Kennesaw State University

College of Computing and Software Engineering

Computer Science Department

CS 4504 Distributed Computing

**Phase 2 Report**

November 25th, 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)

# Table of Contents

[**Table of Contents**](#_hwwtus6sc473) **1**

[**Abstract**](#_6hopv47ehae8) **2**

[**Introduction**](#_s1rmm88ztqlq) **3**

[**Simulation/How it Works**](#_q0ctoeazp1rq) **4**

[**Implementation/What We Added**](#_82e53eq2y409) **11**

[**Data**](#_762ac5lcl67q) **13**

[**Analysis**](#_7nuj3mwv4gq9) **40**

[Receiver Side:](#_rygxrmw18l6y) 40

[Finished time vs Bytes of Data](#_qqpg83ihqe1n) 40

[Bytes per Time vs Finished Time](#_v3nc8j4mydy7) 41

[Sender Side:](#_ifxbcmljzqiu) 42

[Finished time vs Bytes of Data](#_x452rl6ip1dl) 42

[Bytes per Time vs Finished Time](#_4ho6jaf3oygr) 42

[Server Router:](#_t410cgw33bte) 43

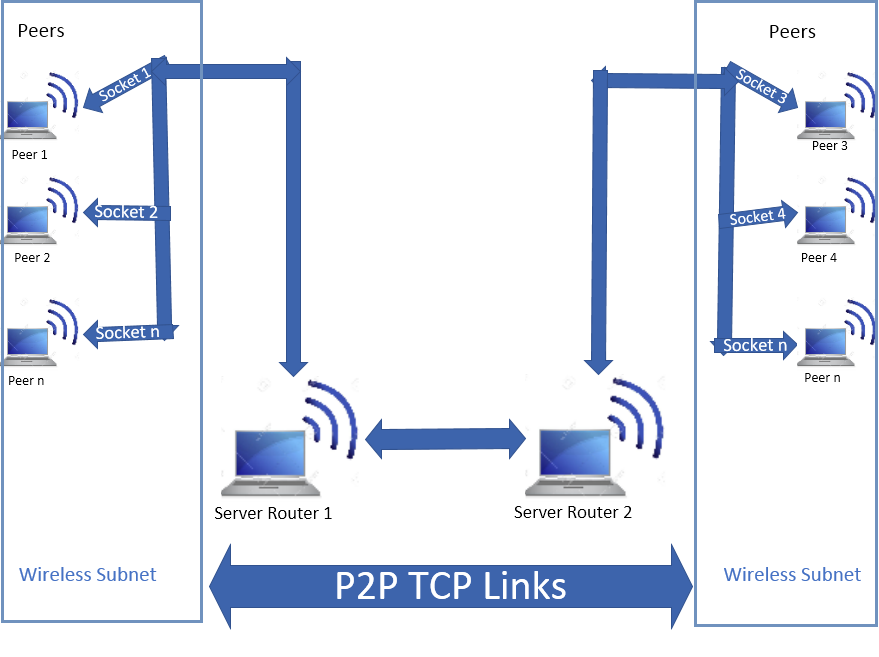
[Average Server Router Lookup times](#_cr3f6uvq19ve) 43

[**Conclusion**](#_phjloezfjmc1) **44**

[**Appendix**](#_bg3g4p8jrmwv) **45**

# Abstract

In phase two of the project, our objective was to use the system from phase one to design and implement a peer-to-peer(P2P) system. The purpose of this experiment was to familiarize ourselves with how peers communicate in a peer-to-peer network. A diagram of the architecture used is pictured below.



One peer sends either a .txt, .mp3, or .mp4 file to another peer using multiple threads. Each peer is registered to a random server router. The sending peer will query the server router that it is registered to to look for the receiving peer’s socket information. If the socket information is not found in that server router, a remote method is called which searches the other server router. The server routers act as super-peers and simply communicate socket information from one peer to another. Once the peers know each other's socket information, they can directly connect via TCP/IP in order to exchange files.

Message sizes, transmission times, time for the routing-table lookups, and number of bytes of data transferred per nanosecond are all recorded as metrics and written to a .csv file. The experiment was conducted 12 times in total, using each file type with 10, 20, 30, and 100 threads. The experiments were all conducted over a Virtual Private Network to avoid issues such as port forwarding and also because we were unable to physically meet over the autumn break.

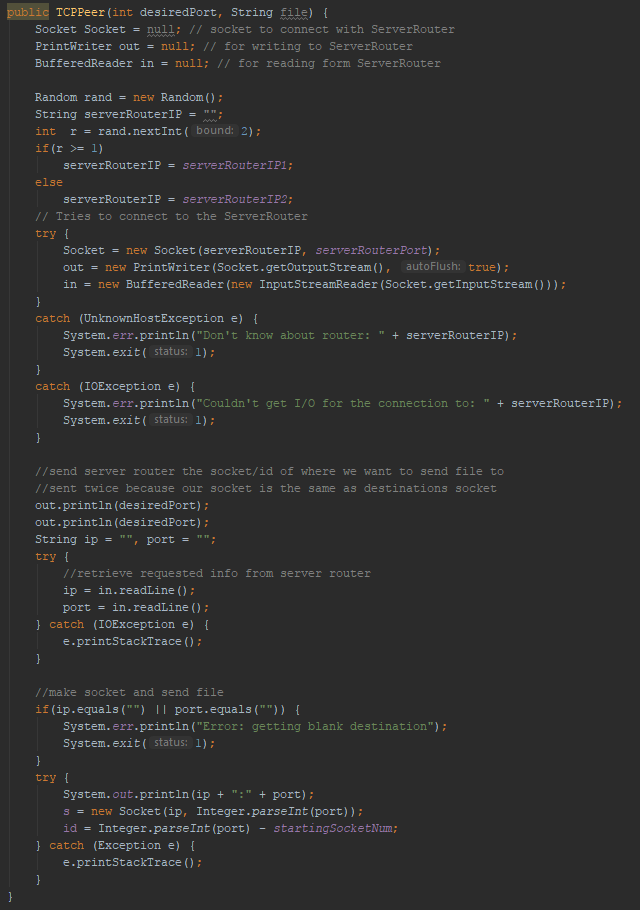
# Introduction

In phase two, we set out to modify our existing code from phase one into a peer-to-peer style network. The client and server from phase one have been replaced withpeers. Peers can receive files from other peers, and also send files to other peers. A second server router and functionality allowing the two server routers to communicate via Remote Method Invocation was also added. In an attempt to simulate a real world environment, each peer connects to a random server router.

After debugging our modifications, we conducted tests with .txt, .mp3, and .mp4 files that were sent from one peer to another peer with varying amounts of threads. The computers were all in separate physical locations and connected to each other via a Virtual Private Network.

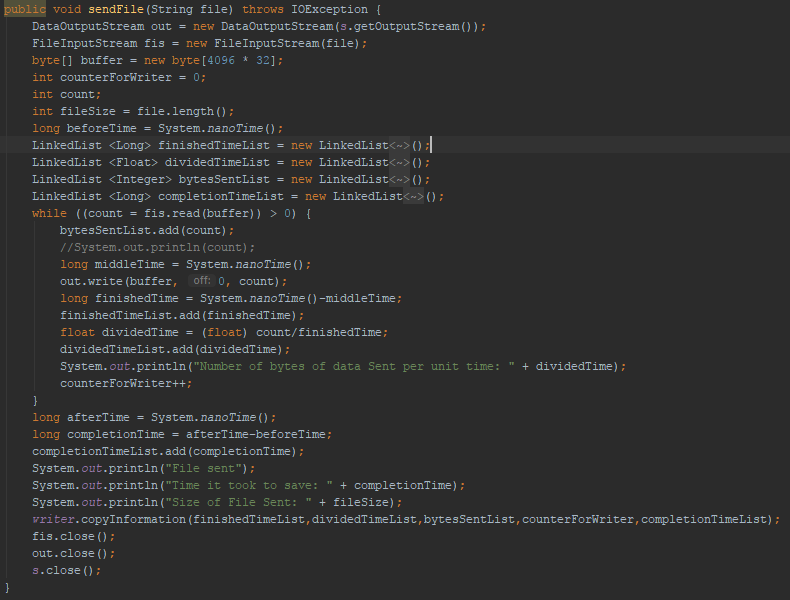
# Simulation/How it Works

**The Client Software:**

Once both of the *TCPServerRouters* are activated, two *TCPPeers* are activated on each end of the network. One *TCPPeer* will take on the role of receiver and the other takes on the role of sender. These roles are triggered by entering the keywords "Client" or  
“Server.”  
 

Code Figure 1-1: TCPPeer Class

Figure 1-1 shows the *TCPPeer (*client constructor)class which chooses one of two IP addresses corresponding to the server routers. This is done randomly to be more like a real life scenario. The *TCPPeer* class then sends the server router the socket id so that the server router can lookup the correct port and ip. The client waits for the server router to return the port and ip.

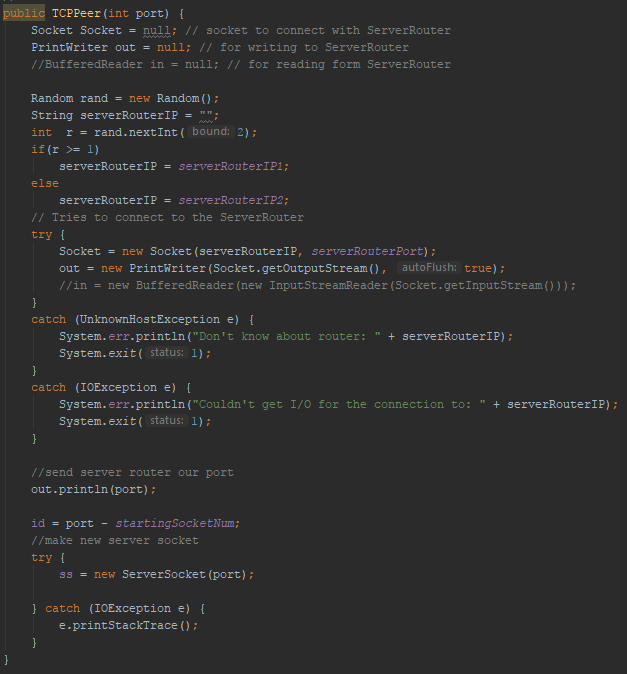


Code Figure 1-2: TCPPeer sendFile Method

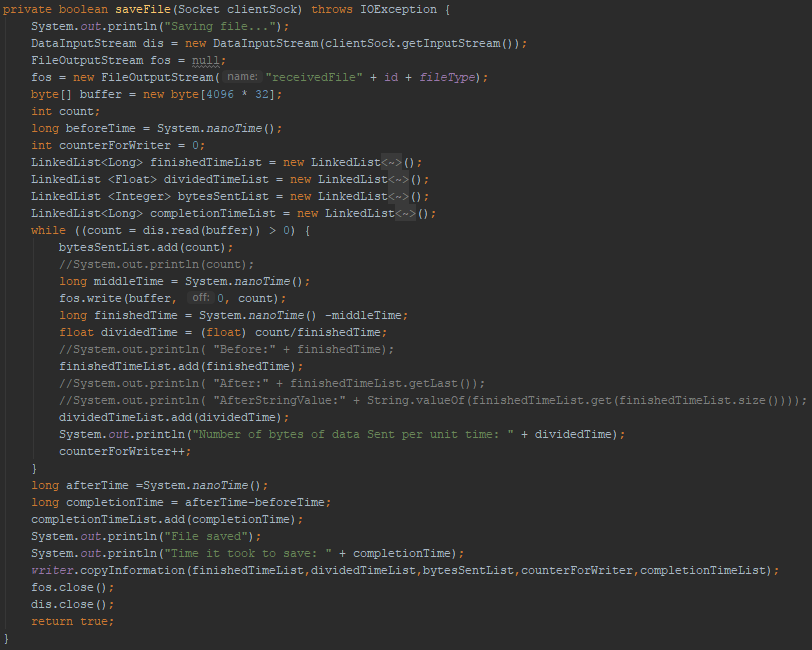
After the client has received the necessary information from the server router, it will begin to send the file over the created socket(Figure 1-2). This is done by writing to a buffer and sending in small increments until there is no more data to write. After the file is finished sending, the thread will have ended.

**The Server Software:**

The server instance of the *TCPPeer* class behaves similarly to the client instance, however their functionality is reversed. The server receives the file that the client is sending.

****Code Figure 2-1: TCPPeer Server Constructor

When the server instance is initialized and the constructor is called(Figure 2-1), it connects to a random server router and sends that server router the port that it will be listening on. Then, the server instance waits to accept a connection from the client.

****

Code Figure 2-2: TCPPeer saveFile Method

When a connection is accepted from the client, the server instance begins to save the file(Figure 2-2). It does this via retrieving from a buffer until that buffer is empty. All of the code regarding LinkedLists is used for gathering statistics and serves no functional purpose in regards to communication.

**TCP Server Router software:**The *TCPServerRouter* class is what acts as the super-peer in our system. Two instances of *TCPServerRouter* communicate with one another to retrieve socket information from peers wishing to receive a file and deliver it to peers wishing to send that file.

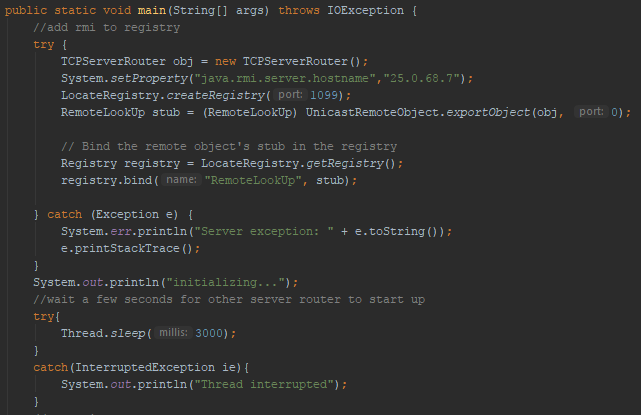
****Code Figure 3-1: TCPServerRouter main Method

Figure 3-1 shows that the first thing that is done is to register our remote method with our local registry. This allows the other server router to find our remote method.

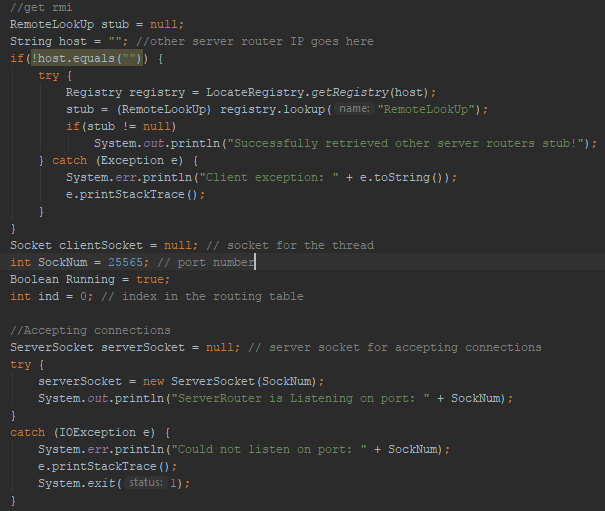
****Code Figure 3-2: TCPServerRouter main Method

Figure 3-2 shows how we retrieve the other TCPServerRouter’s RMI. We find its registry and download the stub. After we get the stub, we begin to accept connections from peers.

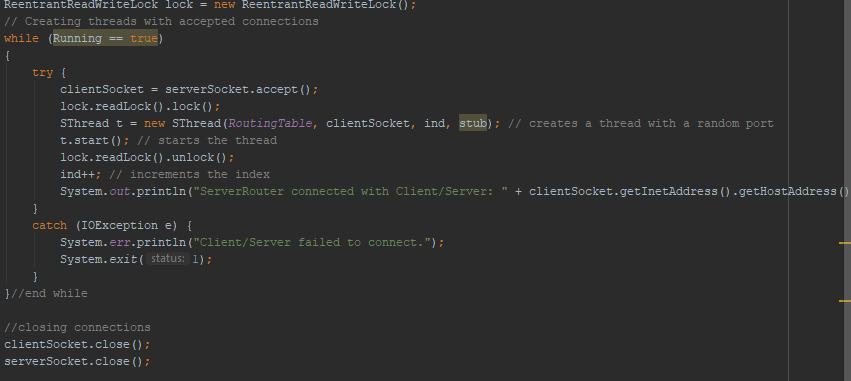
****Code Figure 3-3: TCPServerRouter main Method

Figure 3.3 is where we are creating SThreads for every new connection that we accept from a peer. We give each thread a reference to the stub because the RMI will be invoked in the thread A lock is also used so that synchronization is maintained.

**LogMeIn Hamachi (VPN):**

To avoid problems such as port forwarding which complicates the project, a Virtual Private Network was used. The VPN utilizes the TCP protocol to identify devices on the network.

# 

# 

# Implementation/What We Added

TCPPeer

We recreated the TCPClient and TCPServer server classes into one class named *TCPPeer*. This allows for a truly modular system in which the *TCPPeer* starts by making a query to the user that is used to determine if it should act as a client or a server.

To begin, the program creates the number of threads that is declared in the specification that is defined by the user. Then, with each thread it runs through either the client or server version.

For the client version, it tries to connect to the respective *TCPServerRouter* specified and then creates a socket and receives the output from the socket which is then read into the buffered reader. Once that occurs, it sends the socket and destination IP to the *TCPServerRouter*. Once it is given the correct socket and ip from the *TCPServerRouter* a connection is made and data is sent.

For the server version much is the same as the client, but now the threads created randomly connect to either server router to simulate a real life situation. With this the server version receives code from a server router and saves the file to the local machine that is running it.

TCPServerRouter

*TCPServerRouter* didn’t have any functionality removed and performs the same function. We added RMI functionality so that two server routers can communicate with one another. A remote interface called *RemoteLookUp* was created. It contains one method called *getSocket* which searches though the routing table on the server router instance and returns what it finds to the server router that called the RMI.

At the beginning of *TCPServerRouter*, code was added to set up the RMI functionality. First each server router registers their instance of the RMI with their registry. After that, each server router then tries to get the stub from one another. When a SThread is created, we provide it with the RMI stub so that it is able to call the remote method.

SThread

SThread’s functionality also didn’t change much. One change that was made is how we populate the routing table. Strings are stored instead of sockets, and the socket information is given at the beginning of the run method instead of in the constructor. This is because the socket given in the constructor is not the same one we want to store in the routing table.

The other change that was made to *SThread* is that if it doesn’t find the socket it’s looking for in its routing table, then it invokes the RMI to search the other routing table.

# 

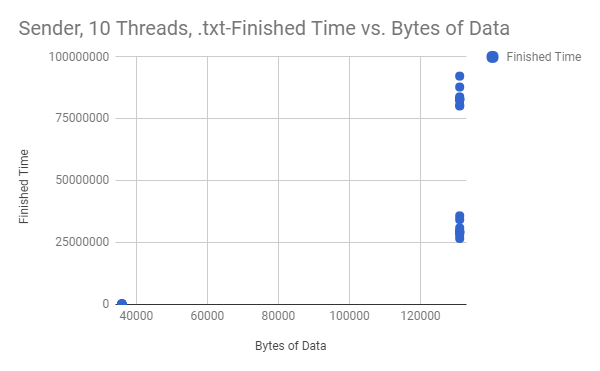
# 

# Data

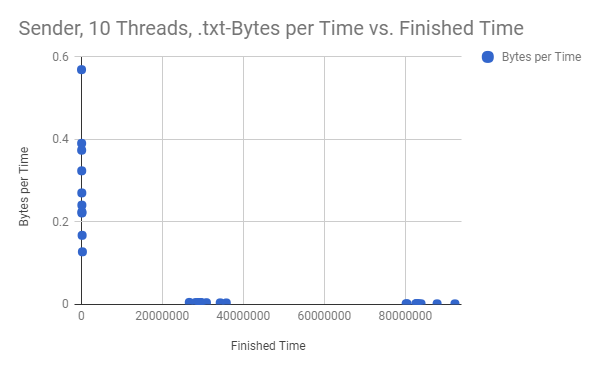
All time data is in Nanoseconds

|  |  |  |
| --- | --- | --- |
| Average Server Router Look-up Times | | |
| Destination Location | In own routing table | On other Server-Router |
| 10 Threads, .txt | 34661.5 | 14992.4 |
| 10 Threads, .mp3 | 4472 | 12406.2 |
| 10 Threads, .mp4 | 10394.1667 | 14110.25 |
| 20 Threads, .txt | 9817.1111 | 12396.4444 |
| 20 Threads, .mp3 | 8378.6667 | 10337.1 |
| 20 Threads, .mp4 | 17417.9167 | 10635.5 |
| 30 Threads, .txt | 8547.2308 | 9386.1333 |
| 30 Threads, .mp3 | 9772.2727 | 189931.2632 |
| 30 Threads, .mp4 | 8665.2308 | 8268.8235 |
| 100 Threads, .txt | 4328.0741 | 5711.6522 |
| 100 Threads, .mp3 | 9536.8636 | 8018.5574 |
| 100 Threads, .mp4 | 6781.6275 | 9442.0444 |
| Overall Average | 11064.3884 | 25469.6974 |

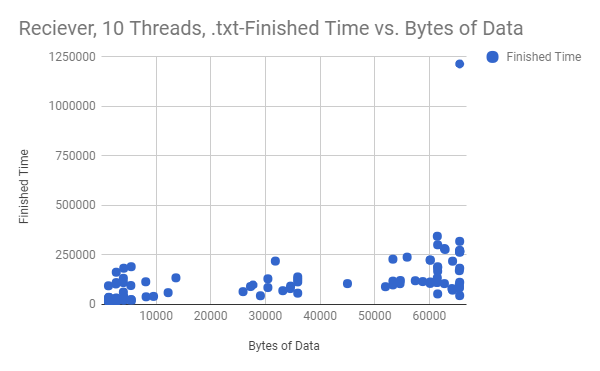
*Table 1.1- The Average Routing Table Look-up Times*

**

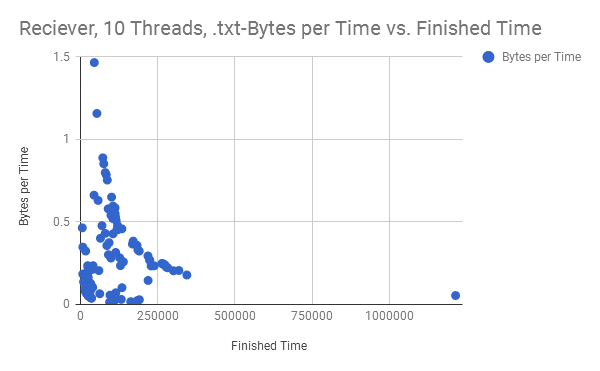
*Figure 1.1 - Comparison of Time Required to Send and the Size of the Data Sent*

**

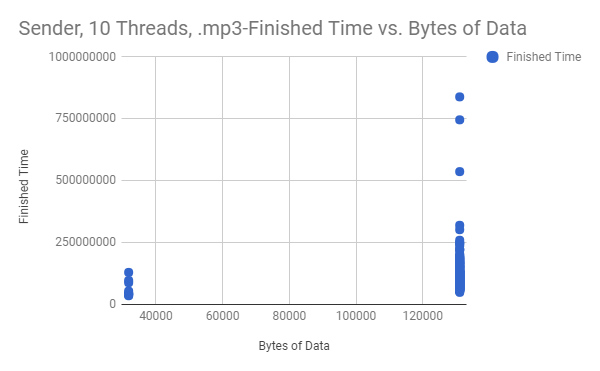
*Figure 1.2 - Comparison of the Speed at which Data is Sent and the Time it took to Send the Data*

**

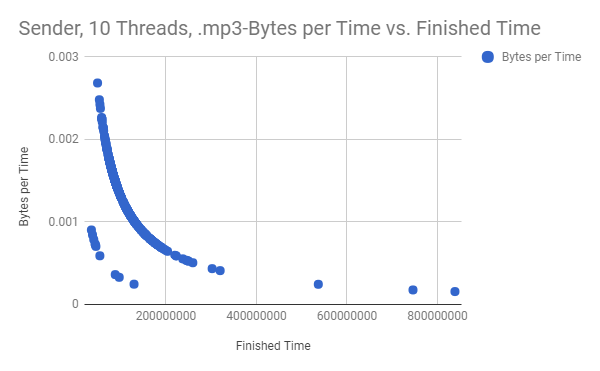
*Figure 1.3 - Comparison of Time Required to Receive and the Size of the Data Received*

**

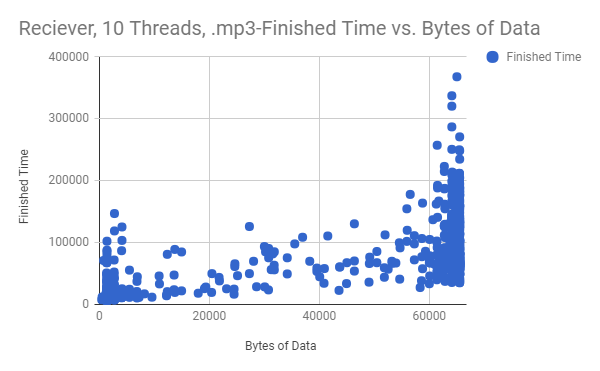
*Figure 1.4 - Comparison of the Speed at which Data is Received and the Time it took to Receive the Data*

**

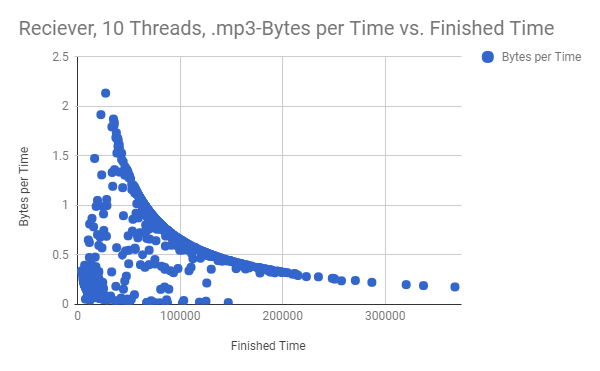
*Figure 2.1 - Comparison of the Time Required to Send and the Size of the Data Sent*

**

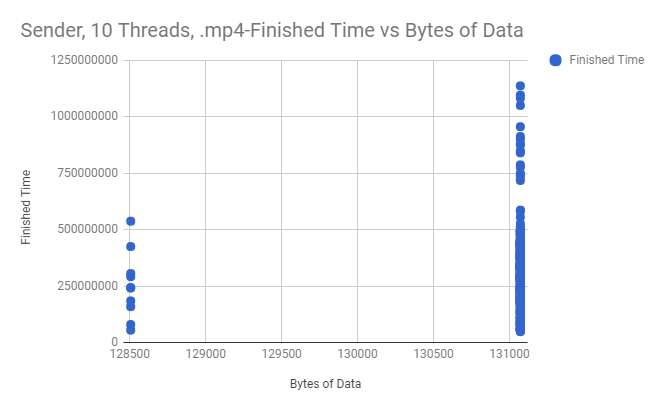
*Figure 2.2 - Comparison of the Speed at which Data is Sent and the Time it took to Send the Data*

**

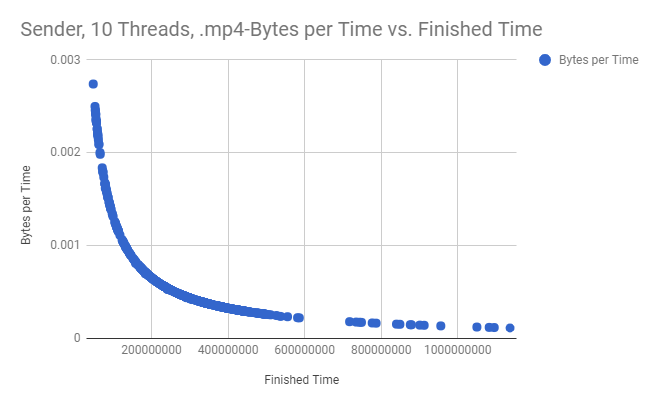
*Figure 2.3 - Comparison of Time Required to Receive and the Size of the Data Received*

**

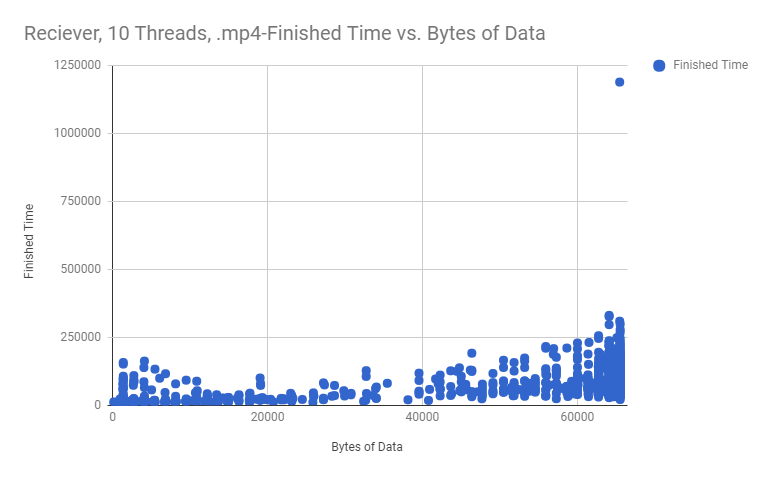
*Figure 2.4 - Comparison of the Speed at which Data is Received and the Time it took to Receive the Data*

**

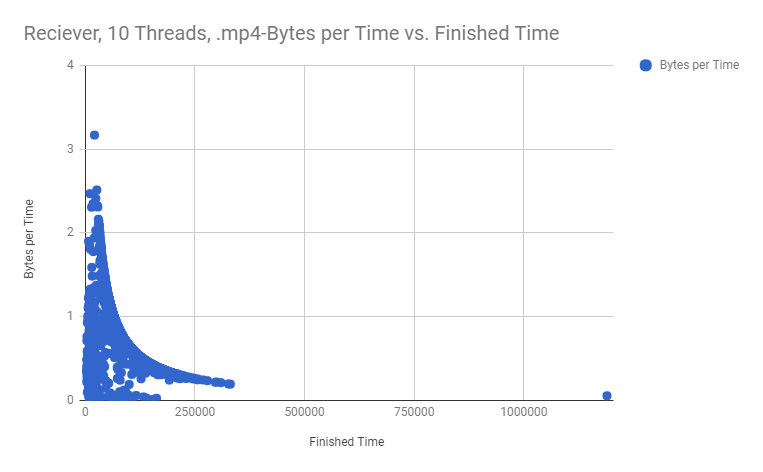
*Figure 3.1 - Comparison of the Time Required to Send and the Size of the Data Sent*

**

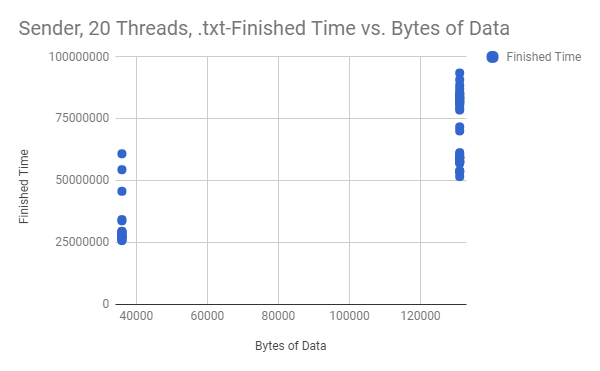
*Figure 3.2 - Comparison of the Speed at which Data is Sent and the Time it took to Send the Data*

**

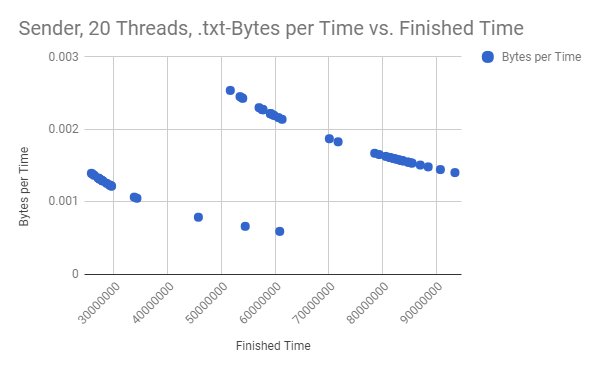
*Figure 3.3 - Comparison of Time Required to Receive and the Size of the Data Received*

**

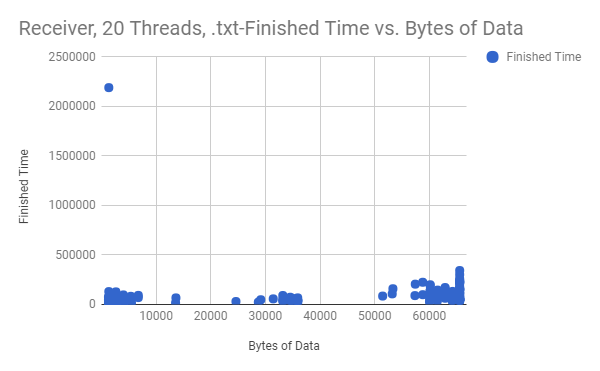
*Figure 3.4 - Comparison of the Speed at which Data is Received and the Time it took to Receive the Data*

**

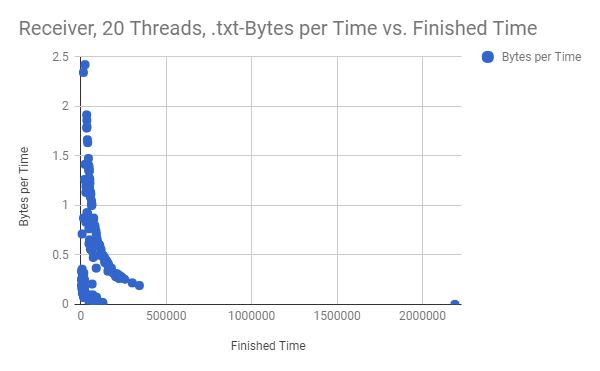
*Figure 4.1 - Comparison of the Time Required to Send and the Size of the Data Sent*

**

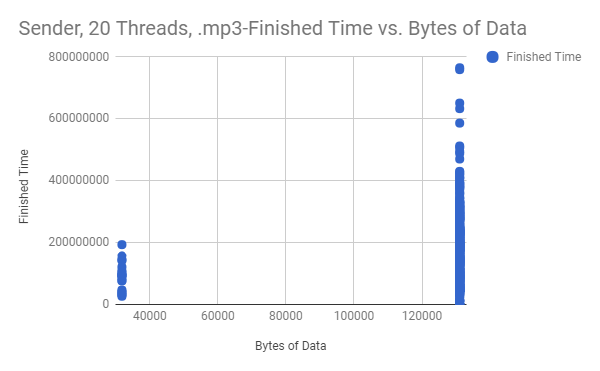
*Figure 4.2 - Comparison of the Speed at which Data is Sent and the Time it took to Send the Data*

**

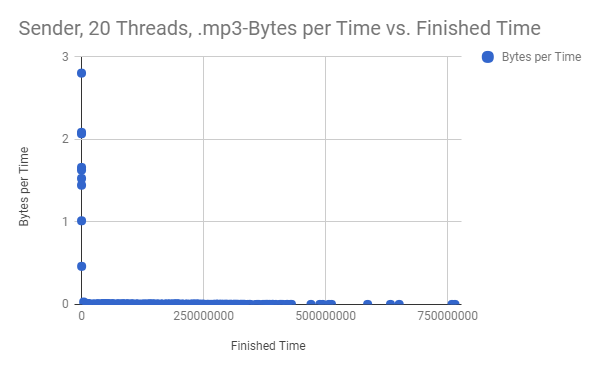
*Figure 4.3 - Comparison of Time Required to Receive and the Size of the Data Received*

**

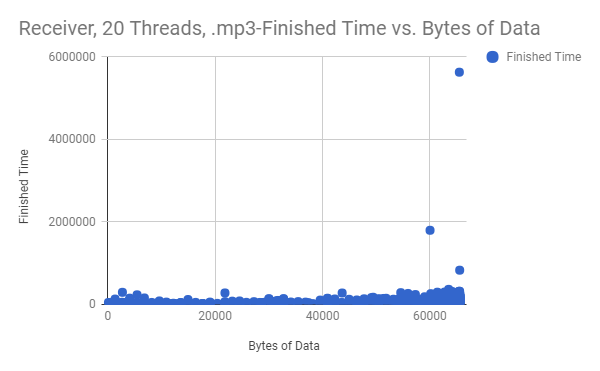
*Figure 4.4 - Comparison of the Speed at which Data is Received and the Time it took to Receive the Data*

**

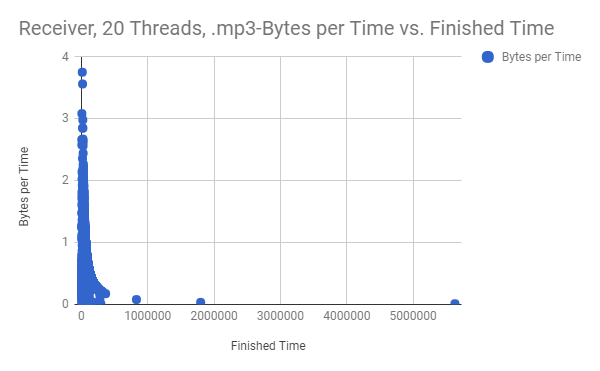
*Figure 5.1 - Comparison of the Time Required to Send and the Size of the Data Sent*

**

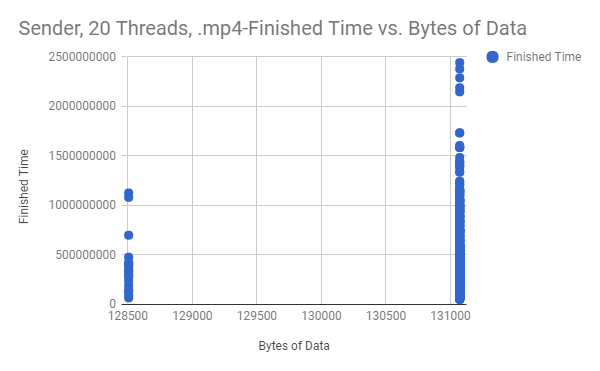
*Figure 5.2 - Comparison of the Speed at which Data is Sent and the Time it took to Send the Data*

**

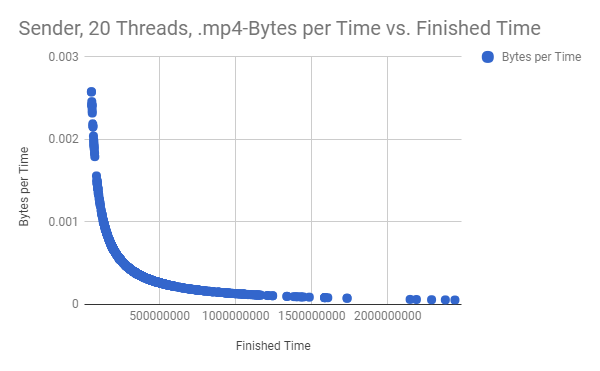
*Figure 5.3 - Comparison of Time Required to Receive and the Size of the Data Received*

**

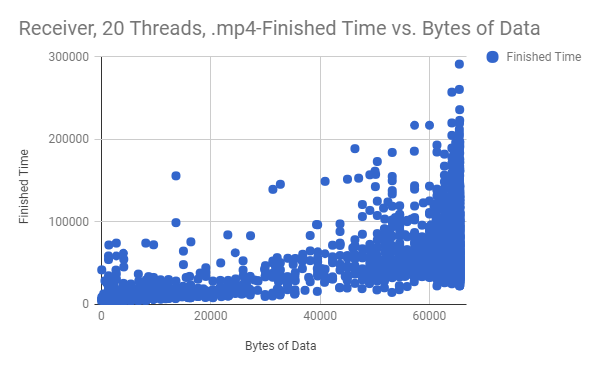
*Figure 5.4 - Comparison of the Speed at which Data is Received and the Time it took to Receive the Data*

**

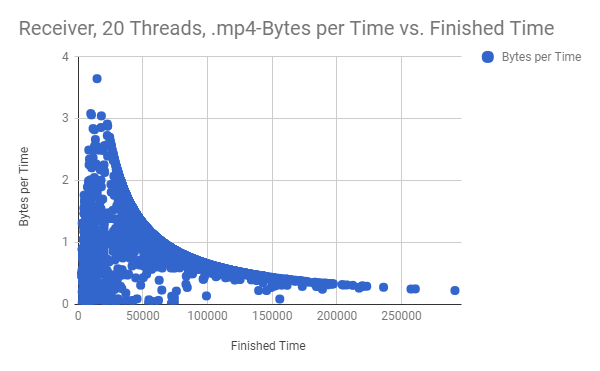
*Figure 6.1 - Comparison of the Time Required to Send and the Size of the Data Sent*

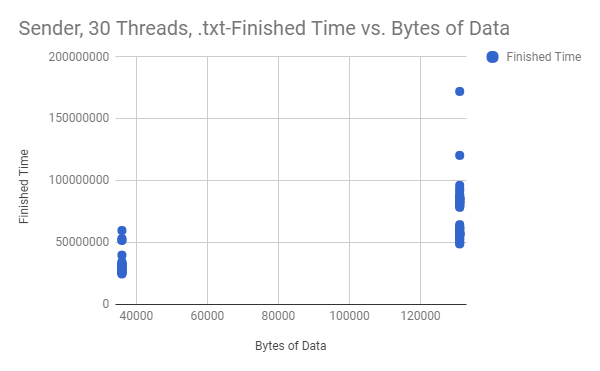
**

*Figure 6.2 - Comparison of the Speed at which Data is Sent and the Time it took to Send the Data*

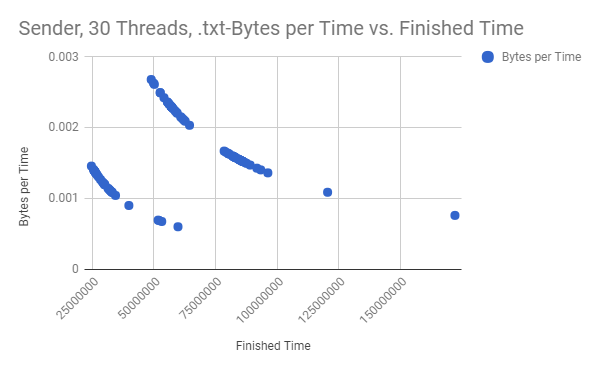
**

*Figure 6.3 - Comparison of Time Required to Receive and the Size of the Data Received*

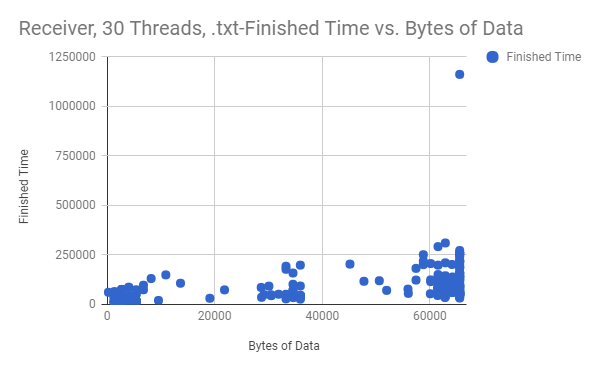
*Figure 6.4 - Comparison of the Speed at which Data is Received and the Time it took to Receive the Data*

**

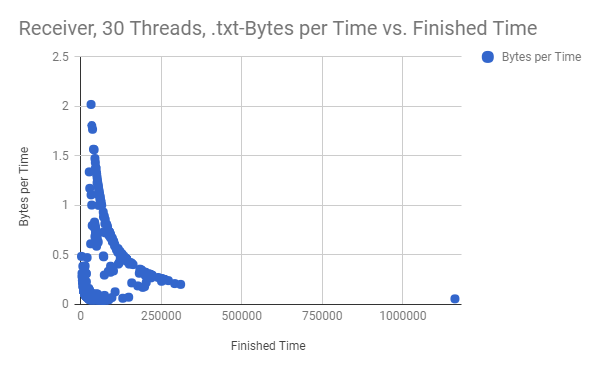
*Figure 7.1 - Comparison of the Time Required to Send and the Size of the Data Sent*

**

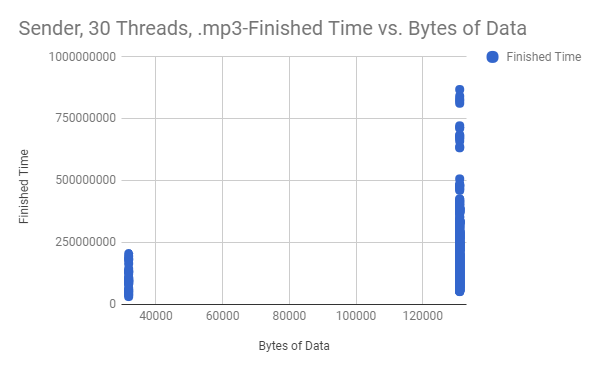
*Figure 7.2 - Comparison of the Speed at which Data is Sent and the Time it took to Send the Data*

**

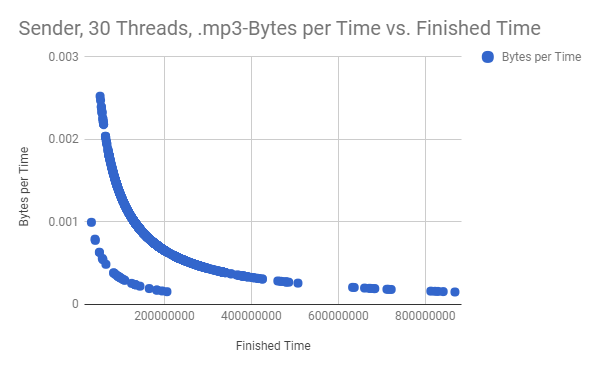
*Figure 7.3 - Comparison of Time Required to Receive and the Size of the Data Received*

**

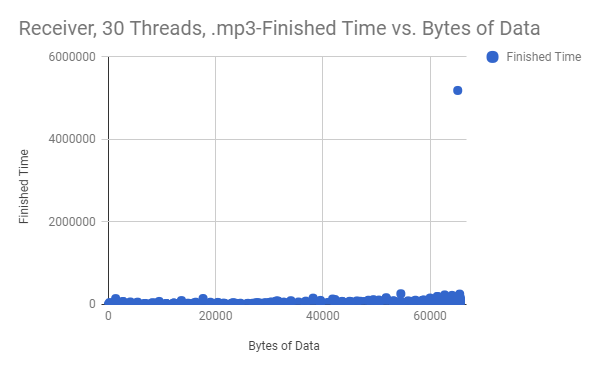
*Figure 7.4 - Comparison of the Speed at which Data is Received and the Time it took to Receive the Data*

**

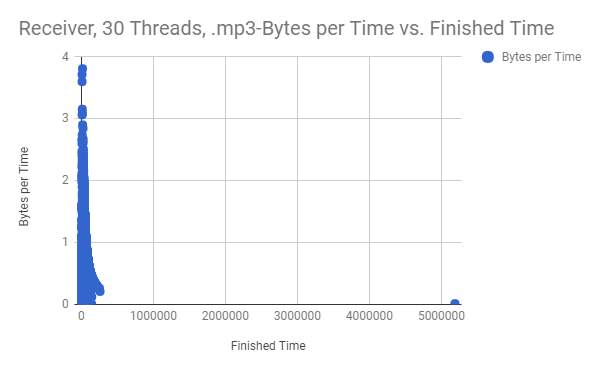
*Figure 8.1 - Comparison of the Time Required to Send and the Size of the Data Sent*

**

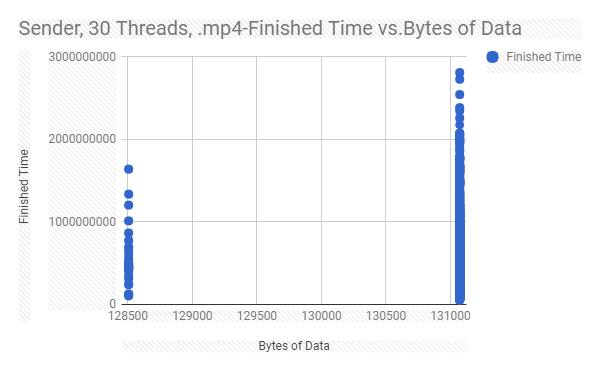
*Figure 8.2 - Comparison of the Speed at which Data is Sent and the Time it took to Send the Data*

**

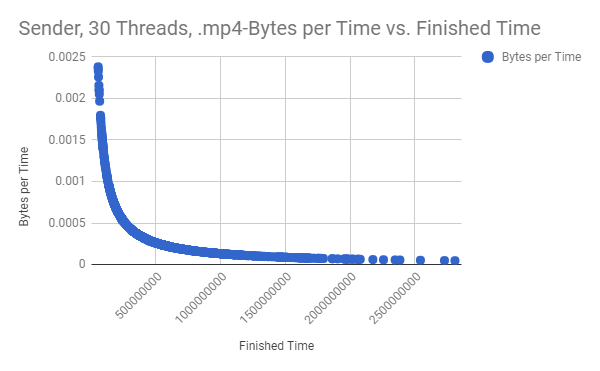
*Figure 8.3 - Comparison of Time Required to Receive and the Size of the Data Received*

**

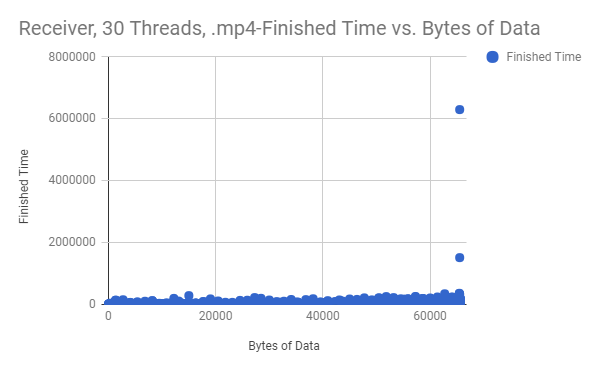
*Figure 8.4 - Comparison of the Speed at which Data is Received and the Time it took to Receive the Data*

**

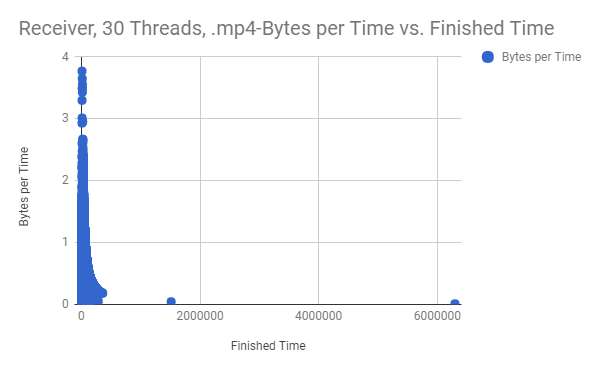
*Figure 9.1 - Comparison of the Time Required to Send and the Size of the Data Sent*

**

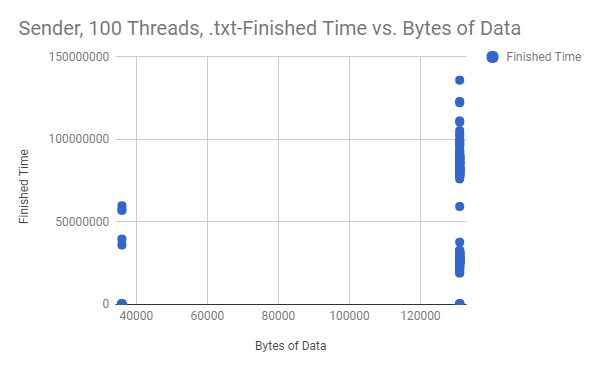
*Figure 9.2 - Comparison of the Speed at which Data is Sent and the Time it took to Send the Data*

**

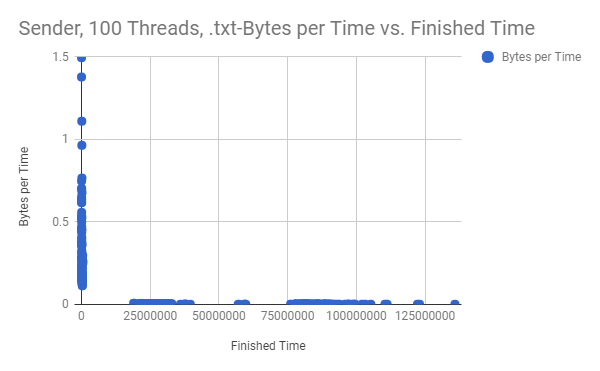
*Figure 9.3 - Comparison of Time Required to Receive and the Size of the Data Received*

**

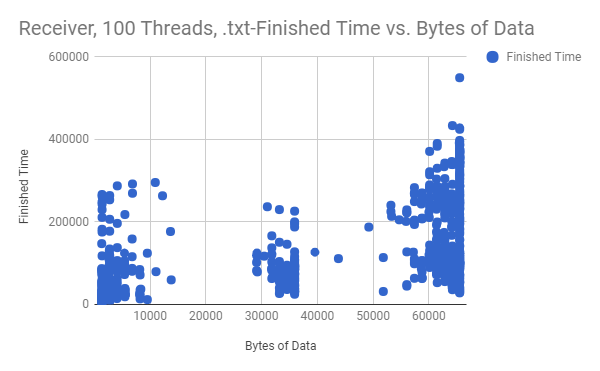
*Figure 9.4 - Comparison of the Speed at which Data is Received and the Time it took to Receive the Data*

**

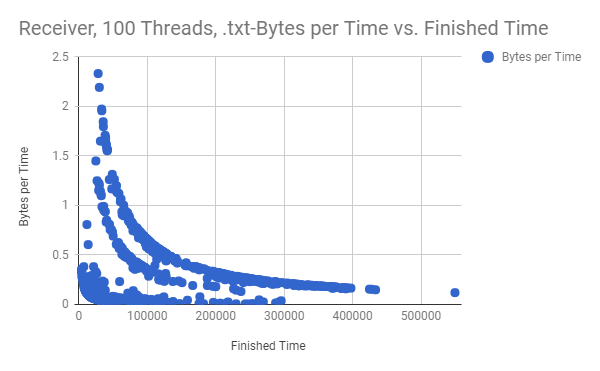
*Figure 10.1 - Comparison of the Time Required to Send and the Size of the Data Sent*

**

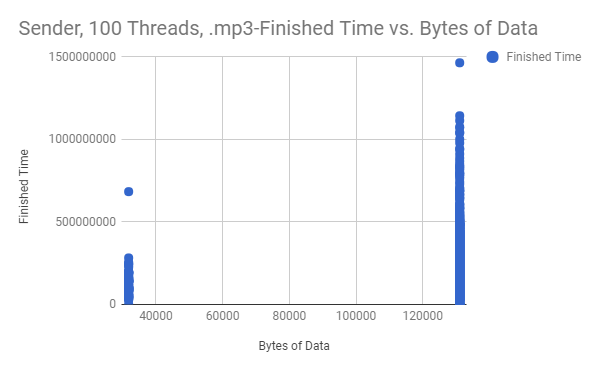
*Figure 10.2 - Comparison of the Speed at which Data is Sent and the Time it took to Send the Data*

**

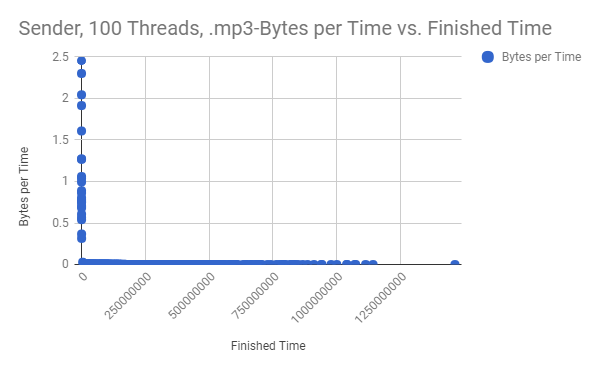
*Figure 10.3 - Comparison of Time Required to Receive and the Size of the Data Received*

**

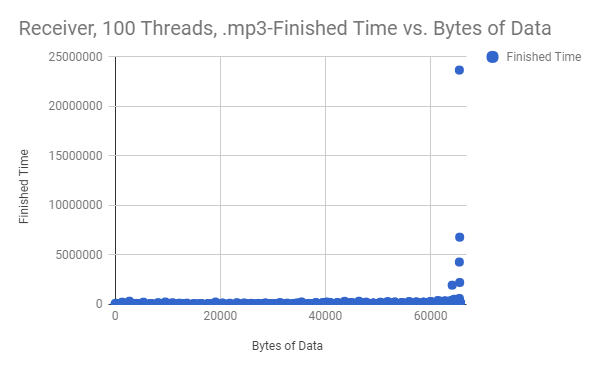
*Figure 10.4 - Comparison of the Speed at which Data is Received and the Time it took to Receive the Data*

**

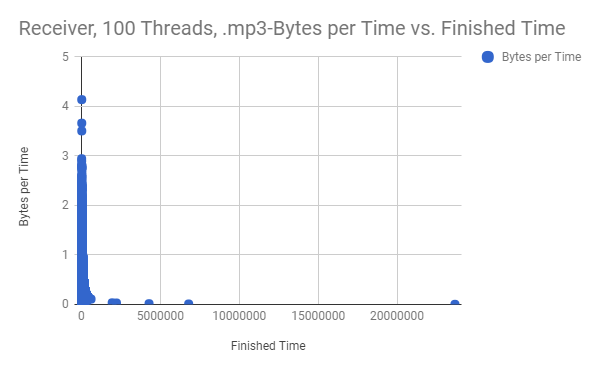
*Figure 11.1 - Comparison of the Time Required to Send and the Size of the Data Sent*

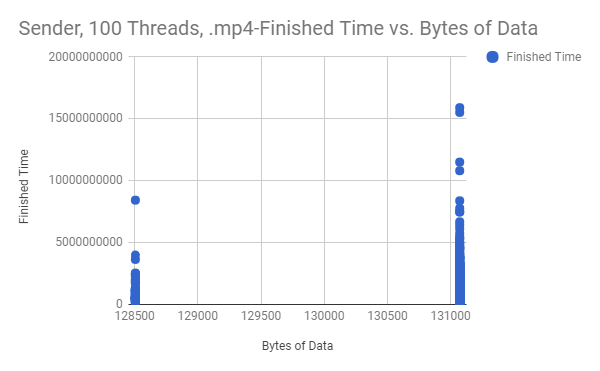
**

*Figure 11.2 - Comparison of the Speed at which Data is Sent and the Time it took to Send the Data*

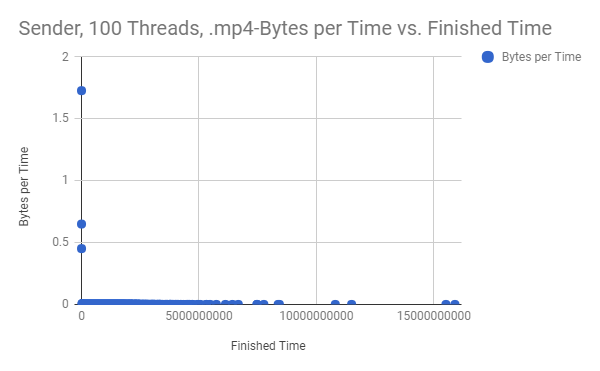
**

*Figure 11.3 - Comparison of Time Required to Receive and the Size of the Data Received*

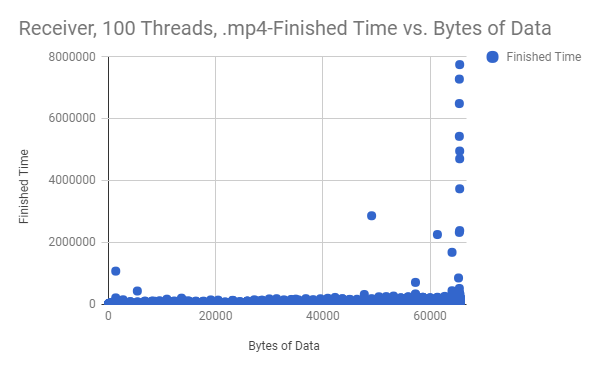
*Figure 11.4 - Comparison of the Speed at which Data is Received and the Time it took to Receive the Data*

**

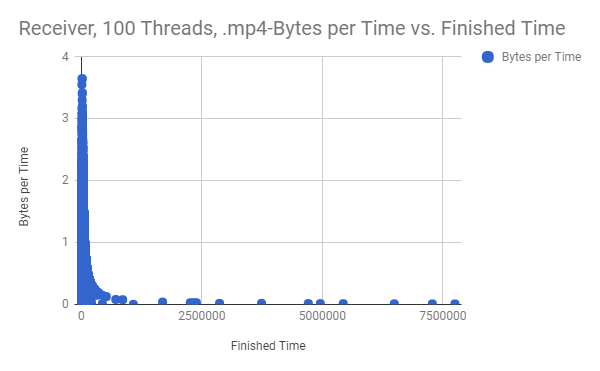
*Figure 12.1 - Comparison of the Time Required to Send and the Size of the Data Sent*

**

*Figure 12.2 - Comparison of the Speed at which Data is Sent and the Time it took to Send the Data*

**

*Figure 12.3 - Comparison of Time Required to Receive and the Size of the Data Received*

**

*Figure 12.4 - Comparison of the Speed at which Data is Received and the Time it took to Receive the Data*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Average Message Size | Average Completion Time | Average Speed (Bytes/Time) |
| 10 Threads, .txt | 99352.6667 | 38088269.43 | 0.0989 |
| 10 Threads, .mp3 | 127970 | 112989154.3 | 0.0013 |
| 10 Threads, .mp4 | 131032.5231 | 299024994.8 | 0.0006 |
| 20 Threads, .txt | 99352.6667 | 59246219.35 | 0.0016 |
| 20 Threads, .mp3 | 127970 | 171766127.9 | 0.0241 |
| 20 Threads, .mp4 | 131032.5231 | 434783383.9 | 0.0004 |
| 30 Threads, .txt | 99352.6667 | 60130438.22 | 0.0016 |
| 30 Threads, .mp3 | 127970 | 193672479.2 | 0.0008 |
| 30 Threads, .mp4 | 131032.5231 | 579983001 | 0.0003 |
| 100 Threads, .txt | 99352.6667 | 39010990.92 | 0.1164 |
| 100 Threads, .mp3 | 127970 | 214579278.1 | 0.0088 |
| 100 Threads, .mp4 | 131032.5231 | 907114701.2 | 0.0007 |
| Overall Average | 119451.7299 | 259199086.5 | 0.0213 |

*Table 1.2 - Various Averages for Sending Data*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Average Message Size | Average Completion Time | Average Speed (Bytes/Time) |
| 10 Threads, .txt | 24493.3913 | 93242.4565 | 0.2475 |
| 10 Threads, .mp3 | 40186.8499 | 65904.4789 | 0.5594 |
| 10 Threads, .mp4 | 44016.093 | 67635.0646 | 0.6808 |
| 20 Threads, .txt | 33302.5698 | 81232.743 | 0.5282 |
| 20 Threads, .mp3 | 41616.2602 | 73297.6367 | 0.6408 |
| 20 Threads, .mp4 | 45303.7979 | 50758.6463 | 0.9512 |
| 30 Threads, .txt | 33489.6629 | 79624.6217 | 0.4522 |
| 30 Threads, .mp3 | 46481.7253 | 52804.1192 | 0.9519 |
| 30 Threads, .mp4 | 43256.0386 | 59879.4244 | 0.7836 |
| 100 Threads, .txt | 31080.0834 | 105438.2117 | 0.3126 |
| 100 Threads, .mp3 | 43462.53449 | 75249.5802 | 0.6579 |
| 100 Threads, .mp4 | 42441.2697 | 53536.5898 | 0.8913 |
| Overall Average | 39094.1897 | 71550.2978 | 0.6381 |

*Table 1.3 - Various Averages for Receiving Data*

# 

# 

# Analysis

## **Receiver Side:**

### Finished time vs Bytes of Data

The receiver side had a linear trend for the finished time in comparison with the bytes of data sent. This is probably because the size of data was larger, so it required more bandwidth which took more time to send over our VPN. When scaling the size of our system up, there was not much of a noticeable difference trend wise other than our test of sending a text file using 100 threads of receiver peers, which can be seen in *figure 10.3*. There are noticeable gaps between data that is a little over 10000 bytes in size and close to 30000 bytes in size, the same goes for between around 35000 bytes in size and 55000 bytes in size. There could be a number of reasons for this to occur, maybe something with how each thread is receiving the file, latency between systems as we were testing across a VPN, as well as the speed of everyone’s system for the code to run.

10 Threads: Figures 1.3, 2.3, and 3.3 all show 10 threads for a txt file, mp3, and mp4 respectively. All show a linear trend but the data for the mp4 shows the tightest group of data points which that the mp4 even though it is a bigger time has a more consistent received time compared to other file formats.

20 Threads: Figures 4.3, 5.3, and 6.3 all shows 20 threads for a txt file, mp3, and mp4 respectively. Like before all show a linear trend but the mp4 file, figure 6.3, has an exponential trend and figure 4.3 and 5.3 both have outliers.

30 Threads: Figures 7.3, 8.3, 9.3 all show 30 threads or a txt file, mp3, and mp4 respectively. Unlike before though the mp3 and mp4 files show a much tighter grouping of data points when compared to the last thread variations. All test as well have outlier in all their data. This could be due to how the computers were connected.

100 Threads: Figures 10.3, 11.3, 12.3 all show 100 threads or a txt file, mp3, and mp4 respectively. The data pattern is almost identical to the 30 threads test even down to the grouping of data for the text file as there are the same 3 distinct groups seen in the graph.

### **Bytes per Time vs Finished Time**

Overall the receiver side negative exponential trend for bytes per time vs finished time for each type of file sent with any amount of threads. The may trend that we saw with the data was as the file type and size was changes, in our case increased, the data became more consistent and created a better trend line. The test with 10 threads and a text file had the worst data representation as there was barely a trend available. We also saw that as the number of threads increased the faster the finished time was for all file types. The exception was with the 100 threads test, there may have been a network issue when we were testing as there was many outliers of data.

10 Threads: Figures 1.4, 2.4, and 3.4 all show 10 threads for a txt file, mp3, and mp4 respectively. We saw that as the file type was more complex that is there was a tighter group of data that was formed. Figure 1.4 that shows text file data has the loosest amount data together. While Figure 3.4 with the mp4 file format shows the tightest group of data. The reason this may be is due to increasing the the overall file size because as we changed the file type we also increased the size.

20 Threads: Figures 4.4, 5.4, and 6.4 all shows 20 threads for a text file, mp3, and mp4 respectively. The trend in 10 threads holds with 20 threads as the file type and size changed the data became more grouped together as well show a better negative exponential trend.

30 Threads: Figures 7.4, 8.4, 9.4 all show 30 threads or a text file, mp3, and mp4 respectively.The trend in 30 threads is similar as 10 or 20 threads, but the finished time is shorter overall for all threads. The bytes sent to each peer much faster and as the file size and type changed the data stayed the same density.

100 Threads: Figures 10.4, 11.4, 12.4 all show 100 threads or a text file, mp3, and mp4 respectively. When testing with 100 threads we saw that the text file took the longest to finish as well it had a ripple effect, it seemed that data was being sent in bursts rather than a stream. This may be due to the connection between computers or an issue with our VPN. We also saw that mp3 and mp4 had many more outliers for the finished time, this may be due to same reason as the text file test, the connection between computer.

## **Sender Side:**

### **Finished time vs Bytes of Data**

*Figure 12.1* and all other figures like it follow a similar trend regardless of the file type or amount of threads. The gaps that are seen in the graphs like in *figure 11.1* represent threads catching up. In *figure 11.1* there are 2 visible gaps. This is most likely due to the nature of how threads work and you can’t really know exactly when they will do something.

### Bytes per Time vs Finished Time

The results from these graphs follow 3 different patterns. The first is that all data is close to the x and y axis (ex: *figure 11.2*). This pattern is in all of the runs that used 100 threads regardless of data type. It also shows up in runs that used 10 threads like *figure 1.2* but not for all of them. This pattern shows that data that is send the quickest is also sent first. The rest of the time the rate of sending data stays the same regardless if it is the second packet or the last. There is no decay.

The second pattern is that of a downward sloping exponential curve (ex *figure 9.2*). This type of pattern is the one you would expect to see. Data is sent the fastest at the beginning and as time goes on the rate at which data is sent decreases. However this is the least common pattern that emerges. It appears in cases where there are fewer threads and the file type is mp4. Mp3 runs either follow this second pattern or the third pattern. This is thought to be because of the size of the files being larger than that of the text file.

The third pattern is that of two downard slowing exponential curves (ex *figure 4.2*). This pattern is most visible in *figure 4.2* and *figure 7.2*, and is less severe and closer to pattern 2 in *figure 2.2* and *figure 8.2*. It is most severe when the data is a text file. This can be contributed to the small amount of threads as well as the small amount of data. What is suspected of happening is that since the files are smaller, threads are being ended sooner so the other threads can run faster.

## **Server Router:**

### Average Server Router Lookup times

According to Table 1.1 average server lookup times were somewhat scattered. When looking at the table though in a graph instead we can see that there is not much of a correlation to the file type and the routing table, which we expected, but there is a correlation to the number of threads and the speed of routing table lookup times then. We can see that as we go from 10 threads up to 100 threads a slight decrease in time, granted that we take away all data that was outliers such as the 10 thread text and 30 thread mp3 test. The difference in times though is very miniscule, as everything is measured in nanoseconds, and since we want to apply this to real-world issues, we can then come to the conclusion that the routing table look-up are almost identical so it would always be beneficial to utilize routing table look-up via remote method invocation rather than a polling system.

# Conclusion

The overall take away from this simulation test was that peer-to-peer was faster in overall testing than having a client-server paradigm. To go about testing this, we wanted to create an environment that simulated as much of a real-life situation as possible, so we made sure to implement remote method invocation to ask another super-peer to check its routing table. We also made sure to transmit multiple file types across multiple peers (up to 100). The most surprising results from our test were that when a super-peer was looking at another super-peer’s routing table, the look-up times were very similar and in some cases the remote look-up was faster. We also found that as the file type changed and the number of threads increased, there was a relationship to speed at which the file was sent and received which can be seen in Tables 1.2 and 1.3. Both of these tables show the most relevant information that should be shown from this project.

We saw natural trends in how the receiver and sender version of a peer worked, and all data shows expected trends when changing the file type and thread count.

The simulation confirms our belief that peer-to-peer is faster than server-client, even with multiple super-peers. All of our data shows that file type, number of threads, and size of the file being sent affect the overall time of each packet transmission of a file.

# 

# 

# Appendix

Corrections from phase 1:

The points that were deducted in phase 1 were for the format, table of contents, and discussion of our graphs and code snippets. We rectified this by correctly labeling all figures and including all figures in our discussions for their respective sections, as well as making sure the Table of Contents is formatted correctly when converted to a docx file.