Kennesaw State University

College of Computing and Software Engineering

Computer Science Department

CS 4504 Distributed Computing

**Phase 1 Report**

October 11h, 2017

Phoenix Alexander-Wright - [palexa15@students.kennesaw.edu](mailto:palexa15@students.kennesaw.edu)

William Zimmerman - [wzimmer7@students.kennesaw.edu](mailto:wzimmer7@students.kennesaw.edu)

Reece Perry - [rperry36@students.kennesaw.edu](mailto:rperry36@students.kennesaw.edu)

Cassidy Caruso - [jcaruso6@students.kennesaw.edu](mailto:jcaruso6@students.kennesaw.edu)

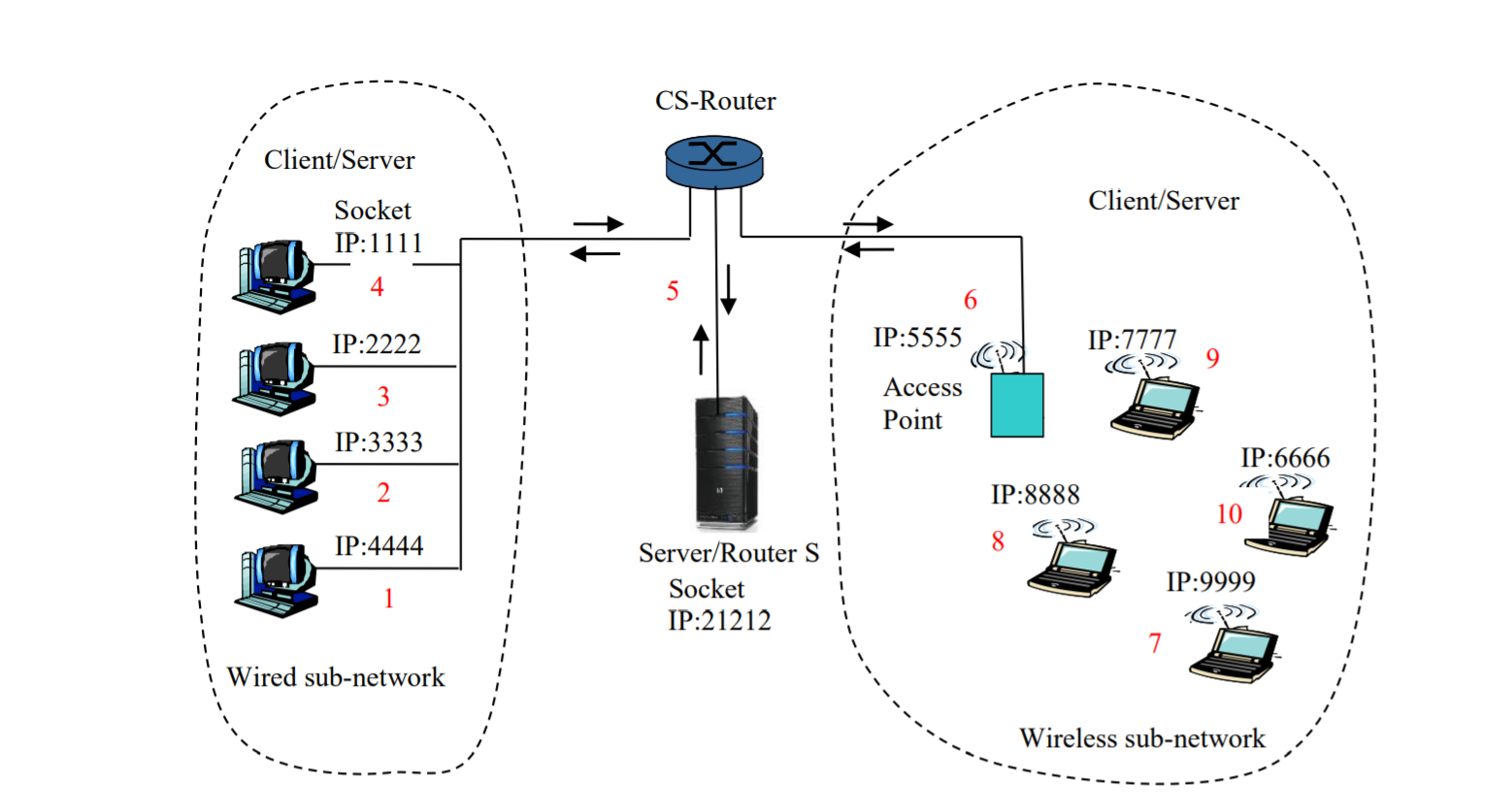
Gilberto Rose Jr. - [grose3@students.kennesaw.edu](mailto:grose3@students.kennesaw.edu)

# Abstract

In phase 1 of the project, our group utilized the Computer Science router in the CS lab to form an inter-connecting network using the CS router and a Server/Router. The purpose of this experiment was to familiarize ourselves with sockets and the interaction between routers and clients.

The client, our laptop, made a request through the CS router to the designated server/router. The server/router looked up the destination i.p. address of the node. This lookup and connection time is measured in nanoseconds. The node contained text which was then sent back and measured for time versus character count. The connection time was repeated for two texts of differing in size.

An illustration of the network is shown below:



# Introduction

In phase one we set out to complete and analyse the code that was given to us as well as run tests and collect data. The given code was mostly functional and needed only one small change to work properly. We analysed the code so that we understood how everything worked. After analysing the code, we then conducted tests with different text files that were sent from client to server. Each module was run on a separate laptop and were connected via wifi on the KSU network. We used two files where one consisted of lorem ipsum text and another bigger file with *John Stuart Mill; His Life and Works by H.R. Fox Bourne et al* text. The goal of this project was to understand the workings of a simple distributed system, which we successfully achieved.

# Table of Contents

[Abstract](#_d0i3n0n9m2kv)

[Introduction](#_v0u7xr5oa9oh)

[Table of Contents](#_6ne1gm758iru)

[How it works](#_k6nbk1958zf7)

[What We Added](#_lnw9tb596xpc)

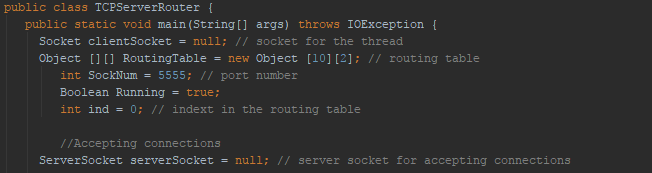
[Data](#_1ng1phowk705)

[Analysis](#_nee57hlzjgvt)

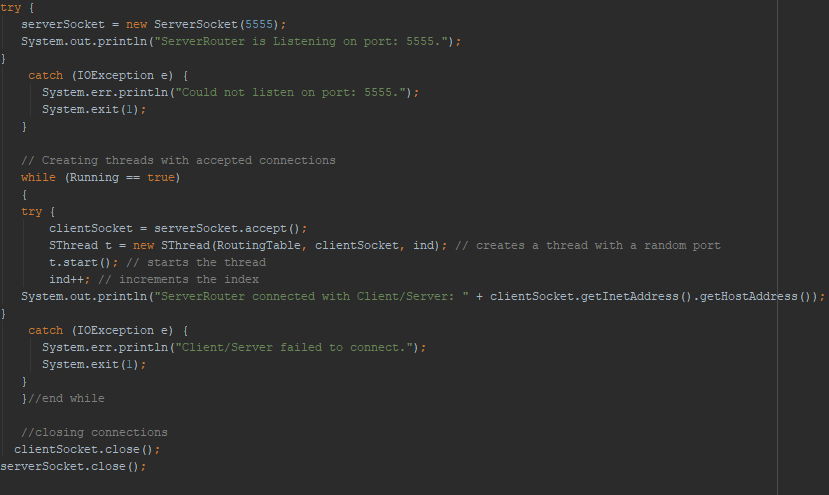
[Conclusion](#_lwcx8vgw03vg)

# How it works

***TCPServerRouter***

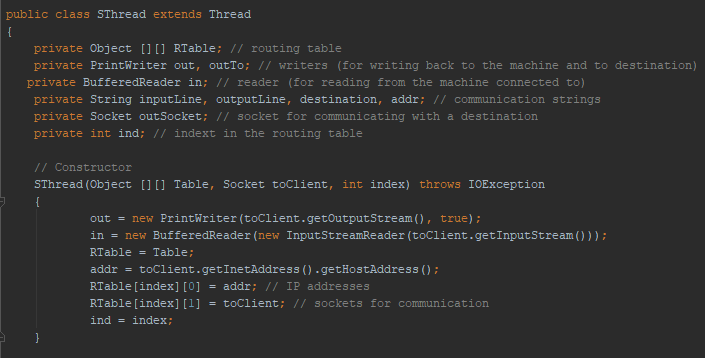


The beginning of the *TCPServerRouter* class creates the necessary objects that would be required to connect to a server and a client, that is a necessary socket to direct traffic, a routing table to keep client information, a client socket for receiving client information, and a Boolean and index for running information.

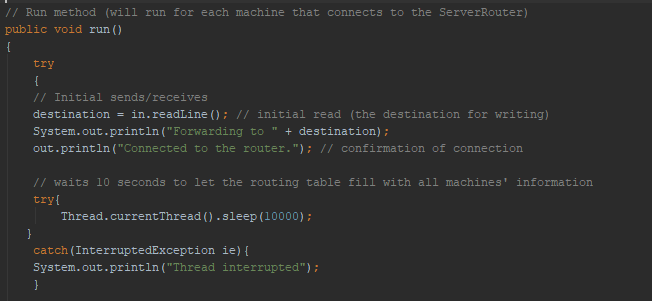


The beginning of the class lets the user know that a new server socket is created on port 5555. When this is met a while loop is used to continue running the threads that are created. These threads utilize the *SThread* class where the transfer of data occurs from client to server and vice versa.

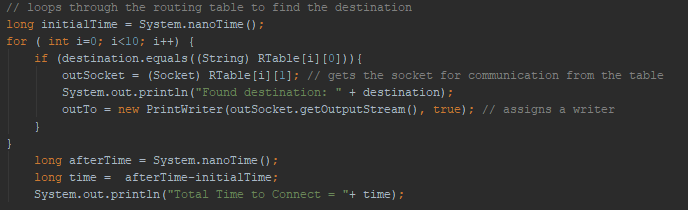
***SThread***



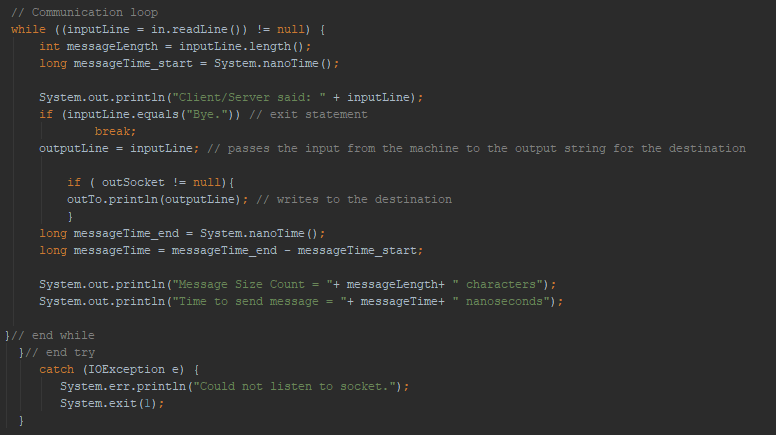
The beginning of the *SThread* class creates the necessary objects to be able to read a text file and write text. It also creates the socket and routing table for use in the created constructor that is used in the *TCPServerRouter* class.



The run method is necessary since we created a thread class, by nature of Thread, a run method is empty and must be filled with necessary logic to be able to run when called. This run method is called in the server router class once the initial thread is created.

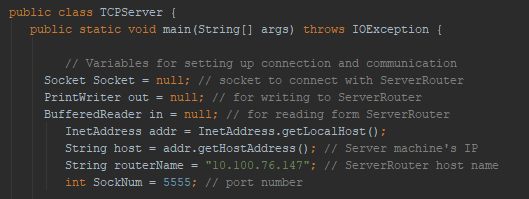


The above is located within the run method and contains the routing table which holds the destination information on where to send converted information.

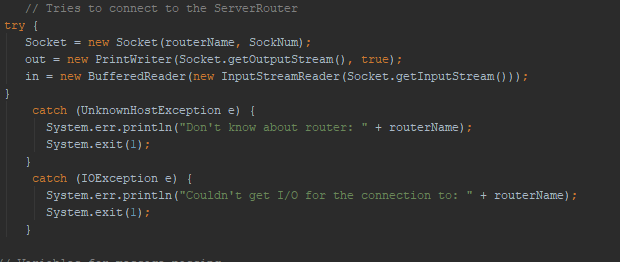


The above passes what is read to wherever the destination is. After the message has been sent, the server prints “Bye” to notify the user that it is finished.

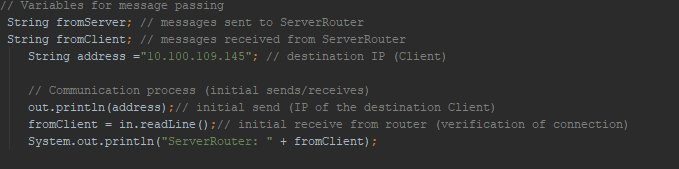
***TCPServer***



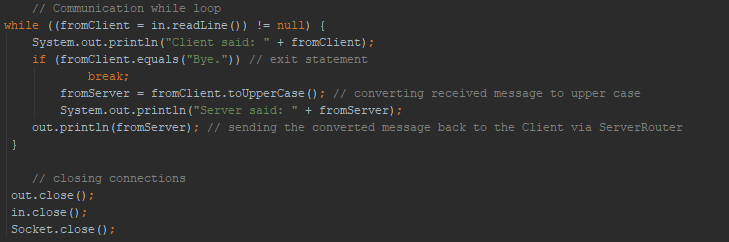
Contains the necessary objects to communicate, such as the socket, host, and router name, as well as the reader and writer for the text file.



The above tries to connect to the Server router based off the IP address provided. It then reads whatever is sent in the socket, and returns two different exceptions: it could connect, or the router IP was invalid.

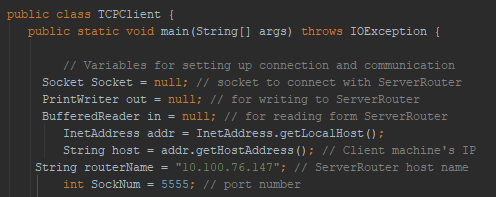


The following code segment defines objects for both the message from the client and the new message that the server creates. It also has the initial communication to the client, and the actual message is read and stored here in the *fromClient* object.

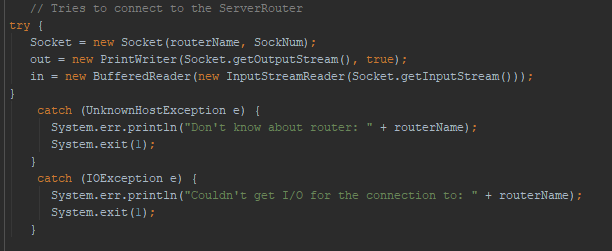


The above reads each line from the client and then converts the message to upper case and sets that as the new *fromServer* object. This is then sent out via the socket output stream. Once the sending is complete, the connections are closed.

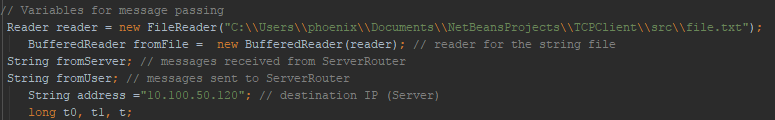
***TCPClient***



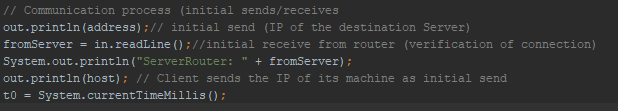
Much like the server class, the client class starts by defining similar objects to specify the router, the socket, and the reader and writer.



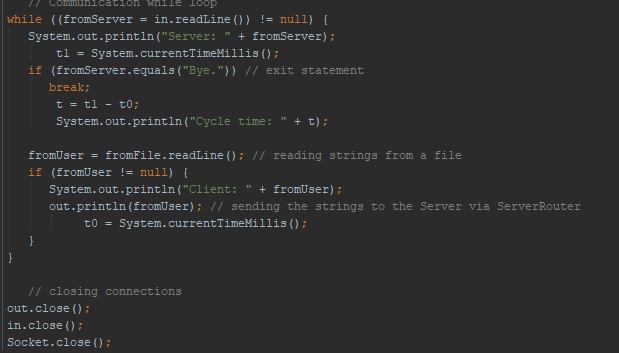
The above is identical to the server code, as it tries to connect to the router and has two exceptions in case it cannot successfully connect. It also reads from the socket input stream and defines where it writes to for the output stream.



Here is where the code becomes different, as the client reads a text file from a specified location and then puts that into a buffered reader. It also defines: the message from the server, the user, and the destination IP.



Here is where the actual communication takes place. As it first connects to the IP of the destination router, it then receives some confirmation message from the router which is saved in the *fromServer* string. This is then put in the console to tell the user the connection is good. Then the client sends its current IP, so the router can save it in it’s routing table for later.

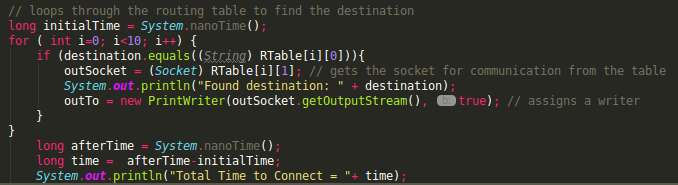


Above is where the actual message is sent to the *TCPServerRouter.* It is done much like in the server class, as it reads from the *fromUser* string that was created before, and sends the strings over the server via the output stream socket. Then when finished, we receive a “Bye” which is our exit statement that breaks the while loop. This causes the next step to close all socket connections.

# What We Added

We added code to ***SThread.java*** for the purpose of gathering metric data.

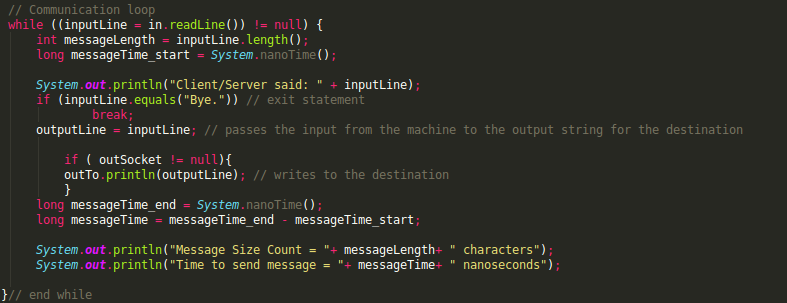
We modified the following section of code by adding a timer to measure how long it takes to loop through the routing table and establish each connection.

*initialTime* holds the time in nanoseconds just before a connection is initiated.

*afterTime* holds the time in nanoseconds just after a connection has been made and the loops exits.

The difference of *afterTime* and *initialTime* is stored in *time* and printed as output. This gives us our “Total Time to Connect” metric.

We modified this next section of code by adding a counter to measure the size of each message and a timer to measure the time it takes to send each message.



*messageLength* stores the length of a message via the java.lang.String.length() method.

*messageTime\_start* stores the time in nanoseconds just before a communication takes place.

*messageTime\_end* stores the time in nanoseconds just after message has been sent.

The difference of *messageTime\_end* and *messageTime\_start* is stored in *messageTime*.

Both *messageLength* and *messageTime* are then printed as output. This gives us the “Message Size Count” and “Time to send message” metrics respectively.

# Data

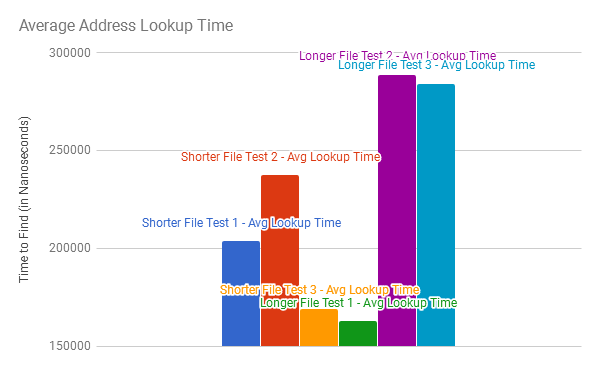
****

Figure 1-1 - Overall Average Address Lookup Time

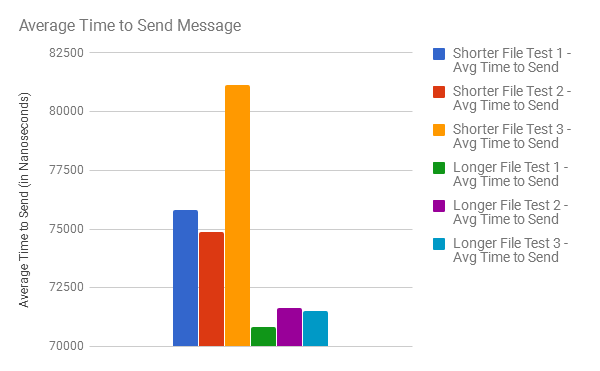


Figure 1-2 - Average Time to Send Message

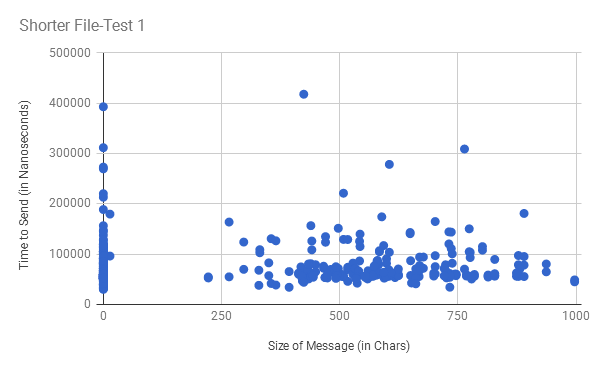
****

Figure 2-1 - Size of message vs time for short file test 1

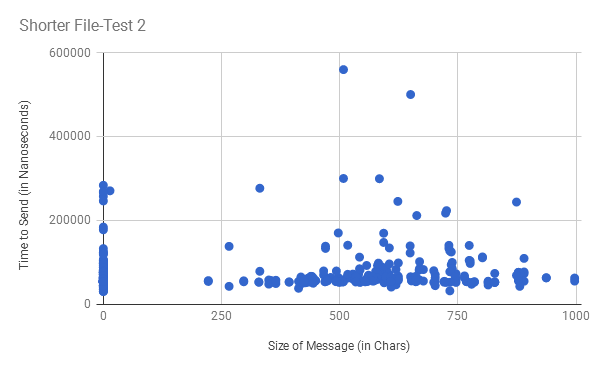
****

Figure 2-2 - Size of message vs time for short file test 2

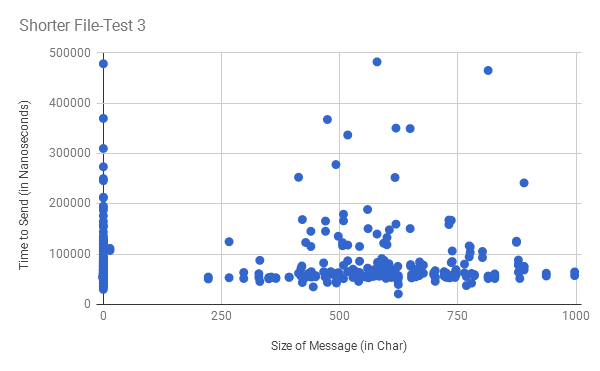
****

Figure 2-3 - Size of message vs time for short file test 3

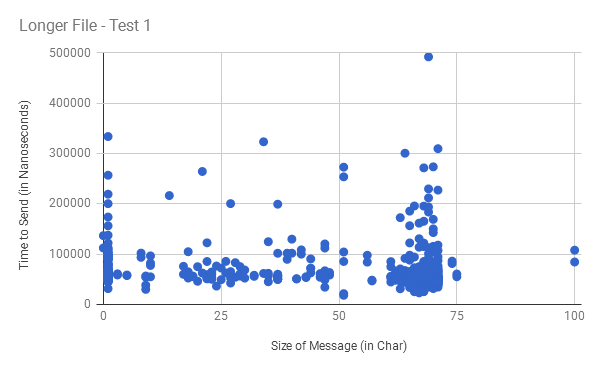
****

Figure 3-1 - Size of message vs time for longer file test 1

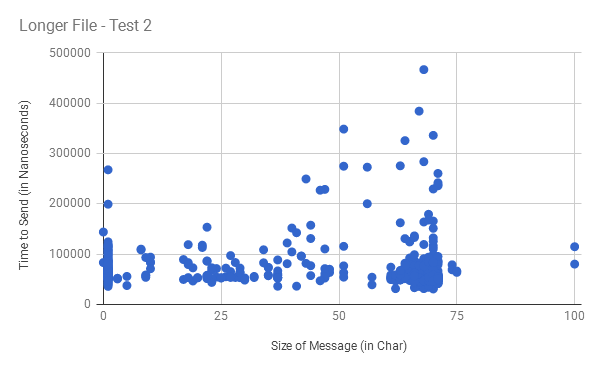
****

Figure 3-2 - Size of message vs time for longer file test 2

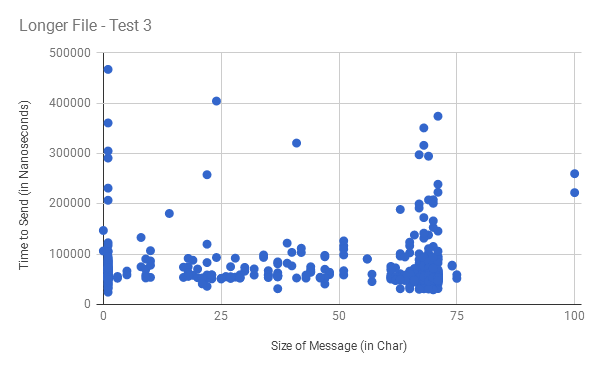
****

Figure 3-3 - Size of message vs time for longer file test 3

# Analysis

It should be noted that with our data the shorter size file, file short.text, has a size of 67 KB but the average number of characters per line is 299.5707965 while the larger size file, John Stuart Mill; His Life and Works by H. R. Fox Bourne et al.rtf, has a file size of 163 KB, but a normal amount of characters per line that you would find in a book. We wanted to not only see how size of a file relates to the lookup time and the average time to send but also how the characters per line affected the data.

During our testing we used multiple laptops and the wireless connection over the school network. We found that when using a MacOS computer the program did not work properly, but when using either linux or windows systems the program worked flawlessly.

We also found that even though some lines have a character count of 0, they took significantly longer to process than character counts that were larger such as 50 - 75 in size, this trend can be seen in all of our graphs.

It seems though according to Figure 1-2 there is a direct relationship between the number of characters per line and how long it took for the message to be sent. This makes sense as the program works by reading the file and send the message line by line. What is interesting though is when looking at the average lookup-time for both files. Figure 1-1 shows that there is not a definite relationship that we can see when comparing the lookup time to the size of the message or the character’s per line. In two of the tests we have the the lookup time for the shorter and larger file be very different from what is expected. The third test for the short file has a very short lookup time compared to the former tests. Then for the larger file, the first test has a very short lookup time when compared to the last two tests as well as the shortest time for the smaller file which is most interesting. The two explanations we have for this is that for some reason the socket was not properly flushed when the command was called except for the sequential short test. There could also be some sort of background process on a system that was causing slowdown or the network we were on caused slowness.

# Conclusion

We were able to prove the files were sent between router and client by recording the time sent versus the character count of those files. We also successfully altered the files from one size text to the other.

The data in the graphs presented as expected. The look up times were not consistent, for the short file the lookup times varied heavily between the three runs. This was also true for the long file. This is expected since the lookup times only involved the lookup of the i.p. address. Activity involving unrelated processes will result in these time variations.

The average time to send messages was also unexpected. The time needed to send a short message took longer to send a long message. The reason for this is along with our message other clients running other processes. Other node points routing other clients not related to our network will tie up said node. When these test were being ran, other computers were also using the CS router. This will contribute to varying times of sending messages.