Authors: Sebastian Reel, Keegan Graf, Kimberly Meza Martinez
Course: CPE 400 Computer Communication Networks
Project: Final Project – Implement a File Transfer System
Due: May 2nd, 2022

Technical Report - Final Project

Objective

The objective of the final project for the computer networking course is to take certain concepts that were learned in the semester and apply them to a team project within actual implementation. The team working on the project decided to implement the option of a file transfer application. The program should be able to transfer files from client to server, where the server will listen to a port and the client will connect to the server and transfer specific files. This means that the server would be able to be placed anywhere on the system and still receive the directory.

By completion of the project, the team should be able to better understand how file transferring is implemented over a network. They should also have a better understanding of concepts discussed within class and how they can apply to real world situations, such as how checksum works within a network and how to correctly multithread over a TCP connection.

Functionality

As explained in the objective, the function of the overall program should be able to handle a client sending files to a server, and the two communicating to one another that it had been completed. Throughout both the client and server, implementation as well as any edge cases will be explained, as well as how they communicate to one another, concurrently send files over a socket.

"fileClient .py" → The Client Implementation

The overall functionality of this file is for the client to be able to send it to the server. In order to do so, the server must be running and listening first so that the client can run and the server can receive. In order to do that, the socket, os, and argparse libraries were imported in python. The socket library provides functionality for client-server communication, the os library allows the added functionality of directories, and the argparse library adds command-line parsing functionality. The sendFile function does most of the functionality where the main calls the function.

SendFile starts with defining the host and the port number and using os functionality to obtain the current working directory and the items in it. Then a condition is created that finds if the directory is inside of the current working directory. If it does then it is going to get the items from the directory the data is being sent from and loop through the files to connect the path. The paths with the current working directory and the directory name are connected, as well as the items. The socket is then defined and the function displays the hostname and port confirming the connection has worked. The function then obtains the file size of the connected paths and the file name and size are printed by the client when the file is ready to send. The client displays it as attempting to send the file to the server and the pathway and file size in terms of bytes by encoding. The server then lets the client know that the file was received by printing the specific file message on the

server-side. This is done by using the buffer size and decoding. If there is data in the file then the file is opened for reading and connected to the path directory. In a loop, a variable then reads the file within the buffer size constraint and if there is nothing then it stops the loop and closes the socket through a break statement. A statement is then printed stating the server has received information.

Within the main function, the parsing is done. Using the argparse function, the directory name is obtained and the sendFile function is called passing in the directory name. Along with this functionality, print statements are added showing the file transfer application and when it is complete marking the end of the client file.

"fileServer.py" → The Server Implementation

This program contains the functionality for the server end of the dynamic between client and server. This is where the implementation of receiving the folder with the files will be shown, and the program will send a check that the files and any associated data is being attempted to be received and that it was sent from the client.

The program begins with defining libraries and variables that will be utilized within the main loop that the program goes through. The main libraries are going to be the socket and os libraries that come natively with python. For variables, we define the hostname of the user's computer, the port, the formatting that the messages should follow, a buffersize of each of the files, and a count that will be iterated through later on.

First, it starts off by connecting to a main socket, where we bind the host name and the port together with. We give a case here, where the program will listen continuously until the client's program has been executed, and check to see if it worked 10 times before giving up completely. This is a solid edge case for seeing that, if the server is not able to successfully listen to the host the first time, it will continue trying until it can. After all of the main variables and libraries have been defined, the server will stay continuously listening to the socket until the client runs their program. After the program that sends the files to the server has been executed, then it will move on to the main loop, where the bulk of the functionality of receiving the files is defined.

After the client has executed their program sending the files over to the server, then the loop iteration that exists within the server program begins. First, it starts by accepting the socket that was also defined by the client (which should be the same socket port from the server). After accepting both the socket and the IP address that the client transferred files on, the server confirms such with a print to the terminal. Once everything has been accepted, the server then decodes what was encoded from the client and splits it by a certain character, so that the program can easily read the name of the files and the size of them in bytes. To ensure that the name of the file is simply just the file's name and not the entire pathway, an os function called "basename" is used to take the last part of the path, so that just what is needed is taken off of it. After confirming that the file was successfully received, and receiving the client's confirmation that it was sent, the server then communicated back at the same instance that the file was received. The server also displays the file name with its associated size, so that the user can compare the size and name that were sent

from the client's end. This is shown more for the convenience of the user being able to also confirm that the files transferred correctly.

This communication that is happening between client and server is important and is mainly how the team implemented the checksum functionality of the program(s). By checking that the file was received/sent on both ends of the spectrum, the program is versatile in ensuring that if there was a discrepancy along the way, that it would be caught within the transfer and stopped if it did not work correctly.

The latter half of the implementation for the server end of receiving the file data is focused on both placing the files in their respective directory, and handling cases for if a file has data. First, the team implements getting to the new file but defining the current directory the user is on and defining the directory name that the field will end up going inside of, as well as what items are currently within the directory. This is done because of another edge case that the team wanted to implement in the case that the directory is not within the current one. This case joins the path to the newly made directory and creates it, and if it is somewhere else, then it simply just joins the two together so that the client can effectively send the files over.

Once the new directory has been defined and joined to the current one the user is in, the program then focuses on the case of if any of the files have data within them and ensuring that the data is also sent over to the server. First, the server joins the new path to the filenames that are defined, then opens them and states that they will be written to via a binary method. Once the file is opened, a loop is instigated where the contents of whatever file it is currently iterating through are read and received from the client. They are then written to the newly transferred file and shown to be complete. The socket is finally closed since the transfer was complete.

Now, if the file that is being received by the server does not have any data written inside of it at the moment of the transfer, that current iteration is then broken, and the program simply moves on to finishing the process of the transfer. This is another edge case the team added to ensure that all cases of file transfers were met and completed. It is important that this case be added to cover all bases of the files that do not have any data within them, so the program knows to only take the case that the file was transferred and not any data, since there was none.

All of the basic socket definitions are defined within the main definition of the program file and include the implementation of the threading done on the file. In this case, the threading is allowing the program to be able to send more than one file at once over a socket. This will keep running, however, so the program needs to be manually killed after the transfer is complete. The thread does help define this functionality though of being able to concurrently define how many files the user wants to send at a time over each socket, which can be useful in many real-world applications of file transfer applications. These can things such as updates for applications,

After everything within the receive functionality is finished, then the file transfer is complete. Once done, the user can look through the original directory that was sent, and then analyze the other directory that should have received the files. Once looking, the received files will all be in the correct directory, and the files that had data in them will have the respective data. The terminal

will then print all of the respective information and show the name and sizes to each file and finish the program.

Novel Protocol

The novel protocol that the team decided to implement is a functionality with the *time* library with pythons' native libraries. With this, the team implemented a functionality where the amount of time it took to run the file transfer between each server/client communication is displayed and calculated in seconds and a few decimal spaces of buffer. This is so the user can see how long it is taking for the program(s) to fully transfer their files from one directory to another.

This novel contribution is implemented within the server end of the communication, so that the user is able to see how long everything took after the transfer was completed. The time calculation is implemented by taking the function time from the library and subtracting the starting time of the transfer (should be 0) to calculate how long exactly it took from the initial point of execution of the client sending the files to transfer them over. This calculation is then displayed to screen for the user to see so they can analyze how long the transfer took.

Typically, the team saw the data ranging between 0.0004 and 0.001 seconds to transfer the data over from the client to the server. This range is simply if the concurrency is the default 1, and there is one file sent per socket. This speed is prone to change depending on how many threads are defined within the concurrency by the user, and how many files there are total as well. All of these factors can shift the time range drastically. For more information on the exact data seen in the novel contribution, refer to the results section, and Fig. 3.

The reason that the group decided to implement this as the novel contribution is because of the way that transferring files works in real life applications. Whenever a user is backing files up to a cloud, transferring them from one server to another, and more, it is almost always important for the user to see how long it took or will take for the files to fully transfer over. This implementation can be immensely important to have for larger applications, like in corporations, since they most likely have thousands of files and directories that they are transferring over many different servers and sockets across larger gaps of space. If we were to use this in a larger application, the time functionality would definitely be something to implement for convenience.

Results

The results of the report will focus on what is happening when the program is executed, showing the output when each respective program is executed. First will be Fig. 1 and Fig. 2, stating how everything looks before any extra contribution the team made to improve the state of the code. After that, Fig. 3 will display how the program executes with the added novel contribution has been implemented. This section will be further explained and analyzed in the analysis section of the report.

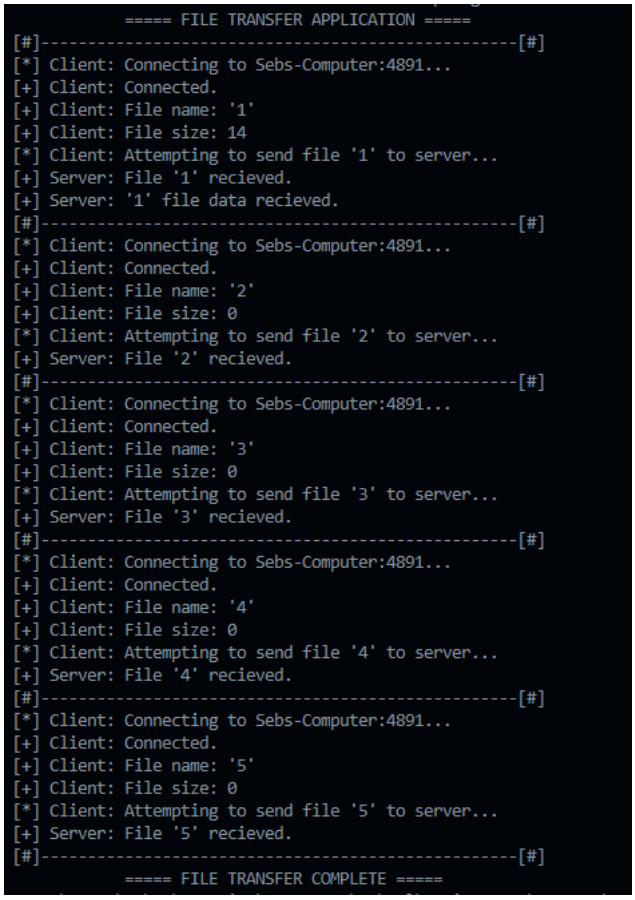

Figure 1: Client sending files over without the Novel protocol implemented in.

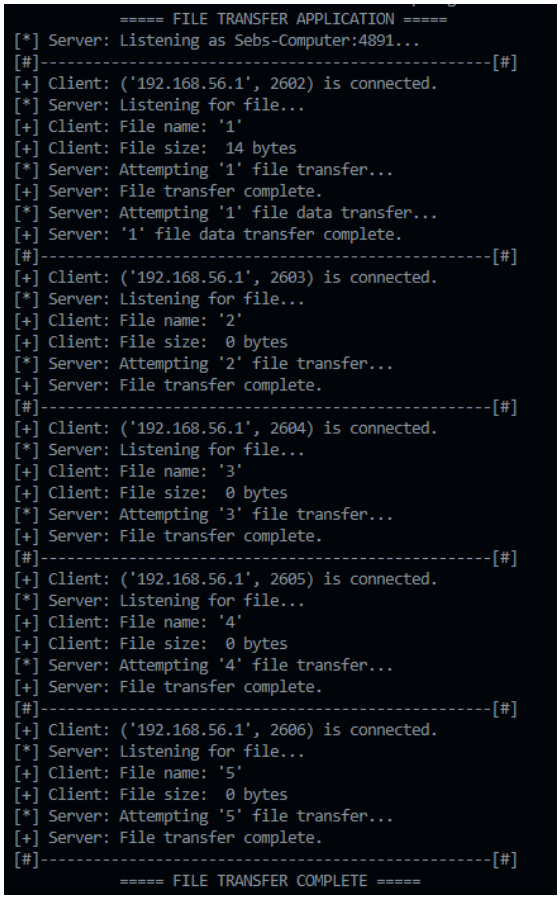

Figure 2: Server listening and receiving files over without the Novel protocol implemented.

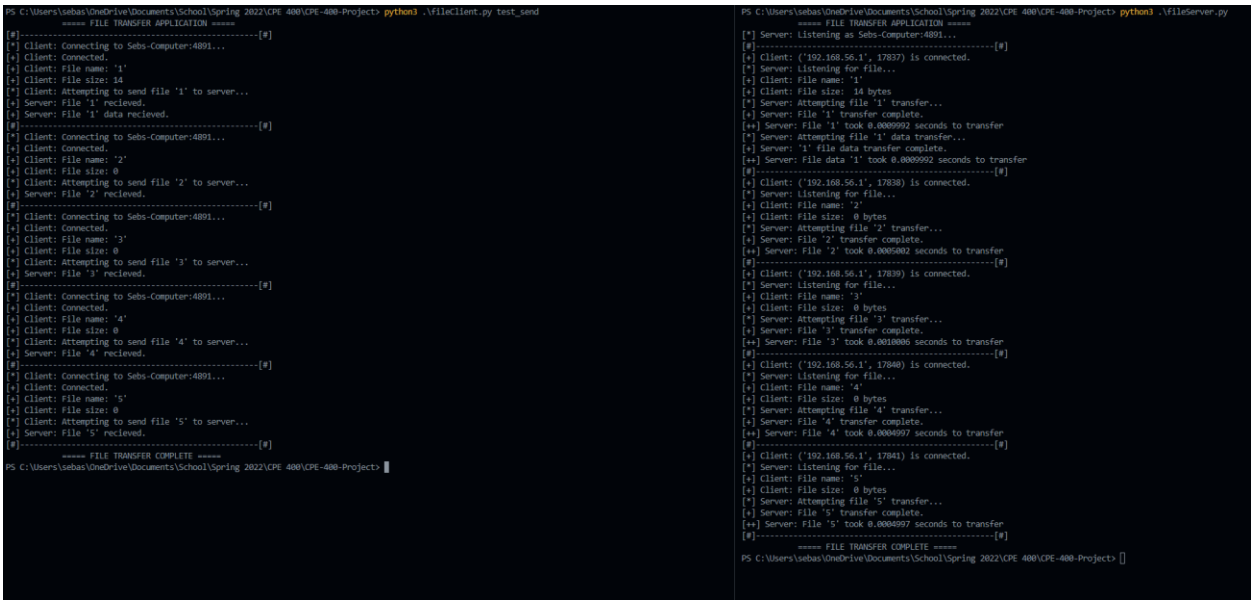

Figure 3: Novel Contribution → The client and server working together to transfer files and implementing (on the right-hand side) a functionality where the amount of time it takes to transfer the files from client to server is shown.

Analysis

Figure 1 → Client:

Figure 1 displays the client sending the files over one by one. It is interesting to note that it first will accept the size of the file, even if there is nothing within it. The program will not work without the server program first listening to the socket port, which is explained in Fig. 2. Every time that the process is attempting to send the file over, or connecting to the socket port, or really doing anything that involves a process being ran, the action is denoted by a "[*]". If it is simply a statement of the file name or size, the server confirming the file being received, or anything else that is not a process, the action is denoted by a "[+]". The team used these keys to differentiate the action for two reasons. Firstly, because it's so the user can understand what is exactly going on during the file transfer. And secondly, the novel contribution was slightly easier to point out and implement later on.

Figure 2 → Server:

Figure 2 displays the server first listening to the socket port, and then receiving the files one by one. Similar to Fig. 1, the server's output also expresses different notations for the processes and statements. The server does a similar thing to what the client does but reads the file size and name based off of what was decoded from the client, instead of simply just displaying the files name and size. Then it will also attempt the transfer and confirm it had been completed afterward.

Figure 3 → Novel Contribution:

Figure 3 displays both the server and client working together to complete the file transfer, and it also includes the novel contribution of adding a timer to the program to display how long it takes for the files to transfer completely. It is interesting to notice that all of the times fluctuate from file

to file, even if they have the same exact size. Also, it is important to see that even if the file has no data inside of it, the file itself still takes time to transfer over.

The novel contribution also exposes that transferring the file data and the file itself, if there was data inside of it, takes a different amount of time to transfer. This is seen with file 1, which has 14 bytes inside of it. Since the file has data, it takes a slightly different amount of time to transfer than just the file itself with the data inside. This is probably best explained with the conclusion that the socket sends over the data within the file altogether rather than separately.

In the novel contributions output, the notation of "[++]" is used to differentiate it from the rest of the output. This is done so that it can be easily seen by the user, since it is one of the most important components to move files from one place to another. Since the time elapsed is a statement, we printed and also expressing a process, the team thought it would be best to display it in this way.

Conclusion

The file transfer application project encapsulated being able to implement everything that was seen across the report created by the team. After implementing the communication between the server and client, the team should have been able to understand more about the real-world applications of file transfer applications and how the concepts learned from the course are implemented outside.

There are many other protocols and functionalities that can be added to even further develop the complexity of the server and client file transferring, like estimating on progression with a progress bar and implementing more argument so that the user5 can further solidify exactly how they want to transfer their files (i.e. from one address to another perhaps). The team will continue to further develop their understanding of the content by implementing and practicing on these ideas in the future as well.