ECE 531 Graduate Final Project

Team Members:
Bhautik (Brian) Amin
Yimai Zhou
Yizhou Zhou

I. INTRODUCTION

The main objective of this project is to design and implement a server-client architecture consisting of a defined list of features and operations; and to learn how communication happens between nodes of a network on the application and transport layers. To achieve this, a TCP server-client network will be built using socket programming using the Python (Python 2.7) programming language

The requirements of the proposed architecture are stated as followed:

- Client is able to send a string to a server, server sends a reply back with the result of a check if that word is a valid palindrome
- The client will be able to send a list of words, and receive a list of results. Client will be able to discern the results through the mapping of order
- Server will keep a running record of all palindromes found. Client will be able to disconnect and reconnect and still be able to recall the same running list from the server
- Client will be able to ask for the number of palindromes found, the list of palindromes, and the latest palindrome added to the list
- Client will be able to ask the server to delete all palindromes from the list, as well as delete palindromes in specific positions in the list
- Client is able to terminate the connection with the server. Or request for a non-persistent connection (Connect persistent by default)

II. APPROACH

Part 1 TCP Network

Using the Python programming language, TCP based server (server.py) and client (play_client.py) scripts were written using the socket and threading libraries. To achieve multi-client functionality; it was determined that the TCP network system will need to employ a multi-threaded server; so that when clients request connections, the server can handle their requests and provide responses all in parallel so the client's demands will be handled simultaneously [1].

For the server script, a socket was created and binded to an arbitrary unused port number (23456), a list of used port numbers was used to corroborate the actual availability of the port, this was done to ensure that no possible network conflicts would happen during the evaluation of the program [2]. In Python, a TCP socket is represented as an object, whose constructor is

the socket() function, in which parameters can be added in to specify the type of transport layer protocol one desires, as well as other features. The following creation of the socket is described here:

```
server_socket = socket(AF_INET, SOCK_STREAM)
```

Similar to the creation of a socket object in other programming languages, the constructor requires the first argument to be a domain, and the second argument to be a type [3]. AF_INET is used to specify that the socket will refer to addresses from the AF_INET family (IPv4 addresses in this specific case) [3]. SOCK_STREAM is used to specify that the connection will be running on top of the TCP transport layer protocol [3].

```
server_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
```

In this next case, socket options are being set. SOL_SOCKET specifies that the options being set are socket-level customizations [3], and SO_REUSEADDR allows the socket to bind and reuse a local address that may have already been binded before. This option was configured for testing purposes, as during prototyping and testing safe exit (closing the socket and unbinding) was not implemented or may not trigger correctly and there will be possible cases where a server won't get the chance to possibly close at the end of a session. The last value within this option is a buffer [3].

From here the server's main process will run in a continuous loop where it will listen for clients requesting for a connection. When a connection is requested, the Python socket library will generate a new connection object which can be used to read and write to the TCP transport buffer. This connection (as well as the connecting client's address provided by the server connection accept method) is sent to a callback function, which will spin into a new thread where it will handle the client's requests and provide responses according to the given requirements stated in the introduction.

Safe exit was implemented in the final version of the program. When the user wants to stop the server, it will unbind from the socket and close all the threads running (safe exit).

Part 2 Protocol Design

The overlying system architecture describing the protocol is defined by the following figure (Figure 1):

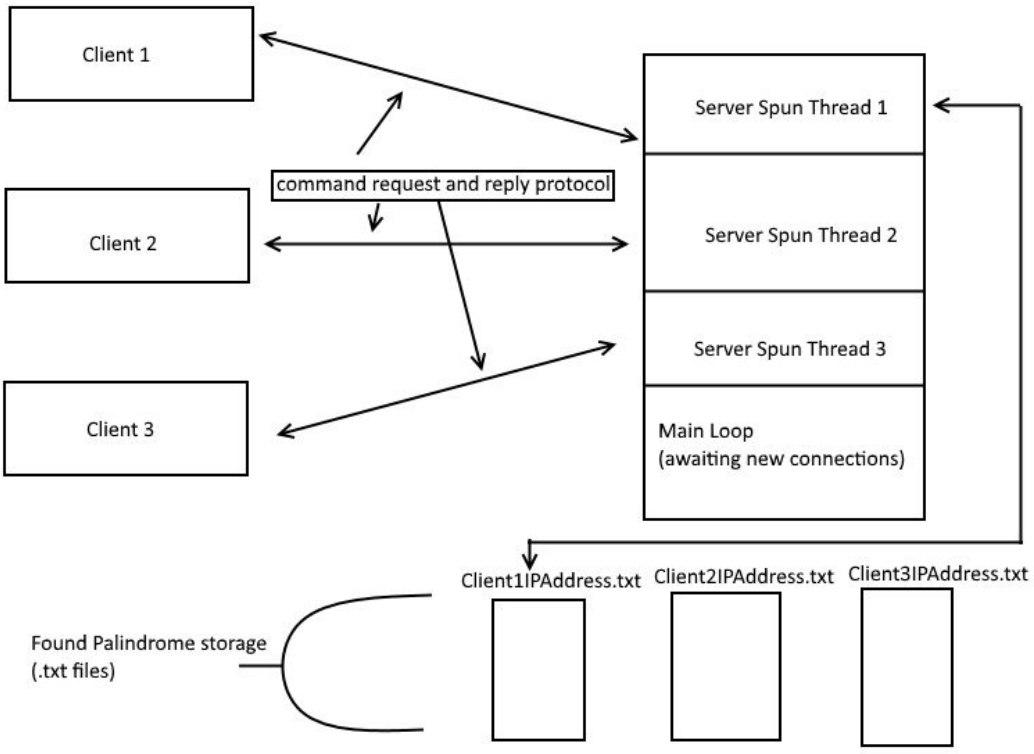

Figure 1. System Protocol Hierarchy

In the protocol, a client is given a list of commands (one command for every feature dictated in the introduction section). This command is sent to the server formatted as a comma delimited string with the following structure:

"Command, Value1, Value2, Value3…"

The commands themselves are simple number values ("1", "2" , "3", etc. formatted and evaluated as strings, not integers). The following data after the command (Value1, Value 2, etc.) is relevant data needed for the server to perform the desired operations (In this case a list of words, possibly single word, specific positions to delete, etc.). On the server side, the command is removed from this list (The command is known by the server to be the first index of this comma delimited string list), leaving the relevant data to be evaluated. When a server sends a response (For example, the results of a list of words) the user will understand the mapping of the results based on the order of how the data was first inputted in the client program.

This command packet structure was chosen primarily for two reasons. The first being that the Python socket library requires strings to be sent [4], and that being able to leverage Python's built in data structures would allow for this project to be simple and achieve the desired requirements with ease and flexibility. The data structure used in specifically was a python List. Being able to have payloads configured as comma delimited strings allows for easy conversion to a python list and back to a comma delimited string (For a possible reply by the server). When the comma delimited string is converted to a Python list, array operations are easy methods included in the Python standard library [5]. Methods such as pop() to remove values at certain positions, or append() to add them at the end of the list [5]. Another feature

that is achievable because of the use of Python lists and comma delimited strings is the ability to write and recall data as a text file.

To dive into more detail regarding the commands and the command packet structure itself, the following table is generated:

Client Action	**Client Command Message Format (Comma delimited string)**	**Server Reply Message**
Send a word to check if it is a palindrome	“1, word”	“result” (A string boolean)
Send multiple words to check their results	“2, word1, word2, word3”	“result1, result2, result3” (Client application will ask user to present words as “word1, word2,word3” so the mapping is based on order generated by the user
Exit and close the connection	“3”	“Bye” client will disconnect from the server and close its connection. Server will remain running but will close that corresponding thread in which was handling the client request/responses
Return total number of palindromes found	“4”	“Amount of palindromes found: " + str(len(found_palindromes))” Server will reply back with a string stating how many palindromes have been found, following with the amount of found palindromes that have been stored server-side
Return list of palindromes found	“5”	“word1,word2,word3” Server will reply back with a list of words from the found palindromes python list (Converted to a comma delimited string).
Return latest palindrome found	“6”	"Latest palindrome: " + latest Server will return the latest palindrome found. “latest” is a variable in which the most recent entry is copied out from the found palindrome python list
Delete all found palindromes from the list	“7”	"Cleared the palindrome found list" Server will empty the running python found palindromes list and send a simple string reply to the client

Delete palindromes at specific positions	“8, pos1,pos2,pos3” where ‘pos’ is some integer formatted string position value	“word1,word2,word3” Server will reply back with an updated list of the found palindrome structure, in which the positions given by the user were deleted
Request non-persistent connection	Handled client side: Boolean is triggered such that at the end of the next command it will send “3” to the server requesting to close the connection, client will close its connection as well and prompt user if they want to restart the program/reconnec t	“Bye”, see command 3 above

Table 1. Command List with Detailed Descriptions

On the client side, the program will show a list of these commands to the user. It will then prompt the user to input whichever number corresponds to the value they want:

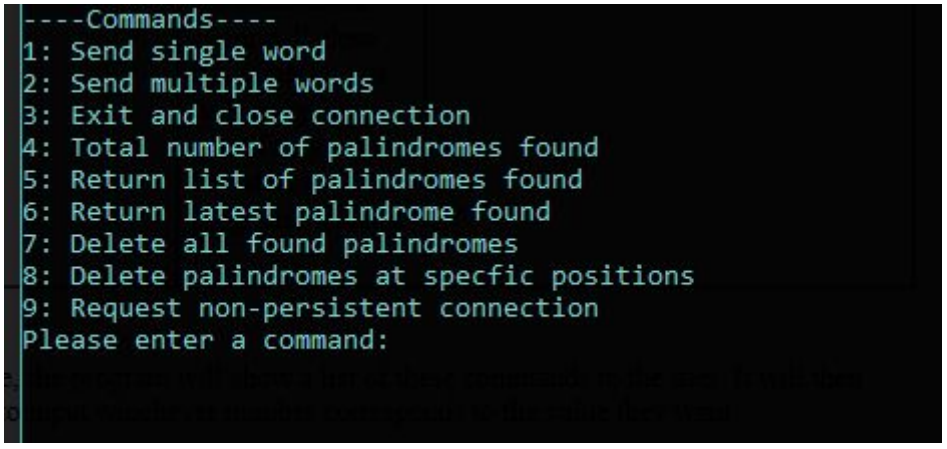

Figure 2. Command list

User input verification is added in to ensure that the program will not crash or freeze if the user inputs some command that isn’t specified in the list above. The program will simply tell the user to reinsert a valid command shown on this list and start again. At the end of every command transaction (Client sends request, server sends reply, client display reply) this menu will show again to let the user know what commands are available. Commands are handled in an if-statement tree in which the command is parsed and the corresponding action is then prompted (Such as prompt user for input, etc.). Regarding the commands, furthur user input

verification protocols were used to ensure valid data was being sent to the server. For example, in command “8”, where the user asks the server to remove palindromes at specific positions, the client script will check the user input to ensure that it is a valid integer, and then it will cast that integer to a string so it can be sent to the server. For python lists, valid positions would be index 0 and above - so the script will check to see if the given position input is valid within that range and then it will add that to a list of positions that will be sent to the server. If such a input is not valid, it will trigger an exception and will notify the user to try again and insert a valid input.

User input verification also happens on the server end as well. For example, if the given positions prompted by the user are not in range, it will simply discard such values. Also, if the found palindrome list was emptied before - it will simply pass over the deletion loop (A simple for loop that will go through a sorted list of the user specified positions) and will show the user that the palindrome list is empty. When a palindrome list is empty, commands such as “5” or “6” in which user asks the server to return the amount of palindromes or the words in the list themselves will reply back that the list is empty as well. This sort of self-verification allows for the program to be robust and not throw errors regarding that the list might possibly be empty.

For the data structure chosen to handle the user found palindrome data, a Python list was chosen as mentioned earlier. When a client disconnects from the server (Command “3” or “9” for example), before the server closes the thread, it will write the found palindrome list data into a .txt file. A .txt file was chosen as writing and reading a Python list is simple and can be achieved by the built in file writing and reading functions within Python 2.7. When a server accepts a connection with the client, the IP address of the client is available for the server handling thread. This IP address is used to generate a filename for the database .txt file. Which is saved locally in the directory the server script is running from. This allows for when clients to reconnect to the server for the server thread to easily find the corresponding user data based on their host’s IP address, and load that back into the program as a python list where it can be manipulated.

Even though this method is simple and it works in the implementation, there are some factors that may break the protocol that should be noted. For example, the IP addresses of the hosts (server and client) are not static. So if the IP addresses change, the client files will not be accessible, and new files will be created based on the new IP address. Also, on the client-side, when it wishes to connect to the server the IP address of the server is hard coded in. So the user will have to know the new IP address of the server to connect, and the server would have to understand more than just the client’s IP to successfully operate with changing addresses. There are some ways to fix this however that has been investigated, such as setting the IP addresses of the host machine statically through configuring a local router’s firmware settings (Using the MAC address of a device and having the router automatically assign some static IP address to that device itself for example). Another method that could be developed is that during the initial connection the client could share some uniquely generated phrase that would serve as the filename for the server database text file. The implementation of this project does not have this feature however due to time constraints. A security flaw within this system is also that if a different host happens to obtain a previously used IP address, the user data corresponding to that address could be seen. So the uniquely generated file names would also help in that aspect as well.

One other problem that is expected regarding this file-saving system is that if the client accidentally disconnects from the session, the server will not able to close properly and it will not save the data. This occurrence is namely because the server won’t exit and save the data

unless prompted by the client (Sort Of how a program may ask if you wish to save your changes if you exit). If the client does not send a command “3” to prompt the connection to close, the server will not save the changes or operations performed on the data. This issue could be solved by having the server continuously write to the .txt database file at the end of every command, or implement better safe exit protocols in case of accidentally disconnections.

It should also be mentioned that the server and client have a close loop regarding to the command packet structure. In which the first index of the comma delimited string is the command given by the user. The user only gets control on how the data is ordered after this command index. This allows for the server to understand what operation the client application wants to perform, and no illegal operations may occur. Because the system runs on TCP, the data in the string is received in-order by both the client and sender, so both parties understand how the command packet is structured.

To elaborate on command “9”, in which the client requests the server to go into a non-persistent connection. When command “9” is inputted by the user, the client-side application will trigger a local stored boolean called “nonpersis”. The client-side application will then show the command menu again. When the user selects a command, a transaction will occur. At the end of the transaction, the client will send a command “3” to the server, telling it wishes to close and end the connection session. On the server-side, it will receive this command “3”, and exit and save the database list to the .txt file. On the client side, when the connection is disconnected the Python program will prompt the user and ask if they wish to restart and open a new connection to perform a new transaction.

III. EVALUATION

Testing of this server and client application was performed using two computers connected to the same network. Before the applications were ran, both computers ran the command terminal internet protocol configuration command (ipconfig or ifconfig) to determine the IP addresses. After this, the ping command was ran to ensure that both computers were able to ping each other and nothing was blocking their communication.

From here one computer was designated as a dedicated client (Will refer to this computer as HostB). The other computer would be running the server as well as an instant of the client application as well (HostA for the server, HostC for the second client application). The operating system used for HostA and HostC was Windows 10, and the HostB client application was running on Lubuntu 16 (A UNIX operating system). All hosts were running the scripts using Python 2.7.

For the first set of testing, the commands of the protocol were tested between HostA and HostC (So onboard one computer). A list of known palindromes were used to test the palindrome checking algorithm running on server-side [6]. Each command was tested, the results shown by the server are immediately shown after the user input in most cases as shown in the following figure:

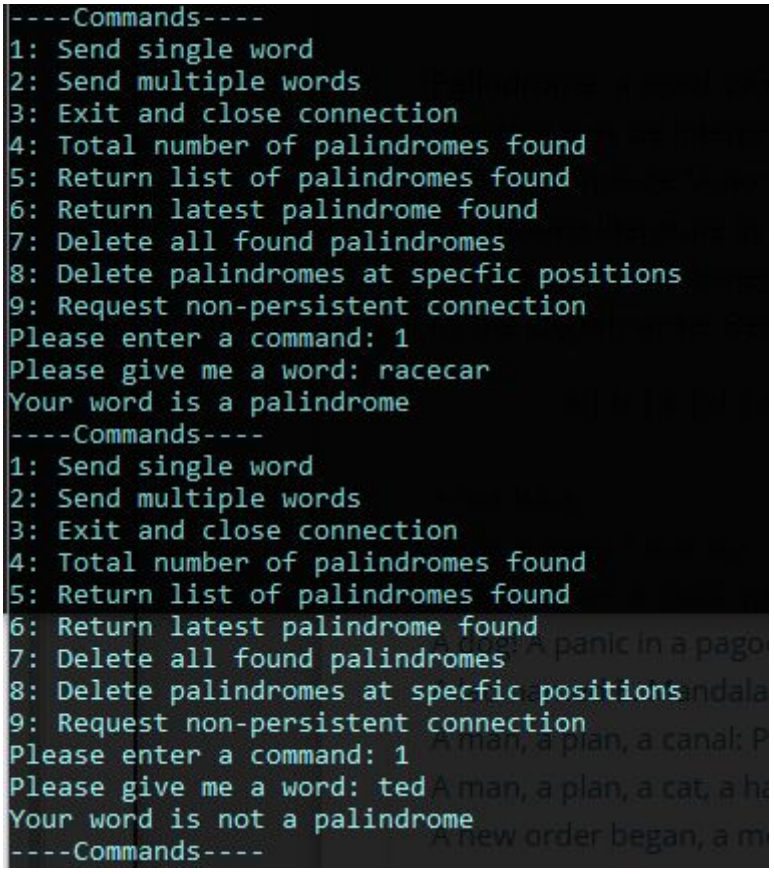

Figure 3. Sample server reply

It was noted in this specific test scenario all of the features work. The connection was moved into non-persistent mode, and the .txt file database was checked after every transaction to ensure that the data has been written and loaded correctly.

In this test scenario, actions were performed to try to break the client-side and server-side applications. Such as inputting improper commands into the client application, or inputting words instead of position for the command “8” option to delete specific positions. As discussed in part 2, the user input verification is also working for the accounted test cases.

When HostB was introduced into the system however, the protocol and network began to run into errors. With both HostC and HostB making requests, the server should theoretically be making multiple threads to handle the scenarios. Using Windows Task Manager, it could be seen that multiple processes are indeed running during connection requests. However when the clients try to send commands or data to the server - only one of the clients would be handled (Usually the client who sends the request first). The other client will receive replies, but they would be null. So it can be seen here that the implementation multi-threading capabilities as some sort of error. Looking closer, it could be estimated that the error has to do with resource management of the application possibly.

IV. CONCLUSIONS

In conclusion, it can be seen that the core requirements of the application have been met, aside from multi-client functionality. Looking closer at the server implementation code, other ways of handling multithreading could be investigated to see if the server can achieve handling multiple clients simultaneously. The implementation has the threads utilizing the socket connection object, and perhaps some function is blocking new connections from utilizing that same object as well. More low level control may be necessary to allow for multiple processes to utilize that connection object to read and write to the TCP message buffer.

For further suggestions regarding the implementation, a better naming system should have been implemented in the case where the client IP addresses may be changed (As discussed in Part II). The client side application UI could also be improved to clearly show the user the server reply, as a lot of text appears when the command window opens. This could be simplified by hiding the command window and allowing the user to input a “help” command as done by a lot of conventional terminal based applications.

References

[1]. http://net-informations.com/python/net/thread.htm
[2]. https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers
[3]. http://beej.us/net2/html/syscalls.html
[4]. https://docs.python.org/2.7/library/socket.html
[5]. https://www.w3schools.com/python/python_lists.asp
[6]. http://www.palindromelist.net