CPSC 471 Programming Assignment

TCP File Transfer Protocol

Project Report

By: Group # 5

Tommy Ly

Diego Barrales

Kevin Cacho

CPSC 471

Fall, 2023

Professor: Mutalip Kurban

Department of Computer Science

California State University, Fullerton

December 2, 2023

2. Table of Contents

3. Protocol Design

At the start of this project, none of us fully understood how a client/server TCP socket connection worked. To tackle this issue, we spent a few meetings simply reading through the sample code given to us, and we googled and watched various tutorials in order to understand the basic structure of a TCP socket connection. After a week, we began to focus on how we wanted to implement our project in accordance with the requirements. Diego took it upon himself to create a representation of our design through the tools the website Figma provided.

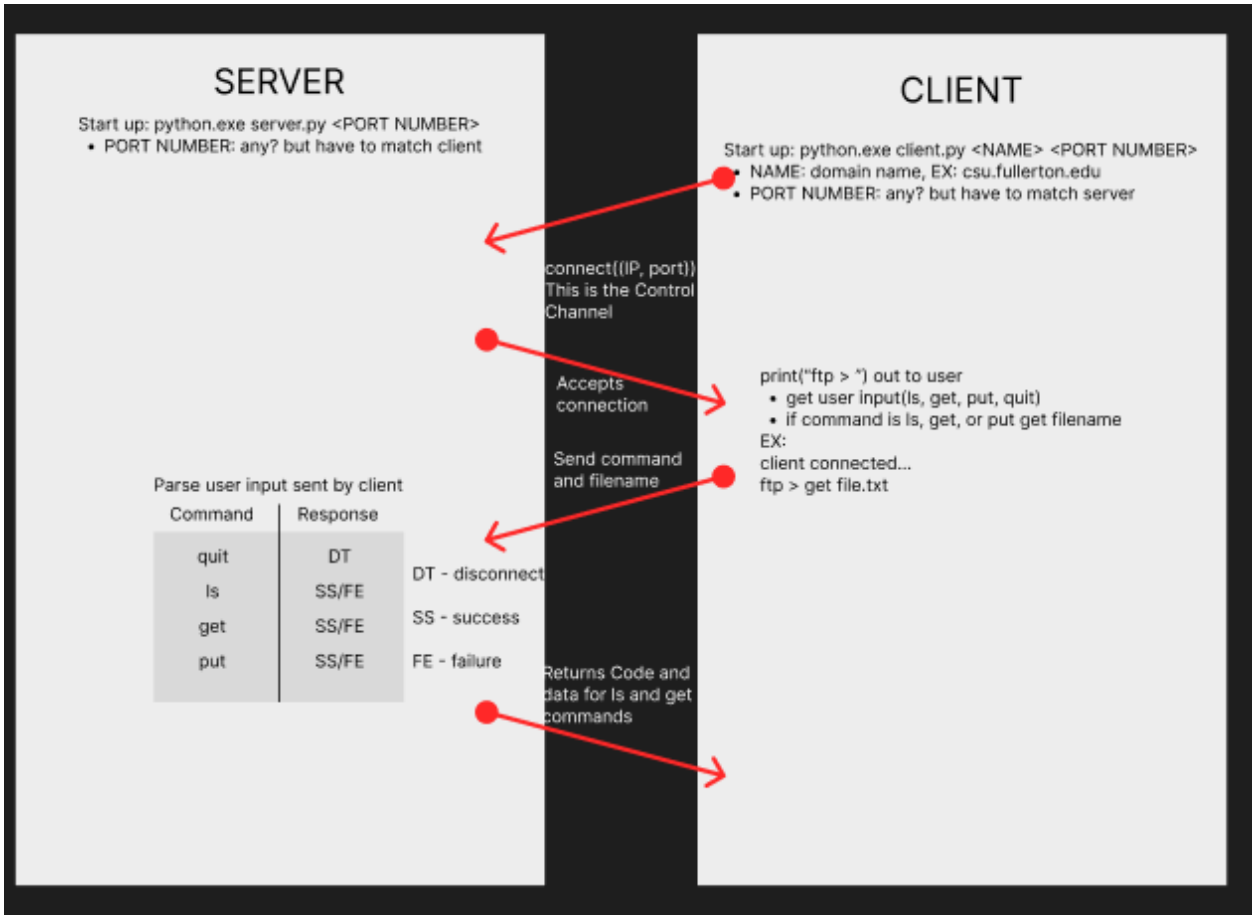
One difficulty that occurred while creating the design was learning how to use the Figma tools and the spacing required to make the image look presentable. After that, there was a discussion about how we wanted to implement the connections and the commands. After the discussion, we determined we wanted to keep the order of communication as a back-and-forth between client and server, as well as keep a strict format for how the server would read the commands. This strict format was implemented by formatting the packet structure used to send data to and from the server. To send any data to the server, we must first send four bytes to indicate the length of the command that follows. Four bytes was more than enough because the longest command, “quit,” is 4 bytes long. Next, we must send the command itself. After that any remaining data that needs to be sent must first send ten bytes that represent the length of the data that follows. This formatted packet structure is required because on the server, we do the steps backward to receive the data. First, we listen for and receive four bytes of data, then use that length to receive the following command. Then repeat listening for ten bytes of data, and use that length to receive the data that follows. On top of this, we use a switch case to determine the command received because depending on the command, the server sends back different amounts of data.

For the get, ls, and quit commands, we reviewed the requirements document in order to provide the desired functionality properly. The first command we worked on and completed was the “get” command. We used the reference code to make functions that would allow the client and server to send data across the TCP connection. Our first iteration of this working command did not utilize the strict format that we designed because we wanted to ensure proper functionality before adding error checking. However, this caused future problems when implementing the other commands since they also didn’t follow any format. After learning that the formatting was the issue, we implemented a packet structure to send across the TCP connection. Once the “get” command was working, the “quit” was easily implemented by simply sending the command to the server to end the conversation with that client.

Before work began on creating the “put command,” there was a lot of research done beforehand and even then issues came up. We referenced the sample code we were given in class first in order to understand the basic syntax and to get refreshed on how the Python language worked. Afterward, we looked up the sample socket tutorials on GeeksforGeeks in order to verify our understanding of the back and forth between server and client. After following along with the tutorial, it turned out that the functionality of the code was not what was desired from the requirements, so we initially set out to modify it until it worked. The modifications began by trimming down the code until it did the bare minimum so that no bugs would occur but still bugs persisted. It was a big issue because the file being created was not in the correct location as it would be placed alongside the server and client Python file instead of being in the appropriate folder. Afterwards, there was an issue where the data being transferred into the file was not only the data within the file but also the data being sent from the client such as commands.

In order to rectify these issues it took a few hours of our combined brains looking at the problem and referencing both google search results as well as the completed code for the other commands. There were plenty of hazards such as file corruption and having to begin anew from git cloning the repository as well as the toughest enemy of all, using a virtual machine while livestreaming that screen. Eventually we resolved the issues step by step going over every line in the code and debugging by using print statements in order to look at the values being stored and sent. After the struggle we immediately attempted to update the main branch and faced an issue where it was not allowed to upload to main and so we pushed to another branch and from there used a pull request in order to merge the branches and finish updating the main branch with all of the commands functional.

4. Diagram of our Protocol



5. Example of Running the Program

1. Running Server

```
kcubu@kcubu:~/code-environment/cpsc471/latestcode2/cpsc471ProgrammingAssignment$
 python3 server.py 1234
The server is ready to receive
[WAIT] For connections...
```

2. Running Client (Failure, wrong port)

```
kcubu@kcubu:~/code-environment/cpsc471/latestcode2/cpsc471ProgrammingAssignment$
 python3 socketClient.py localhost 2222
Could not connect to localhost:2222. The server may not be currently running or
the port is wrong.
```

3. Running Client (Success)

Client:

```
kcubu@kcubu:~/code-environment/cpsc471/latestcode2/cpsc471ProgrammingAssignment$
 python3 socketClient.py localhost 1234
connected to server
ftp >
```

Server:

```
kcubu@kcubu:~/code-environment/cpsc471/latestcode2/cpsc471ProgrammingAssignment$
 python3 server.py 1234
The server is ready to receive
[WAIT] For connections...
[CONNECTED] to client: ('127.0.0.1', 36402)
[WAIT] For command from client...
```

4. Running the “ls” command to print the list of files on the server

Client:

```
kcubu@kcubu:~/code-environment/cpsc471/latestcode2/cpsc471ProgrammingAssignment$
 python3 socketClient.py localhost 1234
connected to server
ftp > ls
Files in Server:
   - put.txt
   - large_file.txt
   - small_file.txt
   - file2.txt
   - file1.txt
ftp >
```

Server:

```
kcubu@kcubu:~/code-environment/cpsc471/latestcode2/cpsc471ProgrammingAssignment$
 python3 server.py 1234
The server is ready to receive
[WAIT] For connections...
[CONNECTED] to client: ('127.0.0.1', 36402)
[WAIT] For command from client...
[INFO] Recieved command: ls
[INFO] ['put.txt', 'large_file.txt', 'small_file.txt', 'file2.txt', 'file1.txt']
[SUCCESS] Executed: ls
[WAIT] For command from client...
```

5. Running Get command to download a file from the server

Client:

```
kcubu@kcubu:~/code-environment/cpsc471/latestcode2/cpsc471ProgrammingAssignment$
 python3 socketClient.py localhost 1234
connected to server
ftp > ls
Files in Server:
   - put.txt
   - large_file.txt
   - small_file.txt
   - file2.txt
   - file1.txt
ftp > get small_file.txt
Getting small_file.txt from server
Saving the file data to `clientfiles/small_file.txt`
Data transfer complete! Filename: small_file.txt, Bytes Transferred: 1471
ftp >
```

Server:

```
kcubu@kcubu:~/code-environment/cpsc471/latestcode2/cpsc471ProgrammingAssignment$
 python3 server.py 1234
The server is ready to receive
[WAIT] For connections...
[CONNECTED] to client: ('127.0.0.1', 48282)
[WAIT] For command from client...
[INFO] Recieved command: ls
[INFO] ['put.txt', 'large_file.txt', 'small_file.txt', 'file2.txt', 'file1.txt']
[SUCCESS] Executed: ls
[WAIT] For command from client...
[INFO] Recieved command: get
[INFO] Recieved file name: small_file.txt
[SUCCESS] Executed: get
[WAIT] For command from client...
```

6. Running the Put command to upload a file on the server

Client:

```
kcubu@kcubu:~/code-environment/cpsc471/latestcode2/cpsc471ProgrammingAssignment$
 python3 socketClient.py localhost 1234
connected to server
ftp > ls
Files in Server:
   - put.txt
   - large_file.txt
   - small_file.txt
   - file2.txt
ftp > put file1.txt
putting file1.txt to server
ftp >
```

Server:

```
kcubu@kcubu:~/code-environment/cpsc471/latestcode2/cpsc471ProgrammingAssignment$
 python3 server.py 1234
The server is ready to receive
[WAIT] For connections...
[CONNECTED] to client: ('127.0.0.1', 37854)
[WAIT] For command from client...
[INFO] Recieved command: ls
[INFO] ['put.txt', 'large_file.txt', 'small_file.txt', 'file2.txt']
[SUCCESS] Executed: ls
[WAIT] For command from client...
[INFO] Recieved command: put
[SUCCESS] Executed: put
[WAIT] For command from client...
```

7. Quitting the program

Client:

```
kcubu@kcubu:~/code-environment/cpsc471/latestcode2/cpsc471ProgrammingAssignment$
 python3 socketClient.py localhost 1234
connected to server
ftp > ls
Files in Server:
   - put.txt
   - large_file.txt
   - small_file.txt
   - file2.txt
   - file1.txt
ftp > quit
```

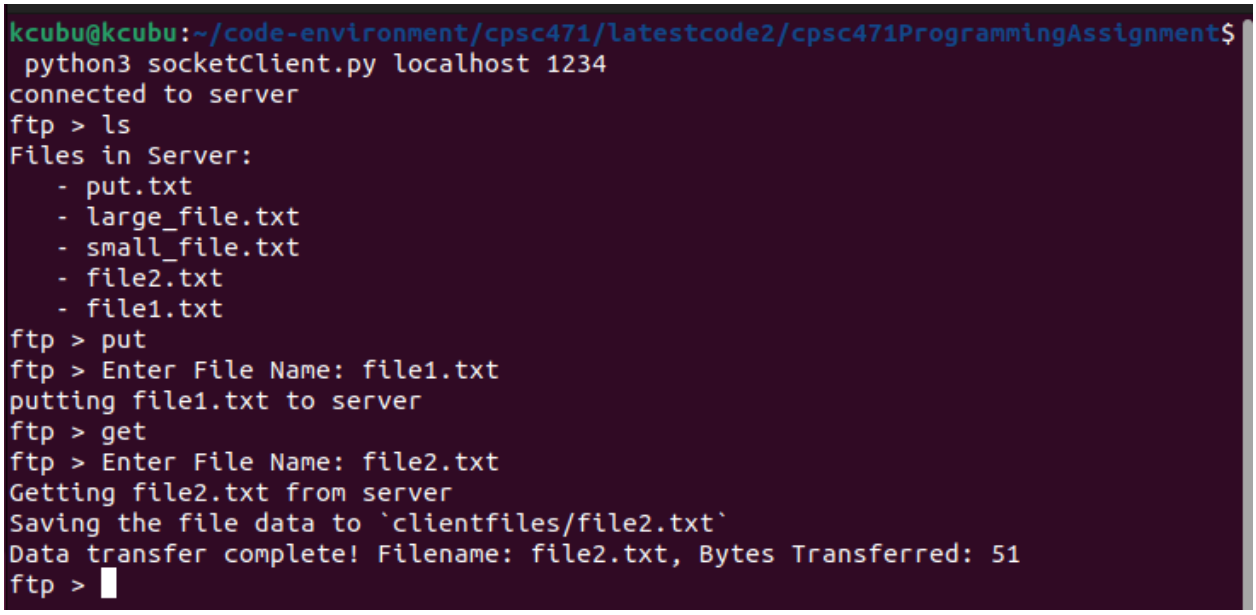
Server:

```
kcubu@kcubu:~/code-environment/cpsc471/latestcode2/cpsc471ProgrammingAssignment$
 python3 server.py 1234
The server is ready to receive
[WAIT] For connections...
[CONNECTED] to client: ('127.0.0.1', 50020)
[WAIT] For command from client...
[INFO] Recieved command: ls
[INFO] ['put.txt', 'large_file.txt', 'small_file.txt', 'file2.txt', 'file1.txt']
[SUCCESS] Executed: ls
[WAIT] For command from client...
[INFO] Recieved command: quit
[DISCONNECTED] From client: ('127.0.0.1', 50020)
[WAIT] For connections...
```

8. Other Cases

- If the client just enters the get or put command without providing a filename, the program will ask the client to provide that file

Client:


kcubu@kcubu:~/code-environment/cpsc471/latestcode2/cpsc471ProgrammingAssignment$
python3 socketClient.py localhost 1234
connected to server
ftp > ls
Files in Server:
- put.txt
- large_file.txt
- small_file.txt
- file2.txt
- file1.txt
ftp > put
ftp > Enter File Name: file1.txt
putting file1.txt to server
ftp > get
ftp > Enter File Name: file2.txt
Getting file2.txt from server
Saving the file data to `clientfiles/file2.txt`
Data transfer complete! Filename: file2.txt, Bytes Transferred: 51
ftp >

6. Modules Used

1. **socket**: The main library that was used for the project. It was used to create a server socket and client socket and connect them together. When connected, it was used for both sending data and data retrieval in all commands (get, put, ls, quit). Finally, when the client disconnects from the server by running the quit command, the client socket is closed.
2. **sys**: Used to extract the command line arguments when running the server and the client. Sys extracts just the port number when running the server and both the port number and the server name when running the client.
3. **os**: Used on the server side to obtain a list of files that are currently on the server.

- Note: All modules used in the project come from the Python Standard Template Library meaning that they come with a standard installation of Python. This means that you would not have to use pip to install these modules manually.

4. **datatransfer.py**: This module was created by us and was imported for both the client and server code. This module defines the main core functions of data transfer involved in the program. There is a function for sending data (sendData), sending files (sendFile), and receiving data (recvData). There is also a function called prepareSize which returns the size of length of the data as a byte string. This will be prepended before each piece of data that is being sent (command, filename, file contents) so that the server knows what the size of the data is and how much data to expect.