KENNESAW STATE UNIVERSITY

CS 4504 PARALLEL & DISTRIBUTED COMPUTING

PROJECT REPORT – PART 1

Spring 2024

Instructor - Dr. Patrick Bobbie

Names: Eli Headley/02, Charlie McLarty/02, Sam Bostian/02, Ernesto Perez/02, Daron Pracharn/02, Michael Rizig/02, and Jonathan Turner/W01

***Abstract*—The Client-Server paradigm is a fundamental model in distributing computing; the design of Client-Sever distinguishes between two asymmetric roles: servers, providers of a resource or service, and clients, who requests resources or services from the servers in a network. The model abstracts the synchronization and**

delivery of messages: the always-online server waits for requests, and the client sends a request and waits for a response. Compared to the more basic message passing paradigm, where every computer in a network must personally handle both sending and receiving message synchronization, the model-server model simplifies communication. Typically, a router is used to direct traffic, and acts as a repository for IP addresses. In this project we created a basic network to investigate the use of a Client-Server model in distributed computing. Given Java code for setting up the role of a client, server, or a router, we created a network between pairs of Clients and Servers, and a router to direct traffic and build a routing table. We initially investigated sending and modifying strings of characters over our next work, but we later modified the code to support transmitting audio and video files. Performance metrics such as transmission time, and routing-table lookup were also modified in the code.

Keywords—

1 Introduction

The client-server is a distributed computing paradigm often associated with network communications due to the clear division of roles for the collaborating processes. It includes a client process that will make requests to a separate always online server process whose job is to wait for and handle requests coming from clients. This abstracts the message-passing paradigm and simplifies communication synchronization. This report aims to implement a simple message-passing system for text, video, and images that utilizes a server router to redirect communication for multiple client-server pairs. The time to establish the connection and send the data between client and server.

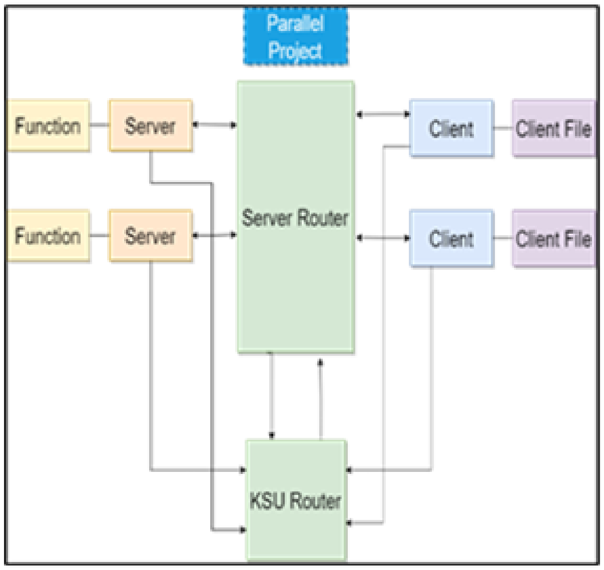


Figure 1: Diagram of our network design

The remainder of the paper is organized as: The design section summarizes the overall plan and layout of our project. The implementation section which provides the specifications of our TCPServerRouter, SThread, our Client and Server implementation in the Java programming language. The Simulation section presents the metrics used to test our implementation as well as the results. The final section, the Conclusion, discusses the takeaways and experience gained in this project.

2 Design Architecture:

As shown in Figure 1 the server router listens for requests for connections on socket port 5555, and for each node connected it adds the node and its IP address to the routing table. For every client/server pair, right and left of the Server Router in Figure 1, the server router creates an SThread that facilitates data flow to and from the client-server pair.

The client begins by connecting to a router, and after sending the IP of the server it wishes to connect to, it sends the file one byte at a time through an OutputStream on socket port 5555. At the router, the dedicated SThread [6] looks up the destination in the routing table and takes in the data in an array of bytes, the buffer, before sending it to the destination. An SThread object is just a Java thread in the router that handles a client/server pair. At the server's end, expecting an array of bytes from a specified client address, takes in the data and writes it to a file.

3 Implementation Approach:

The final implementation of this project revolves around 4 files: TCPServerRouter.java, JSThread.java, Client.java and Server.java.

TCPServerRouter:

Upon execution of TCPServerRouter.java, a ServerSocket [3] object is instantiated along with a routing table implemented as a two-dimensional Object array with ten rows and two columns. The first column represents the IP address in a string format, and the second column contains the socket object used to facilitate connections. Afterwards, the Server Socket calls the accept() method blocking the execution of the process until a connection is received. After receiving the connection, it is saved into a socket object called clientSocket. Subsequently, the routing table is updated to include the IP address of the new connection and its associated socket before passing it to a thread implemented in SThread.java.

```
String[] ip =
(clientSocket.getRemoteSocketAddress() +
"").split(":");
pw.println(ip[0].substring(1));  //this
line updates the routing table with a new
line holding the clients ip in slot one
and a delimiter eg. [clients ip] , [slot
for servers ip]
pw.flush();
RoutingTable[ind][0] =
ip[0].substring(1);
RoutingTable[ind][1] = clientSocket;
SThread t = new SThread(RoutingTable,
clientSocket, ind); // creates a thread
with a random port
```

SThread:

From here, the SThread is created, and running it is as simple as calling t.start(). at this point, the threads constructer will take the passed table with Ip's and sockets, as well as the connection socket and pass information from the client socket to the desired output socket from the routing table. This works because as soon as a client or server connects, its OutputStream [4] push is the address of its intended connection to the threads InputStream [5], who saves that as a string. This string is then run against the table, and when a match is found, it creates an output stream to the corresponding socket in the second column of the routing table. This can be seen below:

```
// loops through the routing table to
find the destination
for (int i = 0; i < 10; i++) {
    if (destination.equals((String)
RTable[i][0])) {
        outSocket = (Socket)
RTable[i][1]; // gets the socket for
communication from the table
        System.out.println("Found
destination: " + destination);
        outToClient =
outSocket.getOutputStream(); // assigns a
writer
    }
}
```

Client and Server:

The client and server work in a similar fashion as they both have the server router IP address and port (which is the same for the client, server, and router) saved in separate strings in order to create a socket that is used to connect to the server router. Their destinations are pushed to the server router thread which will handle communication. However, for the actual transmission of data, the client's role is to send the data while the server will receive it and respond in certain scenarios. If the file sent is a text file, the server will respond by converting the message to uppercase and sending it back to the client. If the file's format is not a text (video, audio, image, etc.), then the server will not respond to the client.

Communication Loop:

Step 1: Host Send

The communication loop works sending a byte stream. We implemented this by utilizing the DataOutputStream and DataInputStream to send raw bytes through the sockets. We also utilize a FileInputStream to read bytes from the input file, which is stored in a file object. By utilizing a buffer, we can take bytes from the input file, load them into a buffer array of bytes, and pass them through the data output socket. If the buffer is empty, meaning no more bytes exist to be sent, the method fis.read(buffer) will return –1, meaning the connection loop should exit and stop sending. Below is a snippet of code used to send the mp4 file as a byte stream through the socket to the SThread.

```
//Variables for MP4 send
File videoFile = new
File("src/Client/cat.mp4");
//creates file for mp4
byte[] videoBytes = new byte[8192];
//creates a byte buffer for video bytes
InputStream dis =
socket.getInputStream();
OutputStream dos =
socket.getOutputStream();
int sentCount = 0;
FileInputStream fis = new
FileInputStream(videoFile);

//write file to dos
while ((sentCount =
fis.read(videoBytes)) != -1) {
    dos.write(videoBytes, 0, sentCount);
}
```

Step 2: Packet Switching

The next step of the communication between hosts is the SThread created by the serverRouter routing the packets to the destination. This is done in a communication loop stored in the SThread which simply accepts the bytes and passes them directly to the other socket of the thread that was identified earlier in SThread. This works in a simple loop that takes in a line from the inputStream and passes it to the output stream. This can be seen below:

```
// Communication loop
byte[] buffer = new byte[8192];
int count;
try {
    while ((count =
inFromClient.read(buffer)) > 0) {
        if (outToClient != null) {
            outToClient.write(buffer, 0,
count);
        }
    }
}
```

Step 3: Host Recieve

The final step of the communication happens on the receiving end (client or server) which involves the server receiving the byte stream and rebuilding it into the original file sent. This works by taking the input stream from the socket and creating a FileOutputStream into a new file named received.x with x representing the respective file type being sent or received. The next step is to pass each buffer size worth of bytes from the data input stream into the output stream which is the file. This can be seen below:

```
InputStream dis =
Socket.getInputStream();
byte[] buffer = new byte[8024];
//creates a 1gb buffer
int bytesRead = 0;  //to keep track of
how many bytes read
FileOutputStream fos = new
```

```
FileOutputStream("src/Server/Received.mp4
");
while ((bytesRead = dis.read(buffer)) !=
-1) {
    fos.write(buffer, 0, bytesRead);
    fos.flush();
}
```

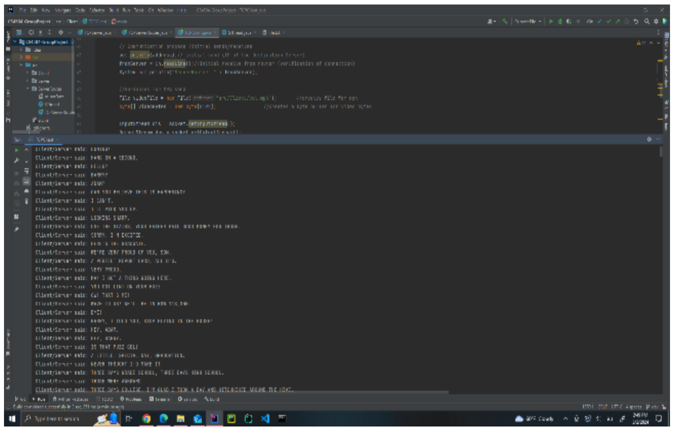
When this while loop concludes (the buffer is empty causing the read() to return -1) the file should be completely built, and the communication will cease. At this point, the file has sent from the sender, through the serverRouter and finally to the receiver. This concludes the communication loop. Modified versions of this loop are used to send different file types, and to resend/respond to requests from the client (such as when the server capitalizes the text and returns it to the client).

4 Simulation Method:

For our experimental setup we conducted three separate simulations where are client computer transmitted various types of files and files sizes to the server computer. This was done to measure the efficiency of our network on Text, Audio, and Video files of varying sizes. For each set of test files we conducted three runs with 27 runs in total and 9 runs for each file type.

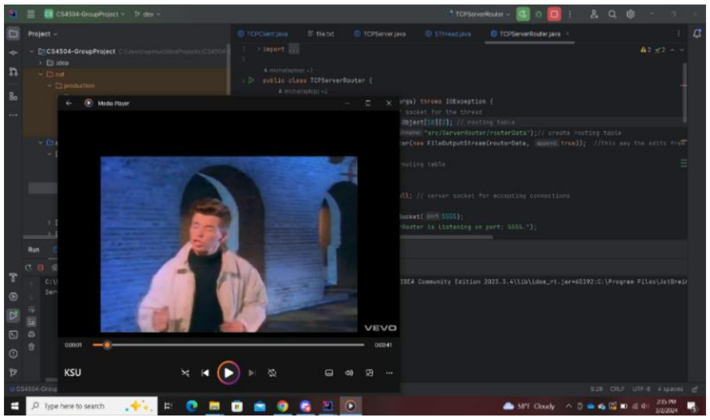
Text Files:

For the first experiment we used three generated text files filled with random characters. The first file sent was a small file, 1KB in size, the second was 488KB of data, and the final text file was a 9.53MB text file.



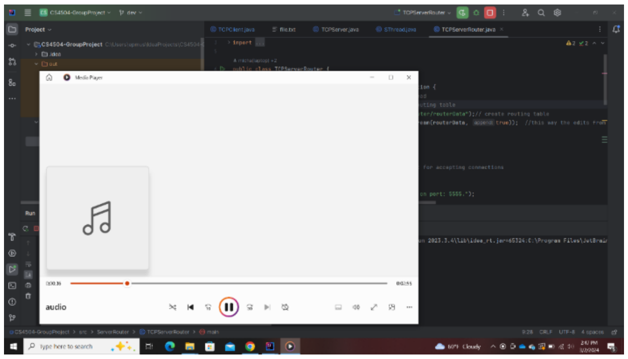
Audio Files:

Like our text files we again used 3 test files for our Audio experiment. The three file sizes used were: 139KB for the small file, 3.62 MB for the medium sized file, and 17.2MB for the large file.

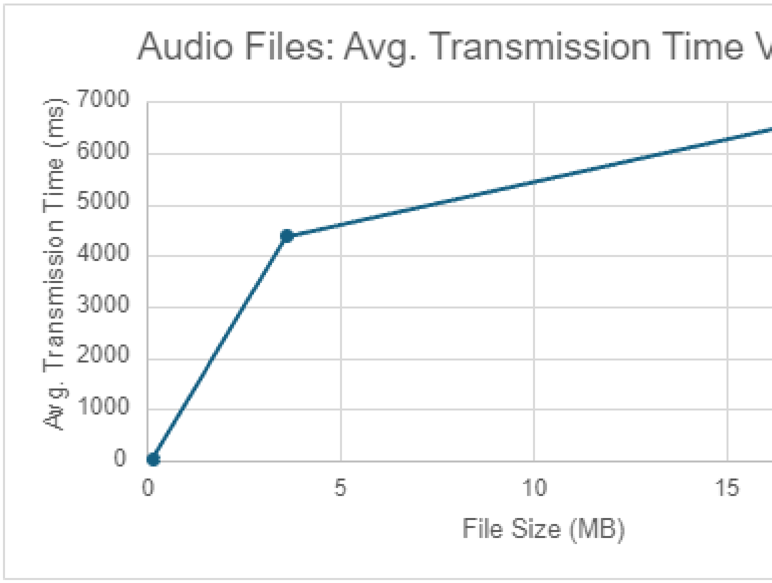


Video Files:

In our final experiment we transmitted three video test files of the .mp4 format from the client to the server computer. The three respective sizes for the small, medium, and large files were as follows: 98.5KB, 10.7MB, and 95.9MB.

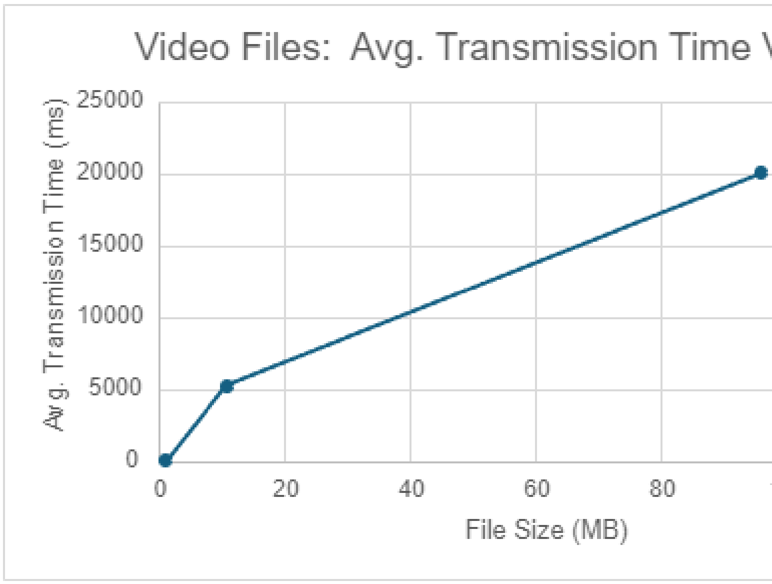


5 Data Analysis:



Audio Files			
Message Size	Transmission Time (ms)		
	Run #1	Run #2	Run #3
139 KB	27	32	27
3.62 MB	4458	4372	4318
17.2 MB	6478	6675	6759

Video Files			
Message Size	Transmission Time (ms)		
	Run #1	Run #2	Run #3
98.5 KB	22	23	21
10.7 MB	5177	5309	5304
95.9 MB	19669	19912	20668



The Client-Server model is a fundamental model because it abstracts event synchronization between the two nodes. The client needs only to send its request and wait for a response. The server's job consists entirely of a continuous loop of listening for requests and fulfilling them. This is much simpler than using message passing, where nodes must be involved in event synchronization because they are given entire control over the messages sent. More abstract models like object space, mobile agent, and remote procedure call can be more adequate for specific problems, but typically come at a cost of more overhead.

As shown in the two graphs above, there is a linear relationship between transmission time and file size. This is especially noticeable in the comparison of video file transmission speed because there is a larger range in file size. Such a linear relationship is to be expected because the entire operation is just sending a 1 to 1 file between two nodes.

Factors that may affect the curve of graphs would have included more than one client for a server or modifying the file in some way.

6 Conclusion:

In this paper, we created our implementation of the client-server paradigm for transmitting messages between multiple hosts in Java. The client and server pair would each establish a connection to the server router which in turn would assign them a thread to handle the connection and allow data to be transmitted back and forth between client and server. This method works for most data types including text, audio, and video which were used in our simulation, because the data being transmitted was in the form of a byte stream.

Timings were taken for each test transmission for various file types, and it showed consistent results when transmitting the same file multiple times. Additionally, the tests showed a linear relationship between the message size and the transmission time regardless of file type which was expected.

The client server implementation proved to be ideal for communication over a network due to the clear division of roles simplifying the synchronization process during communication.

7 References:

[1] M.-L. L. Liu, *Distributed Computing: Principles and Applications*. Boston, Mass: Pearson, 2004.

[2] "Socket". Java Platform, SE 8, API Specification, https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html . Accessed March 3, 2024.

[3] "ServerSocket". Java Platform, SE 8 API Specification, https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html . Accessed March 3, 2024

[4] "InputStream". Java Platform, SE 8 API Specification, https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html . Accessed March 3, 2024.

[5] "OutputStream". Java Platform, SE 8 API Specification,

https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html . Accessed March 3, 2024.

[6] "Thread". Java Platform, SE 8 API Specification, https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html . Accessed March 3, 2024

[7] "BufferedReader". Java Platform, SE 8 API Specification, https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html Accessed March 3, 2024.

8 Appendix:

User-Guide: In order to connect client to server, using the server router, the IP addresses of the client, server, and server router are required. On the client side the IP address of the server router is placed inside the string variable "routerName". Then the IP address of the server will be placed inside the string variable "address". On the server side you need the to place the server router IP address under the string variable "routerName" and the client IP address under the string variable "address". Once the necessary information is entered simply run the program.