

Distributed Systems

Project Phase 2 Report

Fall 2016 – Distributed Computing

Team Members:

Jacob Ertell

Ryan McMichael

Jun Nguyen

Miles Sakmar

Abdul Wahab

Table of Contents

1. Abstract	3
2. Introduction	3
3. Design	5
3.1 Classes	6
3.1.1 Node	6
3.1.2 ServerParent.....	6
3.1.3 Server	6
3.1.4 Client	7
3.1.5 Router.....	7
3.1.6 IpPort.....	7
3.2 Protocols.....	8
3.2.1 Node-Router.....	8
3.2.2 Client-Server.....	8
4. Implementation	9
4.1 Router.java	9
4.2 Node.java	11
4.3 ParentServer.java	11
4.4 Server.java.....	11
4.5 Client.java.....	13
5. Simulation	15
5.1.1 Router Start.....	15
5.1.2 Router Start.....	15
5.1.3 Router Start.....	16
5.1.4 Router Running	16
5.2.1 Node Start	17
5.2.2 Node Start	17
5.2.3 Node start.....	18
5.2.4 Node Running.....	18
6. Data/Analysis.....	20
6.1 Average Transmission Rate	20
6.2 Transmission Rate vs. Buffer Size	22
6.3 - Average Routing Table Look-Up Time	23
7. Conclusion/Comments	25
8. User Guide	26

1. Abstract

This report describes the design, implementation, testing, and analysis of a distributed computing network. The network in question is a peer-to-peer network that is divided into sub-networks by routers. Testing covers various trials in which files of various types and sizes were sent throughout the network, analyzing the time and consistency of the transfer.

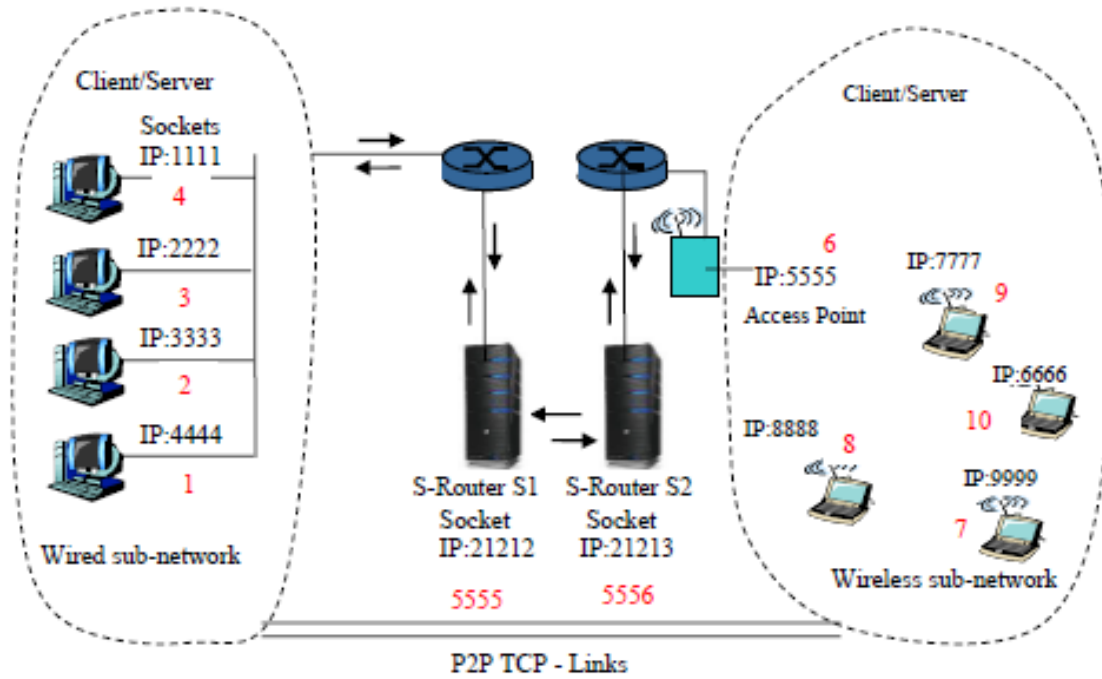
2. Introduction

In Phase 1, a simple distributed system was constructed and tested. The system featured 3 types of programs: a Client, a Server, and a Router. Each node within the system was assigned to one of these roles. The Client sent requests and received responses. The Server listened for requests and sent responses. The Server Router acted as middleman between the two, listening for and forwarding their messages.

Thus, the flow of data within the system of Phase 1 is as follows: when each node is initialized, the Client is waiting to generate a request while the Server and Router are listening for any requests; upon receiving instructions from a user, the Client generates and sends a request to the Server Router, and then begins listening for a response; the Router receives the request and forwards it to the Server; the Server then reads the request, generates a response, sends that response to the Router and continues to listen for further requests; the Server Router, still listening, receives this response and forwards it the Client before going back to listening; finally, the Client receives its response, reads it, and then waits for further instructions from the user.

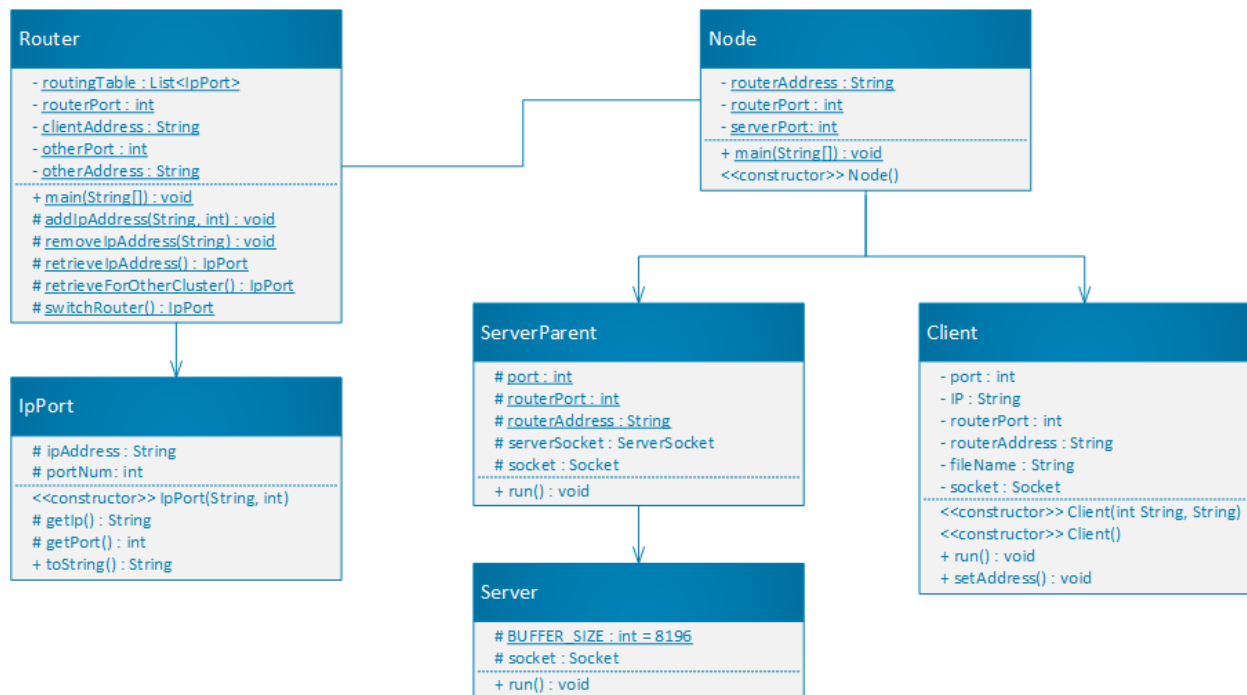
Phase 2 aimed to expand upon this system, to generate a larger and more complex peer-to-peer network. The main differences between the systems of Phase 1 and Phase 2 are the expansion and overlapping of roles, the larger system size, and the division of nodes within the system To further explain, each node will no longer be either a Client or a Server, but will instead act as both, sending and

responding to requests. The network will also be divided into sub-networks, with a Router assigned to each sub-network. This leads to the expanded role of the Router, namely that it will now be responsible for coordinating with other Routers in order to enable communication between nodes of different sub-networks. Further details on this will be covered in the Design and Implementation sections.



3. Design

Since the architecture is peer-to-peer, the main class in this project is Node. Each user will run an instance of the Node class. Each Node class will connect to an instance of the Router class, which helps Nodes connect to each other. It does this by storing a list of connected Nodes and their IP addresses. When a Node requests a file, the Router will send it the IP address of the appropriate Node to allow the Nodes to connect directly.



3.1 Classes

3.1.1 Node

The Node class is the primary class for users. It uses a GUI to gather information about the Router's IP address and port number, as well as the port number to use for accepting requests from other Nodes. It handles the Client and ServerParent classes, as well as connecting to the Router. The responsibilities of Node include:

- Registering the Node's IP address with the Router
- Spawning a ServerParent thread to handle incoming requests
- Retrieving the appropriate IP address and port number for Client requests and spawning new Client threads to request files

3.1.2 ServerParent

The ServerParent class is responsible for accepting incoming connections. While the ServerParent is running, it will always keep a ServerSocket open to accept incoming connections. When the ServerSocket connects, it will pass the resulting Socket object to a new instance of Server, which will be run in a separate thread. The ServerParent will then repeat the process and wait for a new incoming connection.

3.1.3 Server

The Server class is spawned by an instance of ServerParent. Server's constructor accepts a Socket object, which is the (already-connected) Socket generated by ServerParent. When Server is run, it will receive a request, from Client, for a specific file. The Server will then look for the file and, if it exists, will break the file down into byte arrays. The byte arrays are sent to the Client one by one until the entire file is sent.

3.1.4 Client

The Client class is spawned by the Node. Every time the user requests a file using the Node's GUI, Node will create a new instance of Client and run it in a separate thread, giving it the ServerParent's IP address, port number, and the name of the file to request. The Client will attempt to connect to the Server. If it is successful, it will read byte arrays from the Server thread until it has received the entire file. The file is saved with a filename corresponding to the time the download was initiated.

3.1.5 Router

The Router class is responsible for holding a lookup table of all connected Nodes. It is also able to connect to another Router. When connected to a second Router, it can send and request data about connected Nodes.

Each Node must register itself with the Router. When the Node is registered, its IP address and port number are saved in the Router's list. When disconnecting, a Node should "un-register" itself with the Router, which removes its entry from the list.

When a Node is requesting a file, the Router will find the IP address and port number in the list and send it back to the Node. Then, the Node's Client thread will be able to directly connect to the appropriate Node. This makes the structure peer-to-peer instead of server-based.

3.1.6 IpPort

This class is an inner class of Router. It is a data structure designed to hold an IP address and port number pair.

3.2 Protocols

3.2.1 Node-Router

While the Node class is responsible for gathering connection details about the Router, it is not responsible for establishing the connection or registering itself with the Router. This is the responsibility of the ServerParent, which is managed by the Node. The Node will spawn and run an instance of the ServerParent class. The ServerParent will create a new connection (via a Socket object) to the Router, declaring its status as a Server (“SERVER”), its IP address, and the port number it will be listening on.

If the Node chooses to disconnect from the Router, it should unregister its IP address. To do so, the Node class itself will connect to the Router via a Socket object and declare itself offline (“SERVER_OFFLINE”). It will also send its IP address so the Router can remove the appropriate entry from the list of connected Nodes.

3.2.2 Client-Server

To initiate a connection to the appropriate Server, the Client must first request the IP from the Router. The Node will give the Client the Router’s connection information when the Client thread is spawned. The Client will connect to the Router via a Socket object to request an IP address and port number. After successfully retrieving the connection details, the Client will connect to the ServerParent, which passes the connection to a new Server thread.

The Server will then send details about the file transfer. Specifically, the Server will send the size of the file, the size of the byte array, and the type of file to be transferred. From this, the Client will be able to calculate how many byte arrays are going to be sent, as well as the number of remaining bytes that will be transferred afterwards (the remaining bytes are bytes that wouldn’t fill up a full byte array, and are sent at the end).

4. Implementation

4.1 Router.java

The responsibility of a router is to store the IP address and port number of all active servers and distribute those to clients. All of the active server IP addresses and port numbers are stored inside of the list called routing table. When a server wants to be registered to the routing table, it first needs to send a string literal of "SERVER", and then send its own IP address and port number to the router. Once the router receives all necessary information, it then adds the IP address and port number to the routing table. When a server is going offline, it needs to remove itself from the routing table. In order to do that, the server must send a string literal of "SERVER_OFFLINE" and its own IP address to the router; then, the router will search through routing table and delete the passed IP address from the routing table. Finally, when a client retrieves a server IP address and a port number, the client must send a string literal of "CLIENT" to the router; then, the router will randomly pick one cluster out of two clusters, then pick an available server from the list, and then return the IP address and the port number to the client. Since the communication system is P2P, a client is also a server, and therefore the client's IP address is stored in the routing table as a server. To avoid having the router send a client its own IP address, the router checks the retrieved IP address against the client's IP address. If the two are equals, then it retrieves another IP address. "SWITCH_ROUTER" on the Figure 4.1.1 is used to send a client request to another router when a client is attempting to retrieve server information from another cluster.

```

switch (inputLine) {
    case "SERVER":
        System.out.println("server online request");
        serverIpAddress = in.readUTF();
        serverPortNumber = Integer.parseInt(in.readUTF());
        addIpAddress(serverIpAddress, serverPortNumber);
        System.out.println("IP Address: " + serverIpAddress + " was added to routing table");
        break;
    case "SERVER_OFFLINE":
        System.out.println("server offline request");
        int previousSize = routingTable.size();
        serverIpAddress = in.readUTF();
        removeIpAddress(serverIpAddress);
        int currentSize = routingTable.size();
        if (previousSize == currentSize) {
            System.out.println("No IP Address found");
        }
        System.out.println("IP Address: " + serverIpAddress + " was removed from routing table");
        break;
    case "CLIENT":
        System.out.println("client request");
        IpPort ipPort = retrieveIpAddress();
        if (ipPort == null) {
            out.writeUTF("");
            out.writeUTF("");
        }
        out.writeUTF(ipPort.getIp());
        out.writeUTF(String.valueOf(ipPort.getPort()));
        break;
    case "SWITCH_ROUTER":
        System.out.println("client request");
        IpPort ipPortFromOtherCluster = retrieveForOtherCluster();
        if (ipPortFromOtherCluster == null) {
            out.writeUTF("");
            out.writeUTF("");
        }
        out.writeUTF(ipPortFromOtherCluster.getIp());
        out.writeUTF(String.valueOf(ipPortFromOtherCluster.getPort()));
        break;
}

```

Figure 4.1.1 Router.java

4.2 Node.java

A node is a peer on the peer-to-peer system. The responsibility of a node is to prompt the user to input router information and port number that server will be listening on, and then ask user for which file to download. Once the user picks a file to download, the node will spawn a thread of Client.java. GUI was implemented using swing library.

4.3 ParentServer.java

The responsibility of the parent server is to register itself onto routing table in the router and then spawn a server thread. First, the parent server will create a new ServerSocket with the specified port number, then it will start listening for incoming request by accepting requests. A new thread of server is created with an argument of the active socket.

```
// Create a server socket and start listening
serverSocket = new ServerSocket(Port);
System.out.println(String.format("Listening on port %d", Port));

// Get the connection...
socket = serverSocket.accept();
System.out.println(String.format("Connected to %s", socket.getInetAddress()));

System.out.println("Sending Socket object to new Server thread");
(new Thread(new Server(socket))).start();
serverSocket.close();
```

Figure 4.3.1 ParentServer.java

4.4 Server.java

The server will receive a request from a client with a specific file name. The server will then look for a file within its local resources directory. If the file doesn't exist, it then throws FileNotFoundException. Once the file is found, it sends the file size, buffer size, and file extension of the file to the client. The file size and the buffer size will be used to calculate amount of iteration the client will need to listen on the input stream. Finally, the server splits the file into smaller buffers, then sends one buffer at a time to the client until every byte of the file is

sent. Figure 1.2 depicts the process of sending one buffer at a time until it runs out of buffer, then it sends remaining of bytes.

```
FileInputStream fis = new FileInputStream(fileToSend);

// Send the file's statistics (total bytes, buffer size).
// From this, the client can calculate everything we need to know.
out.writeInt(fileSize);
out.writeInt(BUFFER_SIZE);
out.writeUTF(fileExtension);

for (int i = 0; i < numberOfByteArrays; i++)
{
    byte[] fileBuffer = new byte[BUFFER_SIZE];
    fis.read(fileBuffer, 0, BUFFER_SIZE);

    out.write(fileBuffer);
    fileIndex += BUFFER_SIZE;

    System.out.println("Sent bytes " + fileIndex);
}

// Send the remaining bytes
if (remainderBytes > 0)
{
    byte[] remainder = new byte[remainderBytes];
    fis.read(remainder, 0, remainderBytes);

    out.write(remainder);
}

fis.close();
```

Figure 4.4.1 Server.java

4.5 Client.java

The responsibility of a client is to retrieve the server IP address and port number from the router and then establish a connection with a server to download the specified file. First, the client sends a string literal of "CLIENT" to the router to retrieve a server's information. This task is accomplished by setAddress() method on Figure 1.3. Once the server information is retrieved from the router, it will attempt to establish a connection with server. The client will first send a string literal of the file name it wants to download, then the server will send back file size, buffer size, and the file extension. The file size and the buffer size will be used to calculate amount of iteration the client needs to listen from the server, as mentioned previously. Finally, the client downloads the specified file one buffer at a time. As it reads a buffer from the input stream, it creates a file using FileOutputStream. The file is given with an appropriate extension. Figure 1.4 depicts the process of creating a file as it reads file buffer from the input stream.

```
public void setAddress() {
    try {
        Socket routerSocket = new Socket(this.routerAddress, this.routerPort);
        DataOutputStream out = new DataOutputStream(routerSocket.getOutputStream());
        DataInputStream in = new DataInputStream(routerSocket.getInputStream());

        out.writeUTF("CLIENT");
        IP = in.readUTF();
        Port = Integer.parseInt(in.readUTF());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figure 4.5.1 Client.java setAddress() method

```
File file = new File(formatter.format(new Date()) + extension);
file.createNewFile();

FileOutputStream fos = new FileOutputStream(file);

PrintWriter text = new PrintWriter(new File(formatter.format(new Date()) + ".txt"));
long begin, end;

for (int i = 0; i < numberOfByteArrays; i++) {
    begin = System.nanoTime();
    byte[] buffer = new byte[bufferSize];

    while (in.available() < bufferSize) ;

    in.read(buffer, 0, bufferSize);
    end = System.nanoTime();
    fos.write(buffer);
    text.println(end - begin);
}

text.close();

// Get the remainder
byte[] buffer = new byte[remainderBytes];

in.read(buffer, 0, remainderBytes);
fos.write(buffer);

fos.close();
```

Figure 4.5.1 Client.java

5. Simulation

5.1.1 Router Start

When a Router is started, the first prompt asks for the port that it will listen on. This value is used to instantiate a ServerSocket to listen for connections.

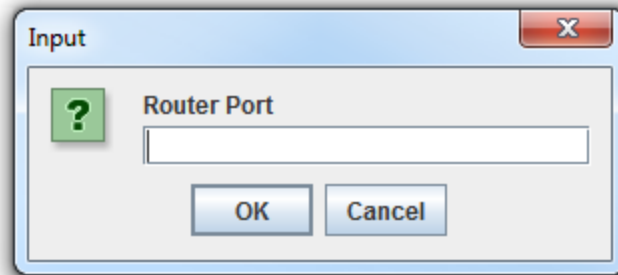
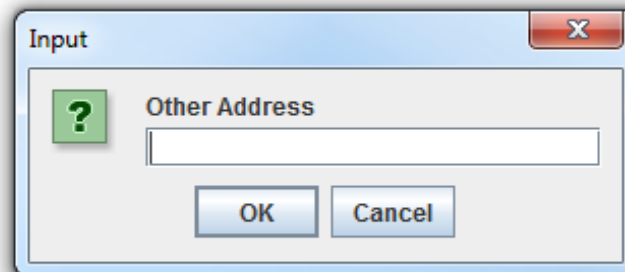


Figure 5.1.1

5.1.2 Router Start

After the first prompt, the Router will then be prompted to enter the address of another



Router, if one is running. If one is not running, then nothing has to be entered.

Figure 5.1.2

5.1.3 Router Start

After the second prompt, the Router will then be prompted to enter the listening port of the other Router, if one is running. If one is not running, then nothing is entered.

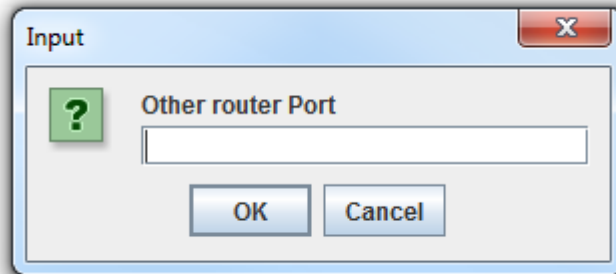


Figure 5.1.3

5.1.4 Router Running

Once the Router is running, no interaction is required from the user. With each interaction, the Router will print the contents of its routing table, as well as the type of request, what was done, and what was returned.

```
"C:\Program Files\Java\jdk1.8.0_73\bin\java" ...
Router is listening on port: 45678
clientAddress: 127.0.0.1
request type: SERVER
server online request
Contents of routing table
-----
IP address: 192.168.0.139
Port Number: 23456
-----
IP Address: 192.168.0.139 was added to routing table
clientAddress: 127.0.0.1
request type: CLIENT
client request
clientAddress: 127.0.0.1
request type: SWITCH_ROUTER
client request
client address: 127.0.0.1
clientAddress: 127.0.0.1
request type: SERVER_OFFLINE
server offline request
Contents of routing table
-----
IP Address: 192.168.0.139 was removed from routing table
|
```

Figure 5.1.4

5.2.1 Node Start

When a Node is started, the first prompt asks for the address of the Router it will connect to. This value is used to register the node with the Router and to request files from it.

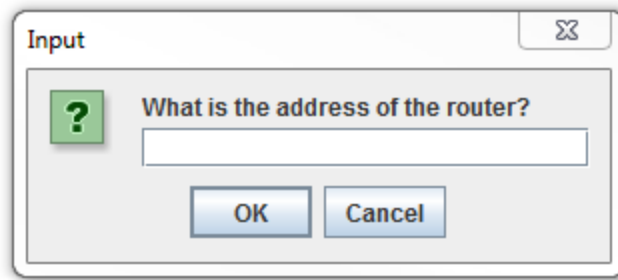


Figure 5.2.1

5.2.2 Node Start

After the first prompt, the next prompt asks for the port number of the Router it will connect to. This is used in conjunction with the router's address, from the previous prompt, to establish communication.

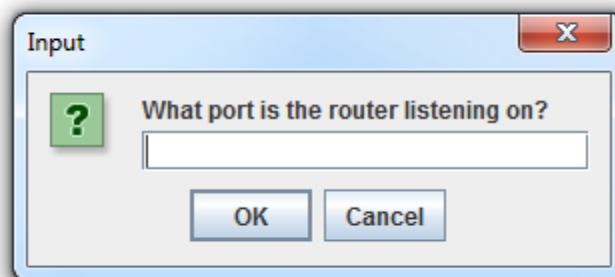


Figure 5.2.2

5.2.3 Node start

After the second prompt, the next prompt asks for the port number upon which the Node will be listening for connections. This is used to spawn a ServerParent to listen for requests for files.

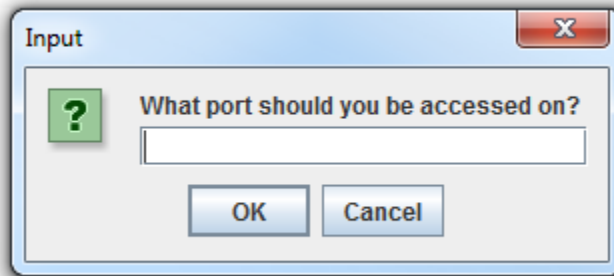


Figure 5.2.3

5.2.4 Node Running

Now a GUI will pop up; this is the interface the user will use to request files to be downloaded. Once a file is selected from the dropdown menu, the user clicks the “GO” button to spawn a Client thread to handle the file request. When the user no longer wants to run the Node, simply click remove, and the node will exit and remove itself from the Router’s table.

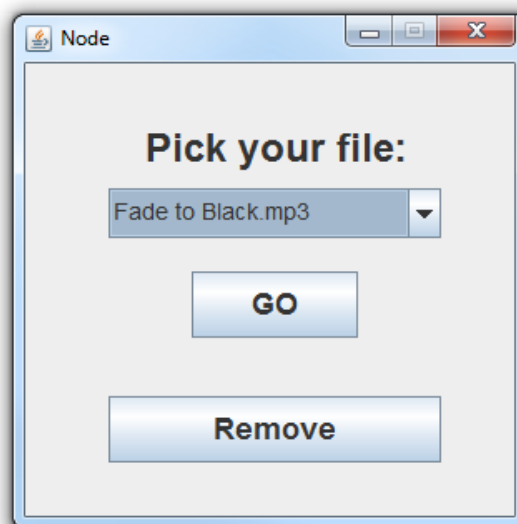
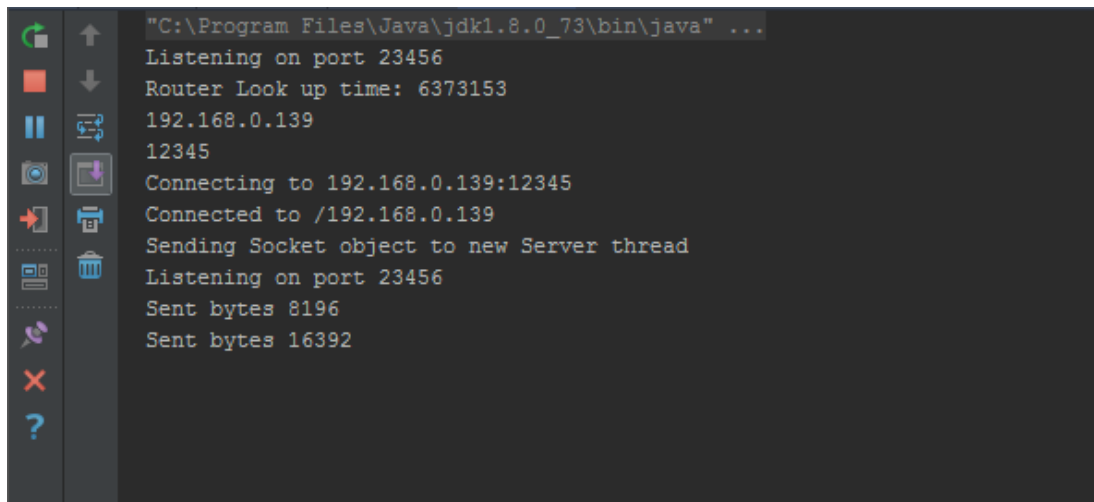


Figure 5.2.4

When the node is running, the console can be used to track what is being sent, received, and connections made.



```
"C:\Program Files\Java\jdk1.8.0_73\bin\java" ...  
Listening on port 23456  
Router Look up time: 6373153  
192.168.0.139  
12345  
Connecting to 192.168.0.139:12345  
Connected to /192.168.0.139  
Sending Socket object to new Server thread  
.....  
Listening on port 23456  
Sent bytes 8196  
.....  
Sent bytes 16392
```

Figure 5.2.5

6. Data/Analysis

This section examines data collected from testing the simulation on send and receive operations. All modules ran on the same Wi-Fi network, thereby, reducing the potential network delay usually associated with messaging. **Table 6.0.1** shown below displays the files, and the size of each file sent across the system.

<u>File Name</u>	<u>File Size (in kB)</u>
Mad Mike.mp4	22,888 kB
Sickness.mp3	5,072 kB
Fade to Black.mp3	9,846 kB
Imperial.mp3	1,934 kB

Table 6.0.1: File Sizes

6.1 Average Transmission Rate

The simulation uses .mp3 and .mp4 files that are transferred between the systems. Each file is a different size, but nonetheless, the average transmission rate for each file falls between the (1700 - 2400) kB/s range. This is because when each file is sent across the system, it is compressed into a byte array buffer; and each byte array buffer takes up file sizes of 4kB, 8kB, 16kB, 32kB, and 64kB for each respective test. Each byte-array size can hold only so much data, which makes the actual size of the file a non-factor. All files get chopped up into multiple pieces of the same size, making the file transmission rate fairly uniform across the system for all files.

Table 6.1.1 shown below displays the average transmission rate of each file in kB/seconds.

<u>File Name</u>	<u>Avg. Transmission Rate (kB/s)</u>
Mad Mike.mp4	1739.6 (kB/s)
Sickness.mp3	2269.0 (kB/s)
Fade to Black.mp3	2291.0 (kB/s)
Imperial.mp3	2352.2 (kB/s)

Table 6.1.1: Average Transmission Rate

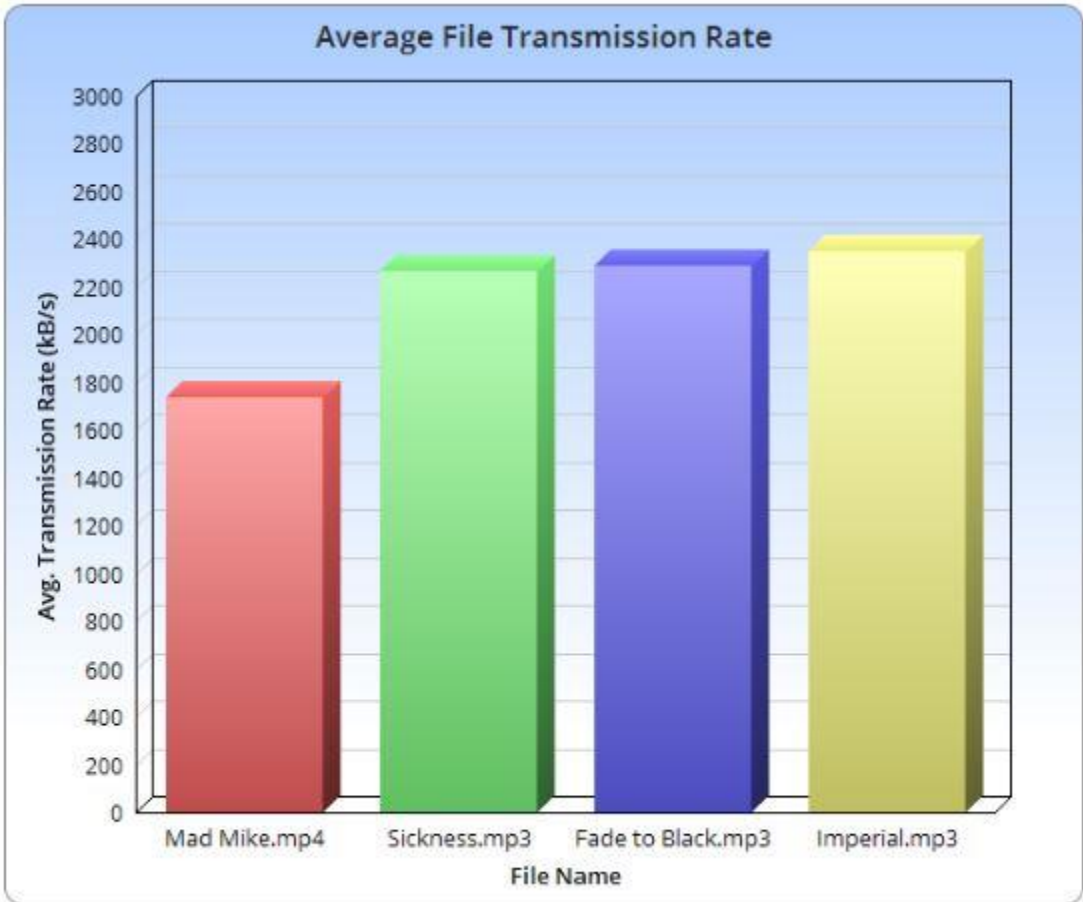


Figure 6.1.1: Average File Transmission Rate

6.2 Transmission Rate vs. Buffer Size

Each file that was sent across the system was broken down into a byte array. The files sent were compressed to a byte array, and then sent over the network. Our simulation sent each of the four test files through 4 different sized byte arrays: **4kB**, **8kB**, **16kB**, **32kB**, and **64kB**. This was done to test and see whether sending compressed files through different sized byte arrays would have any impact on the transmission rate of the files. **Table 5.2.1** shown below displays the average transmission rate of each file, using different-sized byte arrays. We believed that there would be no significant difference between the data transfer rates in this case; the implementation of TCP communication in a Java socket is based around sending streams of byte arrays. This, combined with the fact that TCP is a communication method in which a stream of data is sent, led to our hypothesis being correct. In a UDP system, it would be expected to see a more significant discrepancy here, but the constantly flowing nature of TCP means that this was essentially arbitrary.

	Mad Mike.mp4	Sickness.mp3	Fade to Black.mp3	Imperial.mp3
4 kB	2195.4 kB/s	2541.2 kB/s	2266.3 kB/s	2537.7 kB/s
8 kB	1802.7 kB/s	2226.6 kB/s	2096.5 kB/s	2143.7 kB/s
16 kB	1904.1 kB/s	2356.5 kB/s	2266.9 kB/s	2491.4 kB/s
32 kB	1198.9 kB/s	2374.6 kB/s	2429.9 kB/s	2104.7 kB/s
64 kB	1596.9 kB/s	1846.0 kB/s	2395.6 kB/s	2483.5 kB/s

Table 5.2.1: Average Transmission Rate vs. Buffer Size

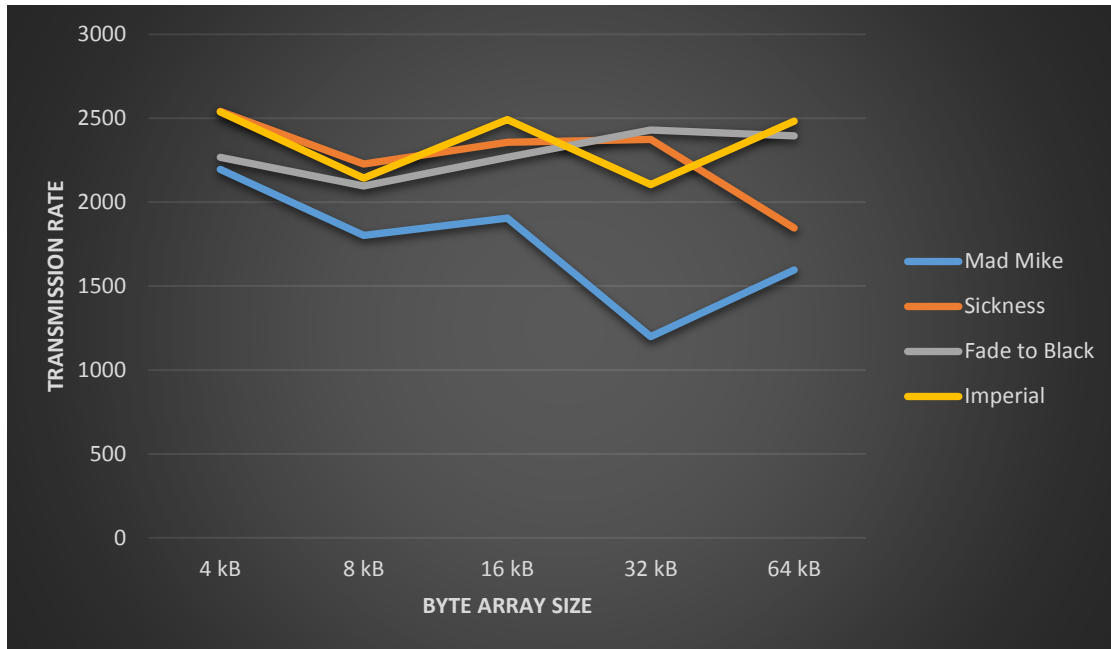


Figure 6.2.1: Average Transmission Rate vs. Buffer Size

6.3 - Average Routing Table Look-Up Time

This subsection will explore the average routing table look-up time for each file. This metric denotes how long it takes the system to find the IP address of a client, to send a requested file. Depending on whether that Client is connected to the same Router as the Server, or a different Router, the timing may vary. By looking at the average times listed in **Table: 6.3.1**, we see that the average times differ at most by 14.1 milliseconds or 0.0141 seconds. The greater lookup time comes from **Mad Mike.mp4**. However, the time it takes for looking up an IP should not be variable to which type of file we are sending. While we tried our best to control variables by having all the test computers on the same network, this seems to be anomalous. The small discrepancies between the files can be attributed to any number of issues, ranging from the campus network, to the computers operating systems, owing to the greater lookup time for Mad Mike.mp4.

Conversely to those ideas, however, is the situation with the lookup time when 2 routers are used. We see a nearly 25% increase in the average look-up time, across the board for all files. This is easily explained when we consider the nature of the routers communicating with one another; every time one router needs information from another, a new TCP connection must be made. Establishing the connection and passing the information back and forth takes time, which leads to the increase in look-up time.

<u>File Name</u>	<u>1-Router Look-Up Time</u> <u>(Average)</u>	<u>2-Router Look-Up Time</u> <u>(Average)</u>
Mad Mike.mp4	0.5503152918 s	0.669183395 s
Sickness.mp3	0.4267932157 s	0.531357554 s
Fade to Black.mp3	0.4482713692 s	0.540167000 s
Imperial.mp3	0.4198214327 s	0.546607505 s

Table 6.3.1: Average Routing Table Look-Up Time

7. Conclusion/Comments

If this project had been designed around a UDP architecture instead of TCP, the system would be much faster and easier to implement. Because a direct connection would not have to be maintained for each transfer, there would be much less overhead to start a connection and transfer data. For the purpose of ensuring that all data arrives, a TCP system is better; there is a constant stream of data being transferred, and it is much easier to guarantee that all data will arrive intact in the correct order. UDP communication does not guarantee the receipt of a message, and situations can arise where data is lost altogether. If speed is the most valuable measurement in a situation, UDP is the correct choice. If reliability is the most valuable measurement in a situation, TCP is the correct choice. For large data transfers, TCP is also the correct choice. Despite the overhead involved with TCP, preventing the congestion of UDP helps to prevent the loss of data when there is a large amount being transferred.

The Internet provides us with the greatest resource in history for sharing and receiving information across the world. Peer-to-peer networks have long been a common system used for sharing information and tools between people; it provides the framework for online communication directly between two computers. For the reasons listed above, we chose to use TCP for the architecture we based our program around; receiving all of the data is more relevant to our software being functional than being fast. The system was broken into progressively smaller classes and functions to allow for efficient testing and modularization.

8. User Guide

1. At least three networked computers are required to run this system.
 - a. Any working wired or wireless connection is acceptable.
2. Each computer must have a functioning Java IDE installed. Examples:
 - a. Eclipse - <https://eclipse.org/>
 - b. InjelliJ - <https://www.jetbrains.com/idea/>
3. Ensure that the most recent Java JDK has been installed.
4. Decide if a given computer will be a Node or Router.
 - a. If Router is chosen, follow these steps:
 - i. Begin by running Router.main.
 1. The first prompt will request that you enter the port you will be listening on.
 2. The second prompt will request that you enter the address of the other router in the system, if one is running.
 3. The final prompt will request that you enter the port that the other router is listening on, if one is running.
 - ii. Once the Router is running, the user does not do anything more. Status can be monitored through the command line output.
 - b. If Node is chosen, follow these steps:
 - i. Begin by running Node.main.
 1. The first prompt will request that you enter the address of the router that you would like to connect to.
 2. The second prompt will request that you enter the port number that the router is listening on.

3. The third prompt will request that you enter the port number that you will be listening on.
 4. The main GUI will now run.
 - a. From this screen, you can choose which file you would like from the dropdown. Once chosen, click “GO” and the file will be downloaded from another Node.
 - b. If you would like to exit the program, click “REMOVE” to remove yourself from the Router’s routing table and exit the program.
 5. Other tasks performed by the Node are handled in the background, and the status of these tasks can be seen in the command line output.
5. When finished, check the root directory of the project on the computer running the Node, and the file requested will be available.