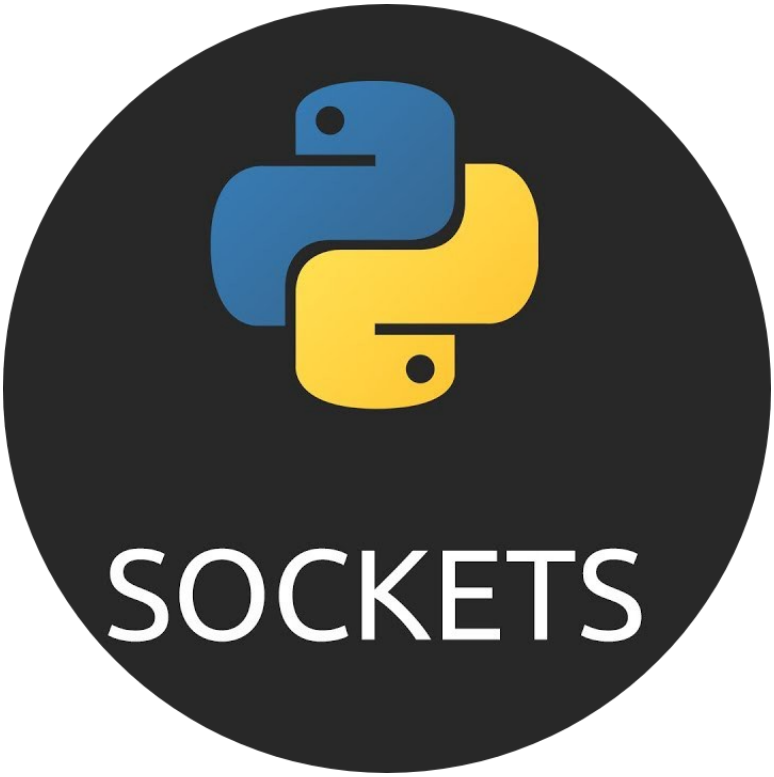
Socket Programming



Lab 2 Code Report

Roshan Poudel | Professor Jihoon Ryoo

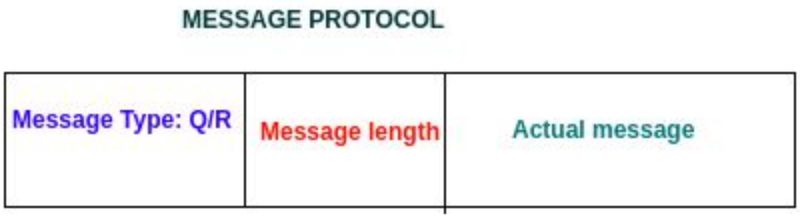
October 19, 2020

roshan.poudel@stonybrook.edu

Implementation & Approach:

The TCP Addressbook client-server model is implemented using sockets in Python. ***server.py*** contains code for the server and ***client.py*** contains code for the client. Both the client and the server follow the general approach of first creating a socket using the IP Address and Port number then using this socket to send and receive messages.

To receive and send messages, a custom messaging protocol is used. The protocol requires each request to have a request type of either ***"Q" or "R"***, where Q stands for Query type and R stands for response type. This message type is packed and unpacked using struct. Similarly, the header contains information about the length of the message to be received. After the request type and message length, the message of pre-specified length is sent. If any client or the server doesn't follow this messaging protocol, the message is discarded.



How to Run:

1. In ***server.py*** change the ***HOST*** and ***PORT*** values to server IP and port number and copy the same values over to **client.py**
2. Run ***server.py*** and don't close it until you need to shut down the server
3. Run ***client.py*** in a different terminal and enter the email address you want to search in the Address book
4. You shall receive a message from the server about your input
5. If you want to check another email, enter ***y***
6. Enter the next email address to check and continue with the same flow.

Implementation & Flow in server.py

The server needs a Public IP Address that it uses to connect to the internet. Our server is hosted in a digital ocean droplet with Ubuntu and is available at ***68.183.131.122*** . Also, port number ***8080*** is considered generally free and is used for the purpose of creating sockets. The server will always listen for incoming connections at the given IP Address and Port number. Any client connecting to the server must know the IP and the Port address of the server for communication. (This public IP may be replaced by localhost while testing in local machine)

The first step in the implementation of server.py is the creation of a socket and the binding of the socket with the server IP Address and port number. The following code creates a TCP socket that can be used over IPV4:
SOCKET = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

The following code binds the IP Address and port number of our server with the socket:
SOCKET.bind((HOST, PORT))

Once the socket is created and the Ip/Port are bound with the socket, our server can listen for connection requests. In the ***handle_connections( )*** method of server.py, we keep on listening for connection requests in the socket, and as soon as a request is received, we accept and handle the request in a new thread. **Using multithreading enables our server to handle multiple requests simultaneously.**

Once the ***handle_connections( )*** method sends the request to ***handle_client(conn, addr)*** method for request handling, we unpack the request header and look for request type and the message length. If the header of the request sent to the server doesn't consist of request type and message length, it doesn't follow the protocol and thus the server rejects the message request and sends an error message indicating that the protocol used for message sending was not understood.

If the request followed the messaging protocol, the request type was ***Q*** and the message length was also specified. Now, we keep on receiving messages of **1 byte** each until the length doesn't reach the specified value. This ensures that we receive all of the messages sent by the client. These small bytes of messages are decoded and combined together to form the original message.

Once the message(email) is received, we check for the given email address in the ***addrbook*** dictionary. If the email doesn't exist, we return a response message of type ***R*** with the message that the email doesn't exist in the address book. The ***send_message(conn, message)*** method handles sending messages to the clients. After the message is sent, the connection is terminated.

Implementation & Flow in client.py

Each client must already know the IP Address and the port number where the server.py is listening for a connection. In our case, it's at ***68.183.131.122*** in port number ***8080.*** Each client first creates a TCP IPV4 socket and connect it to our server's IP and port number:

SOCKET = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
SOCKET.connect((HOST, PORT))

Now that the socket is created and connected, the client can send and receive messages. To send messages, the earlier described messaging protocol is used. First, an email from the terminal is read and a header of query type, ***Q*** is created along with the length of the email. This is sent along with the encoded email to the server.

The server after reading the request replies accordingly. If the email that we were looking for was found, the server returns the name of the email owner. If the email is not found, the server responds back with an error message. We check for the message received from the server using ***receive_response(email)*** method and display it accordingly to the user.

To enable the user to check for multiple emails, we call the main method multiple times and create and use sockets to receive and send messages to and from the server.

Overall Programming Flow

First of all, the server should create a socket, bind it to its public IP and port, and start listening for incoming client connections. This means ***server.py*** should be running even before running ***client.py***. If we try to run ***server.py*** before running ***client.py***, we get a connection error.

Once ***server.py*** is listening for connections, the client sends a message following the prespecified messaging protocol. The ***server.py*** checks for the given email contained in the request and responds back accordingly.