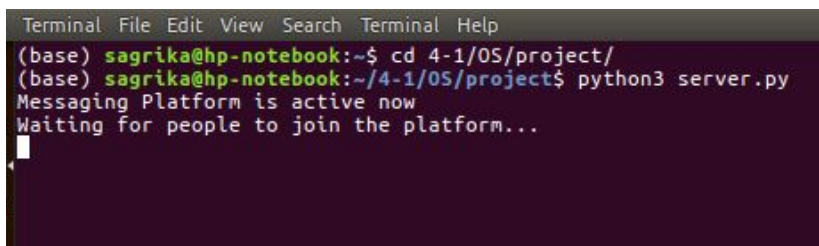
We use the Tkinter protocol to ensure that the socket connection between the server and the client is closed when the chat window is closed by the client, and this is done by calling the *close_window()* function within the protocol.

We have declared the HOST name and the PORT number at the beginning of the program, and we use this to define the address of the server. We use the TCP socket to create the client and then connect the client to the server using the address. Once this connection is established, the thread for receiving messages is started, which uses the *receive_messages()* function, and then the mainloop of the application GUI starts as well.

-
- **Closing the chat window:** This task is done by the function `close_window()`. Here the message of the client is set as "QUIT", and then the function `send_message()` is called to complete the functionality.
 - **Creating the smile emoji:** This task is done by the function `smileEmoji_button()`. Here the message of the client is set as "=", and then the function `send_message()` is called to complete the functionality.
 - **Creating the frown emoji:** This task is done by the function `frownEmoji_button()`. Here the message of the client is set as "=", and then the function `send_message()` is called to complete the functionality.
 - One of the challenges faced while building the client-server model was to improve the usability and convenience of using the chatting application by connecting various clients. We have involved The GUI for exactly this reason, to improve the usage of the application and to increase the aesthetic value of the application.
 - Another challenge faced in the client script was to build the GUI which we could integrate with the sockets that are created. We had to make sure that the functions like `send_message()` and `receive_message()` can be integrated within the button functionalities and the message frame. We had to make sure that none of the messages were lost and could be viewed by the participating client on his chat window. We also have to make sure that the buttons perform the function they need to.

Execution

1. Run "`python3 server.py`" on your terminal to create the server.

A screenshot of a terminal window with a dark background and light-colored text. The terminal shows the command `python3 server.py` being executed. The output indicates that the messaging platform is active and waiting for people to join. The terminal window has a menu bar at the top with options: Terminal, File, Edit, View, Search, Terminal, and Help. The prompt shows the user is in a directory `~/4-1/05/project/`.

```
Terminal File Edit View Search Terminal Help
(base) sagrika@hp-notebook:~$ cd 4-1/05/project/
(base) sagrika@hp-notebook:~/4-1/05/project$ python3 server.py
Messaging Platform is active now
Waiting for people to join the platform...
█
```

2. Then, run "*python3 client.py*" on another window to connect a client to the server.
You can connect multiple clients by running the above command on separate command windows.

```
@hp-notebook: ~/4-1/OS/project
(base) sagrika@hp-notebook:~/4-1/OS/project$ python3 client.py
```

3. After the connection is established, the chat window pops up. Enter your name and you can start chatting on the window.



The blocked text in the image is the address of the client.



4. Press the button "quit" if you want to exit. A message is sent to other clients in the server that you have left the chat.

