Distributed Computing: Analysis of File Transmission Data between Three Machines using Server Router Lookup

Kennesaw State University

Department of Computer Science

CS 4504: Parallel and Distributed Computing

February 13, 2023

Nikita Smith – nsmit210@students.kennesaw.edu

Ethan Butler – ebutle17@students.kennesaw.edu,

Kawika Hodge – chodge26@students.kennesaw.edu,

Carlos Macias – cmacias1@students.kennesaw.edu,

Claudio Gutierrez – cgutie22@students.kennesaw.edu,

Quinn Desmond – jdesmon3@students.kennesaw.edu

Abstract

File message passing in a distributed computing environment can be implemented to collect data on file size, sending/receiving time, and overall processing time. By modifying a system that was already set up to transmit text files, audio and video files were also able to be transmitted between a client and server, using a server router lookup method. Additional code was implemented to collect data on client file size, server file size, client send time, and client receive time. The system was designed and configured to be executed with three machines, using four major classes. One machine utilized the client class, another acted as a server, and the intermediary was the server router class along with a set of SThreads. After data collection from multiple runs of text, audio, and video file transmission of different sizes was completed, graphs were created to show trends across the aforementioned metrics. File size was consistent before and after transmission, and there were large fluctuations in sending times across multiple runs.

Introduction

The following report provides an outline of the design, implementation, and analysis of the distributed system set up to pass various file types between machines. The process begins with the initial configuration and execution of the system and is followed by data collection and analysis, with figures and tables. The implementation goal of this project is to modify a given set of programs for clients, servers, and server routers in order to transmit text, video, and audio files, and provide multiple types of transmission measurements. The nodes in this system communicate using a TCP link and direct/redirect communication using a server router. The different programs run on different computers and code collects data for message size, transmission time, and router-table lookup. This data is analyzed in order to draw conclusions from correlations between metrics of different file sizes and types.

Design Approach

The system is made up of four major classes: two for the client and server, and two to assist with message routing. The TCPServer class handles everything related to the server and is simple in design. It first attempts to connect to the routing server, and if successful, awaits data to be transferred from the client. Before modification, the server will take in any received data as text, transform it into all-caps, and send the modified text back to the client. The client is similarly simple. It attempts to connect to the routing server, and if successful, begins to read data from a file. It takes the data and sends it to the server. It will then wait for the server's response and print out the text that it receives. Both the client and server use sockets to handle data transfer, with the routing table used to properly link the sockets together.

The TCPServerRouter and SThread classes are a tad more complex than the client and server. The TCPServerRouter looks for sockets trying to connect on a certain port and adds the socket's origin and destination to a routing table. This routing table contains a destination address and a corresponding socket to send data through. Each new connection to the router generates its own SThread, with every SThread handling an individual connection to the server. The TCPServerRouter shares the routing table with the SThreads upon creation, and the SThread will fill out part of the routing table using the information passed to it, as well as internally keeping track of the input socket from the initial connection.

When an SThread begins to run properly, it first waits 10 seconds to allow the routing table to properly fill from all other threads. After this, they will search through the routing table to find the matching output socket and begin a loop of passing any data from the internal input socket to the located output socket.

Data Collection and Program Execution

A crucial part of this project was implementing the programs in a robust physical and digital environment that allows for the proper execution of client-to-server communication while collecting valuable data. To achieve this, we defined testing scenarios, one for text file transmission and modification, and an additional setup for client-to-server. In these two scenarios, a total of 4 files were used to ensure the scalability and reliability of the utilized code and increase the diversity of the resulting data.

Data Collection for Text Files

Our data collection process for text files used a set of modified code to track important telemetry while modifying the file using a server-client structure with a router facilitating

communications. Our modified code allowed for tracking several essential statistics that help determine how effective the program execution is at scale. One of the most critical statistics was message processing time, which would track how long it takes for a message to be sent from a client to a server and for a response to be sent back. In addition, we also tracked how many milliseconds of total message transmission and reception are required. We collected data such as sent message byte size and roundtable lookup timing. Since the amount of data sent between the client and server does not change at any time, message byte size was primarily used to ensure that all data was correctly delivered. The table lookup time was minimal due to its small size and the low number of connections, making it have no measurable impact on the program's performance.

The Client Machine is a 2018 Razer Blade 15 with an Intel Core i7-8750H CPU, the router is a desktop tower with an AMD Ryzen 1800X CPU, and the server is a 2016 MacBook Pro with an Intel Core i7-6920HQ CPU. All communications are handled over IPv4 Wi-Fi through socket port 5555, with each device being on the same shared network at the time of code execution. The router java code (TCPServerRouter.java and SThread.java) is run first with the Client-side code (TCPClient.java) and Sever-side code (TCPServer.java) being executed simultaneously. After each successful execution, the required data was printed to the terminal, allowing us to quickly transfer extensive data from each run to an Excel spreadsheet for analysis.

For testing text file modification, two .txt files were chosen. One was a relatively small file, occupying 3527 bytes, named sample3.txt. The second .txt file was 2167739 bytes, named sample-2mb-text-file.txt. Our goal in using an additional large text file was to ensure that the network configuration operates under a more sustained load. Each file is transmitted from the client to the server, swapped to uppercase, and sent back, with the correlated data for the

execution tracked using the terminal output and an excel spreadsheet. We repeated this process 20 times, ten times for the sample3.txt file and another ten times for the sample-2mb-text-file.txt file, with the results from each execution being recorded for all 20 runs. The following figures display an example of the complete execution for the sample3.txt file.

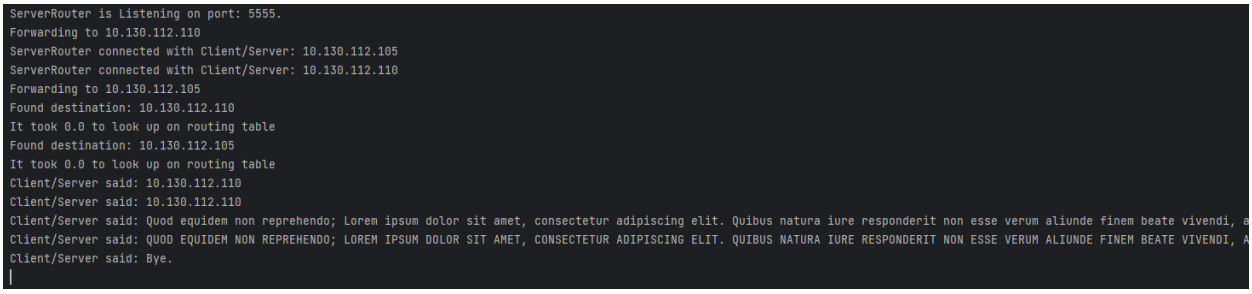


Figure 1. Screenshot from Router terminal for successful text file transmission and modification

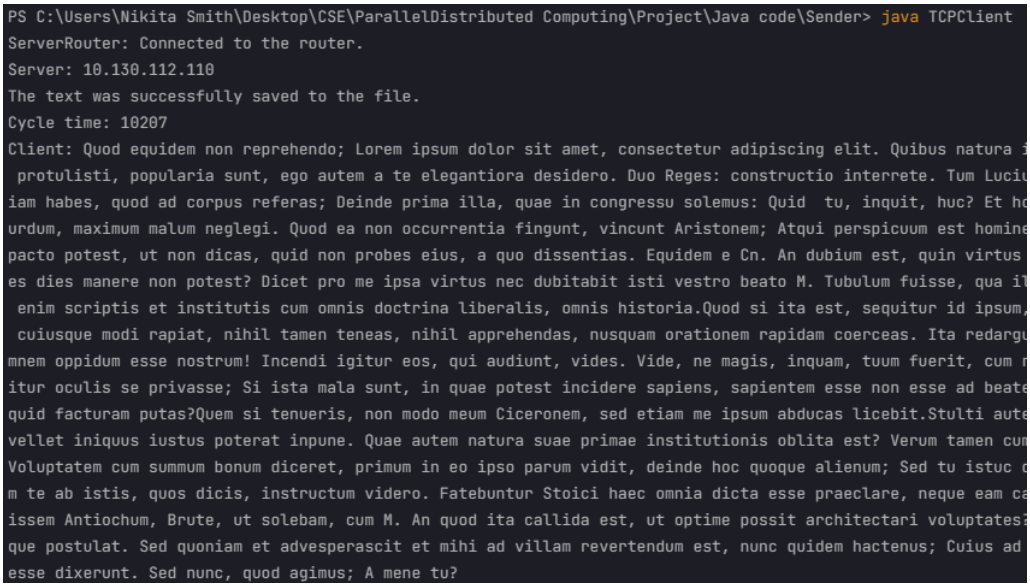


Figure 2. Screenshot from Client terminal for successful text file transmission

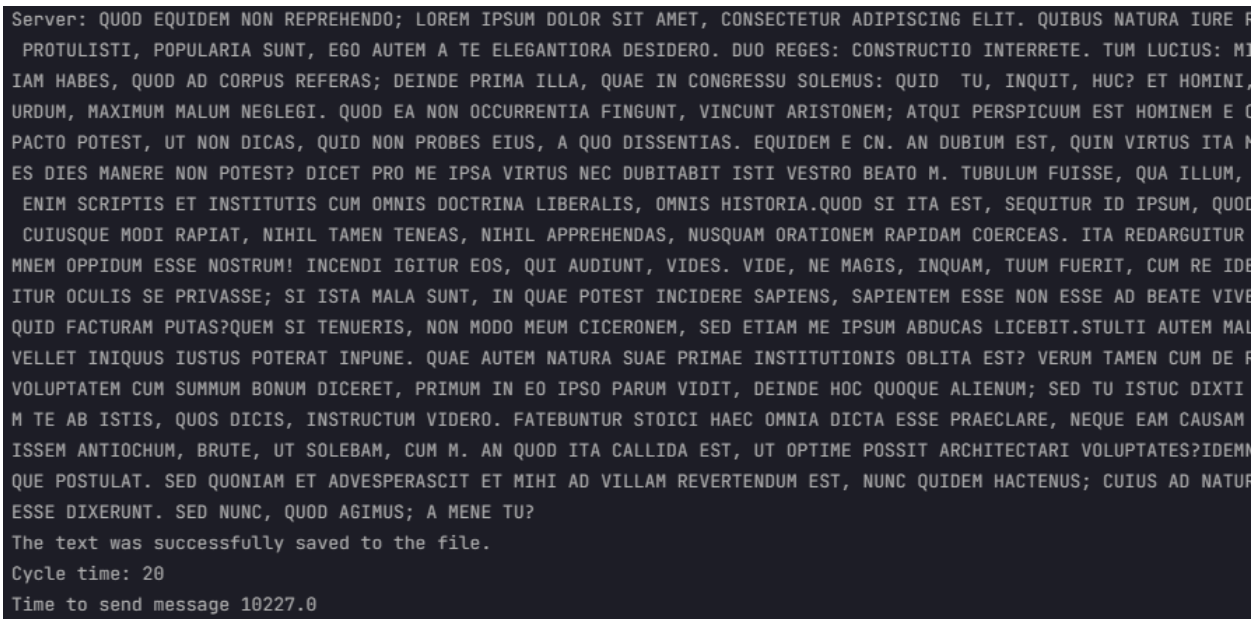


Figure 3. Screenshot from Client terminal for successful text file retrieval after modification

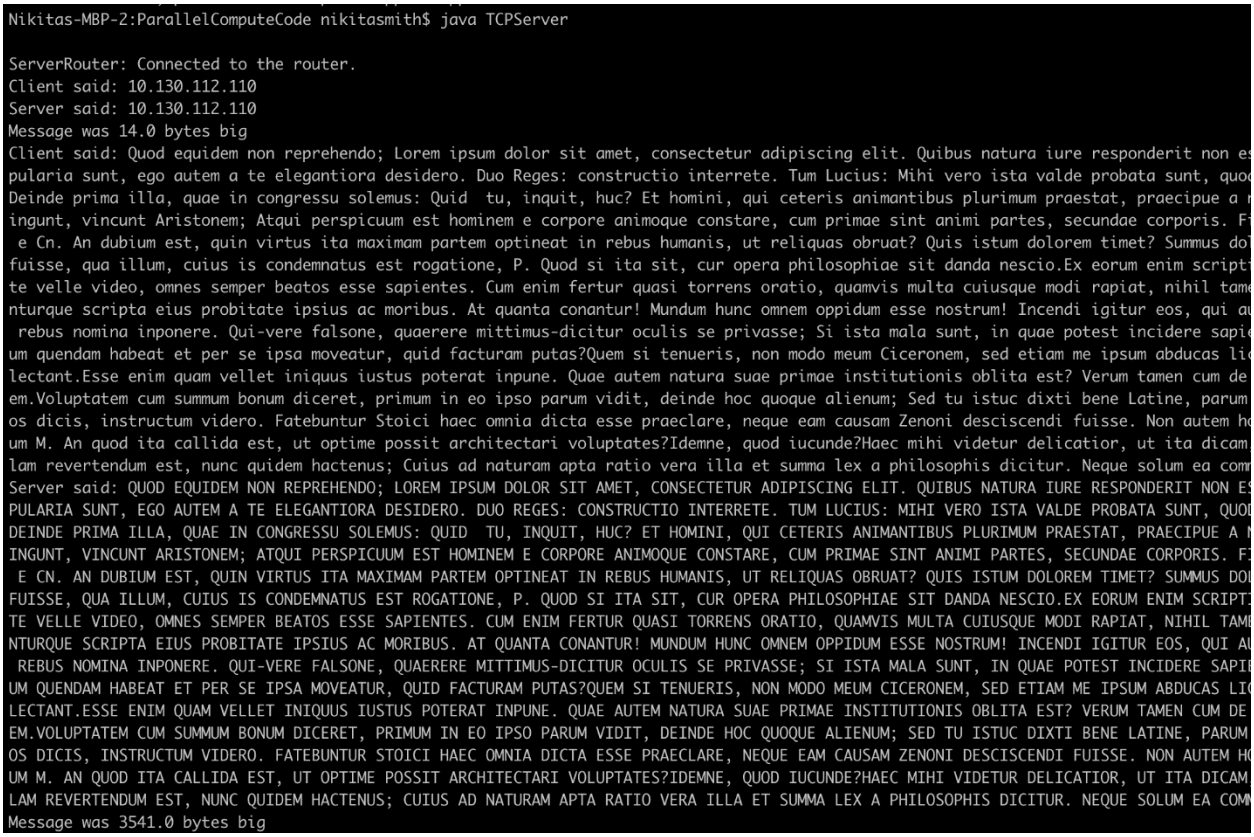


Figure 4. Screenshot from Server terminal for successful text file received and modification.

One important thing to note is how the router, client, and server all output the sent message from the client to the server and all uppercase messages produced by the server, as shown in figures 1,2,3 and 4. Additionally, we can see the printing of essential data, such as the received message byte size by the server in figure 4 and the time to send the message from the client to the server in figure 3.

Data Collection for Audio and Video Files

Our execution and data collection process for .mp4 video and .mp3 audio files were very similar to our process for .txt files, but with a few key differences. The files selected for this process were a 2-minute long, 8,660,917-byte sized .mp4 video file and a 4-minute long, 5,566,125-byte sized .mp3 audio file. These files were stored on the client machine and, through program execution, would be added to the server's file space. The machines used in this process were the same as the client, router, and server, the same three separate machines used for the text file data collection. Additionally, the files executed, and the order of execution were the same. The router would be the first to begin execution, followed by the client and server simultaneously. The only difference is that the file.txt stored alongside TCPClient.java on the client was replaced with testAudio.mp3 or short.mp4, depending on the type of file being used in the execution.

After successful file transfer, the output video or audio file would be stored in the same directory as the TCPServer.java file on the server. This file would be titled out.mp3 or out.mp4, depending on the type of file sent by the client machine. Also, upon file transfer completion, the server and client machine would print valuable data about the transfer to their respective terminals, such as the total time in milliseconds required for file transfer on the client and server side, received and sent file size, as well as the average time for message transfer from client to

server. This data was recorded 20 times from 20 test runs, with 10 runs transmitting short.mp4 and another ten transmitting testAudio.mp3. The following screenshots are from one of the successful transmissions of short.mp4.

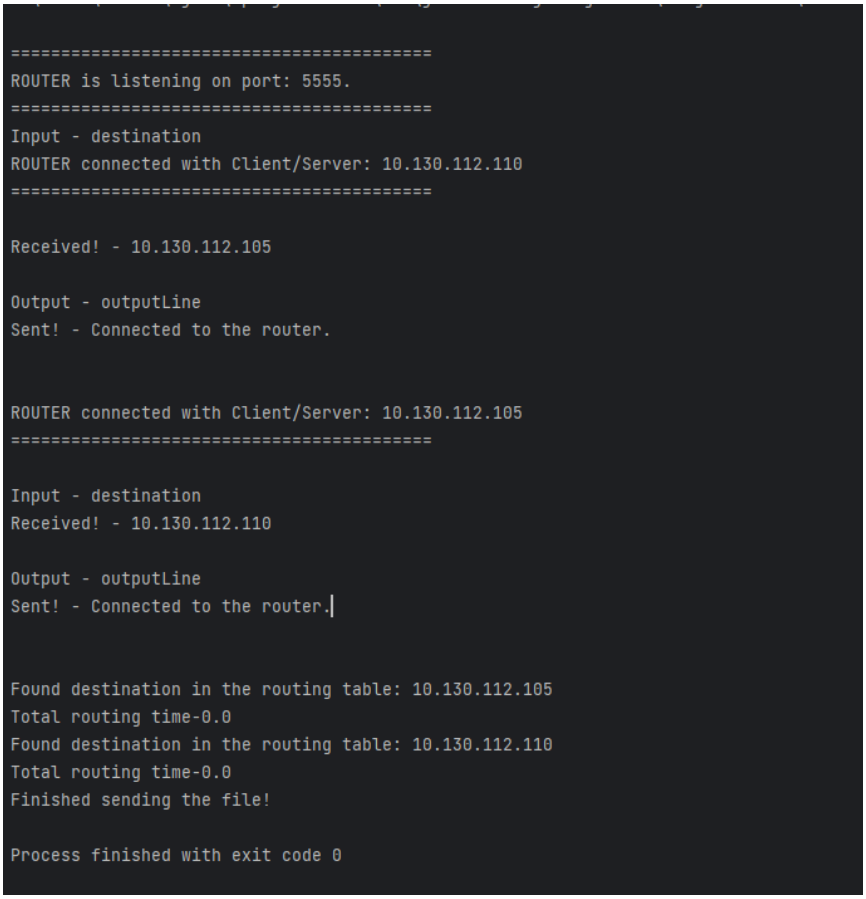


Figure 5. Screenshot from Router terminal for successful video file transfer

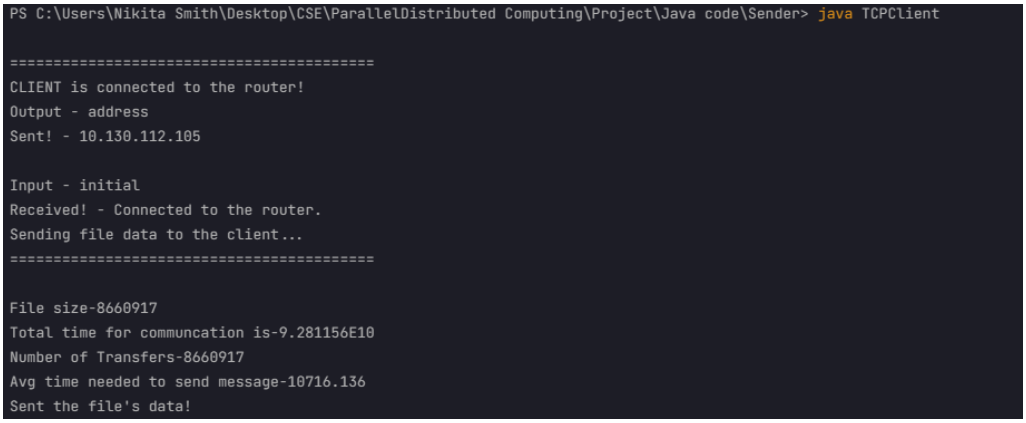


Figure 6. Screenshot from Client terminal for successful video file transfer

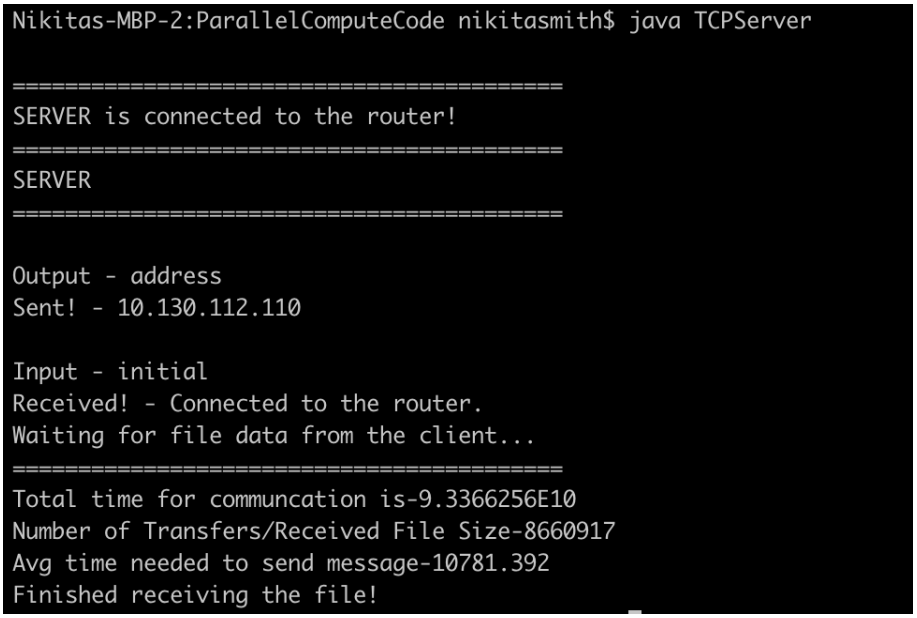


Figure 7. Screenshot from Server terminal for successful video file transfer

As can be observed from Figure 6 and Figure 7, we can verify a successful file transfer by comparing the received file size on the server terminal to the number of transfers on the client terminal. Additionally, we would play the output file on the server at the end of each run to verify the successful file transfer. Also, from Figure 5, we can determine if the server could connect to the correct client and vice versa. Also, we can use the router output to verify when the file has finished transferring to the server.

Implementation of Text File Transmission

Using the provided code, we were able make some minor modifications to successfully transmit text files from the client to the server. Only two things needed to be done to TCPClient.java. The first step was to change the routerName variable to the correct IPv4 address of the router. Secondly, the IPv4 address to the destination, i.e., the server, needed to be set to the server's address. Those two changes were also done to TCPServer.java. However, since the

server's destination was different than the client's, the destination address was a different value, i.e., the IPv4 address of the client.

Implementation of Video/Audio File Transmission

Sending video and audio files provided more of a challenge. It was an iterative process that consisted of reading through lots of Oracle documentation and fixing bugs. There were two main attempts at solving this problem. The initial approach proved to be a dead end which was then followed by a successful approach.

Initial Approach

Initially, a FileInputStream was sent to an InputStreamReader. The InputStreamReader was passed into a BufferedReader like the text files were. This allowed for the data to be sent, but after examining the hex values of the data, extra bytes were passed. This was since the InputStreamReader decoded the bytes into characters using a specific charset. We assumed that if the proper video file encoding was determined, the size of the input file and the size of the input stream would be identical.

Based on these assumptions, we could determine the proper encoding by enumerating through all possible charsets. Using the Charset class and it's availableCharsets() method we retrieved all encodings that Java supports. After enumeration, none of the resulting file sizes matched the original file size, so it was time to take another approach. In hindsight, there are some other problems with this approach that would have arisen, even if it initially worked.

Successful Implementation

The types of data that are needed to be sent between nodes are strings and bytes. The strings are used for general message passing and routing. After examination of the Java

documentation, data streams seemed to be a good choice to transmit the data between nodes. They support reading and writing bytes that are encoded in a UTF format. This allows for strings to be directly passed between nodes. This makes message passing easy and enables the routing table to work properly in SThread.java.

In addition to passing strings, data streams support directly sending and receiving bytes. There is a general approach to passing an input stream to an output stream that is useful and used multiple times in this program.

```
int currentByte;
while ((currentByte = inputStream.read()) != -1) {
      outputStream.write()
}
```

Figure 8. General structure to write an input stream to an output stream.

The variable currentByte, of type integer, is declared to store the value of the byte that is currently being read from the stream. Depending on the stream, different values may be stored here. Regardless, if currentByte is -1, that means the end of the stream is reached. This is used as the condition to terminate the while loop used for reading the stream. Inside of the loop, each value that is read, gets written to an output stream. It is used in TCPClient.java, SThread.java, and TCPServer.java. The client passes the stream of data from the file to the router, which passes it to the server, which passes it to the output file. Their specific implementations can be seen in the figures below.

```
int contents;
while ((contents = fileInput.read()) != -1) {
    // Send: send the current byte of data to the router
    dataOut.write(contents);
```

```
    ...
}
```

Figure 9. Reading a FileInputStream and writing it to a DataOutputStream in TCPClient.java.

```
private int dataIn;
...
while ((dataIn = in.read()) != -1) {
    ...
    // Send: send the current byte of data to the destination
    outTwo.write(dataIn);
    ...
}
```

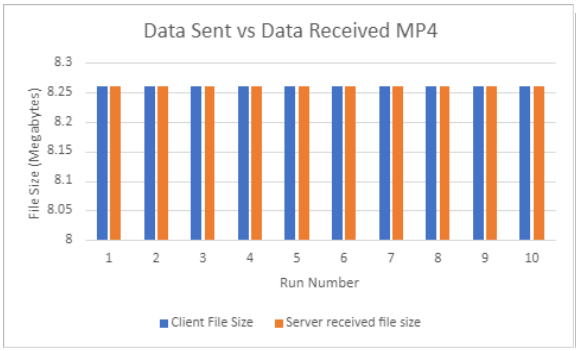
Figure 10. Reading a DataInputStream and writing it to a DataOutputStream in SThread.java.

```
int data;
while ((data = dataIn.read()) != -1) {
    // Send: send the current byte of data to the file
    fileOutput.write(data);
    ...
}
```

Figure 11. Reading a DataInputStream and writing it to a FileOutputStream in TCPServer.java.

Resulting Data and its response

From the data gathered, we found three relationships that inform us of some unknown and understood properties of networked data transmission.



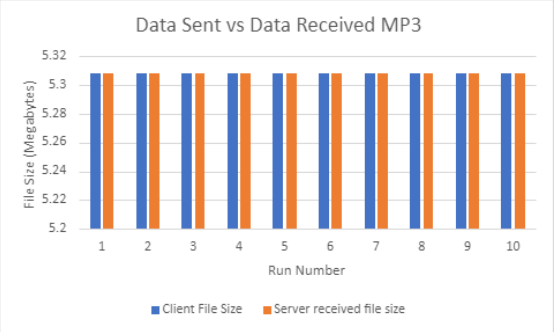
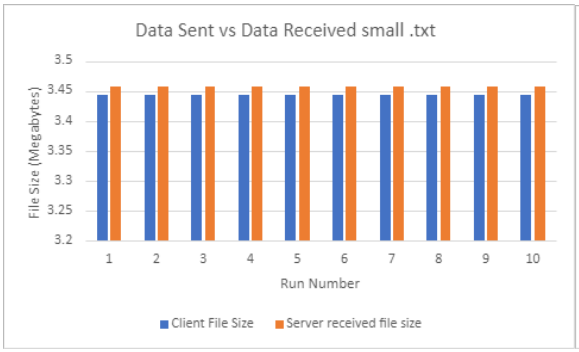


Figure 12. Graph of Data Sent vs Data Received MP4 Figure 13. Graph of Data Sent vs Data Received MP3

The first is Data Sent vs Received Data, which can be seen in Figures 12,13,14,15. In each graph, we can see that the data sent and received during each run remains consistent except in the case of figure 12. A reason for this anomaly may be due to how java encodes the message when it is sent across the network, adding a bit of overhead and size to each message. While this theory does stand with figure 12, it can't be seen in figure 13,14 and, 15. To further prove this theory we would need more testing.



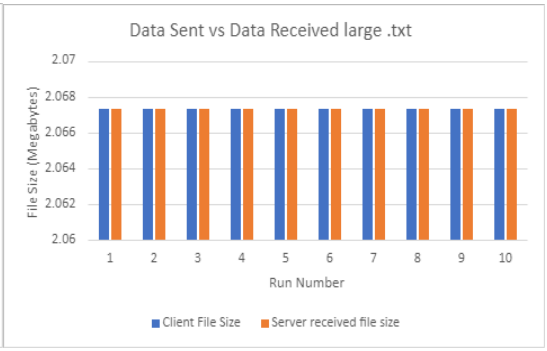
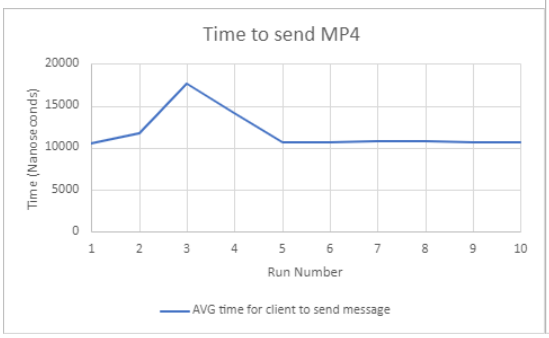


Figure 14. Graph of Data Sent vs Data Received small .txt Figure 15. Graph of Data Sent vs Data Received large .txt

The most interesting relationship out of the three was the total amount of time needed for the client to send a file. As we can see in figure 16,17,18, and 19, during each run, the time to send the file is significantly different.



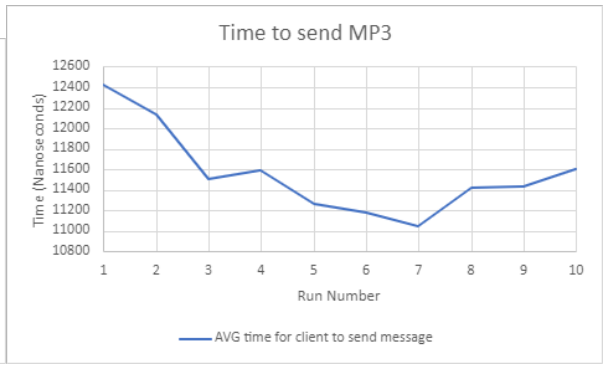
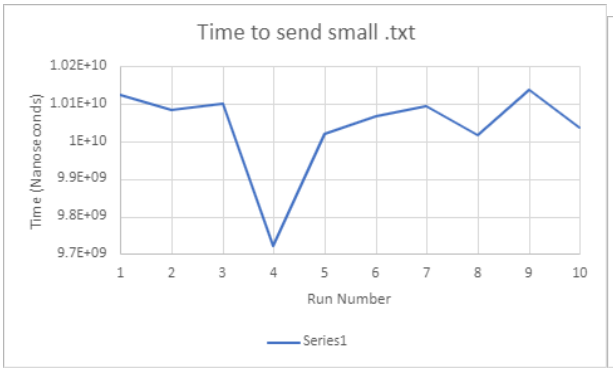


Figure 16. Time to send MP4 Figure 17. Time to send MP3



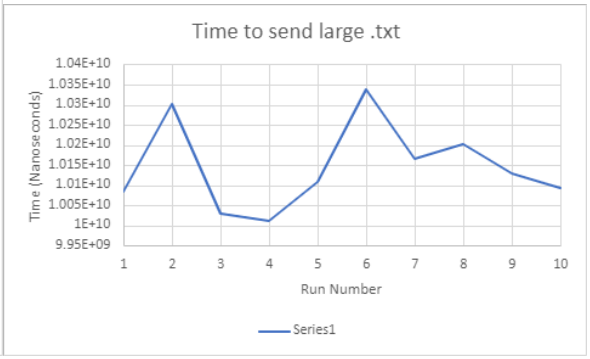


Figure 18.. Time to send small .txt Figure 19. Time to send large .txt

The last relationship is that of the average time needed to send a message, seen in figure 20 and 21. Like sending the entire file, the amount of time needed to send a message (one byte long) between the server and client is always changing and never consistent. Also, if one observes how the graph line moves for figure 17 and 16, it looks quite similar to how the bars move in figure 20 and 21. This lends credence to our theory that the file send time is related to the average message send time.

Figure 20. AVG Client Message Send Time vs AVG Server Message Send Time MP3 Figure 21. AVG Client Message Send Time vs AVG Server Message Send Time MP4

From the interlocking relationships of message send time and file send time, we can support a claim that we all observe in our daily lives: networked commutation is never consistent, and always fluctuates.

Conclusion

The final implementation of the distributed system allowed data for file sizes, processing time, and transmission time to be gathered from text, audio, and video files. Data streams were used for reading and writing bytes in transmission between clients and servers. Multiple runs were recorded for all measurements in order to obtain more accurate results. The file sizes were generally the same on both the client and server sides, which provides evidence that the file being sent was the same file that was received, with no missing bytes. The transmission and processing times for sending files between clients and servers were generally inconsistent across runs, which can be attributed to inconsistent internet speeds. The times between client sends and server sends however were generally very similar, meaning neither the client nor server considerably outperformed the other. Text file transmission time was in a similar time range between large and small file size, and audio/video files did not vary significantly either.

References

Java platform, Standard Edition Documentation - releases. Oracle Help Center. (2022, September 19). Retrieved February 13, 2023, from https://docs.oracle.com/en/java/javase/index.html

Bobbie, P. (2023) *p1+2-SThread, p1+2-TCPClient, p1+2-TCPServer, p1+2TCPServerRouter* Retrieved from: https://kennesaw.view.usg.edu/d2l/le/content/2793722/Home