Project 1 | Phase 2

DISTRIBUTED COMPUTING

Dhruval Darji

Deion Anthony

Mirela Memic

Table of Contents

[Introduction: 3](#_Toc437020040)

[Design Approach: 4](#_Toc437020041)

[Design Modules and Implementation: 5](#_Toc437020042)

[Client Sample Output: 8](#_Toc437020043)

[Data and Analysis: 9](#_Toc437020044)

[Conclusion: 10](#_Toc437020045)

# Introduction:

For Phase 2 of this project we designed and implemented a Peer-to-Peer system. The purpose of this phase was to design a Peer-to-Peer system that allows as many pairs of nodes as possible to exchange messages. As in Phase 1, a message that is sent in lowercase is to be converted to uppercase and returned. For the implementation of our Peer-to-Peer system, we modified and used our Client-Server programs from Phase 1. We used four main modules in the implementation of our system; TCPServerRouter, SThread, TCPServer, and TCPClient. We also used our custom statistics module, called Stats, to monitor and keep track of variable data, and our main module, called Main, to dynamically run the system. More detail on each of the modules, and how they were modified from Phase 1, will be found in the Design Modules and Implementation section.

# Design Approach:

We began Phase 2 of this project with our programs from Phase 1. The basic function of our system in this phase is the same as in Phase 1; to send a message, in the form of a text file, in lowercase, convert it to uppercase, and return it. With this in mind, we simply used the TCPServerRouter, SThread, TCPServer, and TCPClient modules from Phase 1 and modified them from the Client-Server system to a Peer-to-Peer system. We also used our Stats module for data collection and our Main module to run our system. In addition to the modules we used from Phase 1, we also added another module called RoutingInfo to hold and use all of the necessary router information. As in Phase 1, we used IntelliJ IDEA as our IDE and the GitHub remote storage in order to easily and efficiently make changes to and share all of our project files between each group member.

# Design Modules and Implementation:

For our Peer-to-Peer system, we implemented 7 modules. These modules were: Main, TCPServerRouter, SThread, TCPServer, TCPClient, Stats, and RoutingInfo. To run the processes of our system, 2 or more machines are required. During the implementation and testing of our system we used 3 machines.

The execution of this system is managed by the Main module. In order to run this module, the user must first change the default IP and port of the router they wish to connect to, routers in the known subnet, the client name, and the server name. Then, when running Main, the user can select whether they want to run a router, server, or client. The user is also allowed to run multiple instances on the same machine, making it possible to run multiple routers, clients, and servers on the same machine.

We will now discuss in detail each of the modules, how they function, and how they are set up in our system, starting with the Main module.

The Main module allows users access to a very simple interface that allows them to run a router, a server, or a client. The server and client methods have the option to be run 50 times each as well. The main class allows the user to pre-setup a subnet list with IP's of known routers, as well as ports. Additionally, once the routers are configured, the users can select names for the clients and servers that they would like to connect to, as well as the input text file for the client to send.

The TCPServerRouter module sets up and runs the router on a specified port. The Subnet list is also passed through the constructor. When run, the TCPServerRouter serves as the main thread and spawns SThreads whenever a new client connects. The router then passes on the handling of the client to the SThread along with a copy of the socket, the local RoutingTable, and a RoutingInfo object for the new client.

The RoutingInfo object holds the ipAddress, port, Socket, usage status, and client name of a connecting entity (router, server, client). If the object is a router, it has a boolean called isRouter set. The object comes with getters, setters, and a dedicated toString method.

The SThread module is derived from the Thread class and it handles connections to a client for the router. The constructor takes in information about the parent router (ip, port), the local RoutingTable, and a RoutingInfo object with information on the client that has connected. The constructor creates input and output streams for the client, and sets private global variables to be used when the SThread’s run method is called. When the thread is ‘run’, it firsts attempts to get the client’s name and stores the value in the dedicated RoutingInfo object. If the connecting client is another router, we run the ‘router’ method, otherwise we run the ‘clients’ method.

For the ‘router’ method in SThread, we know that routers can only be clients if they are searching the subnet for a node, so we take in the name of the node it is looking for. If the node does not exist within our subnet (our RoutingTable), we tell the router “Bye Bye Bye”. If the node does exist within the subnet (our RoutingTable), then we set socket to read through to the connected client, setup the input and output stream, and return “RingADingDing”. This lets the connecting router know that the node was found and is ready to receive data. Then, until the client says “Bye.”, the routers act as a bridge between the nodes and pass data through.

For the ‘clients’ method in SThread, we know that the first message sent from a client/server is always the name of the destination node it wishes to connect to. So we save the client info, and wait for changes to propagate through to our routing table. If the destination exists within the subnet (our RoutingTable), then the input streams and output streams are created and a clientFound variable is set to true. If the destination is not found in the subnet, the router queries other routers that it knows of that exist within its routing table. If the node exists within other subnets, then input and output streams are created and clientFound is set to true. Otherwise, clientFound is set to false, queries are terminated and the process ends with the result that the node could not be found. Finally, if clientFound returned true throughout that process, the router begins passing data from its client to its destination node until it receives a “Bye.” At that point, the last termination message is sent, and the SThread terminates.

The TCPServer module establishes and runs a “server” connection to a router of the user’s choice. First it sends the router its own name for reference, and the name of the client it wishes to connect with. Once connection is established, the only purpose of this class is to take each message it receives from the client, convert all of the text from lowercase to uppercase, and return the uppercase messages. Once it receives a “Bye.” message from the client, it exits.

The TCPClient module establishes and runs a “client” connection to a router of the user’s choice. First it sends the router its own name for reference, and the name of the server it wishes to connect with. Once connection is established, the class reads in from a pre-specified file. As long as there is data within the file, the client will continue to send data through the router to the server. Once it sends a line, it waits for the server to reply before sending another line. Once the entire file has been sent, the client sends the termination response, “Bye.”, calculates statistics, prints out the results to a CSV data file, and then terminates.

The Stats module is used to manage connectionTime, averageTransmissionInSize, averageTransmissionOutSize, AverageTransmissionSize, list of transmissionInSizes, list of transmissionOutSizes, averageTransmissionTime, list of transmissionTimes, efficiency of sending data, averageEfficiency for sending data, and the list of efficiencies for all the data sent in a run. The Stats can also print out the results as a string, as well as print the data to a CSV file for analysis using Excel or comparative software.

# Client Sample Output:

# Data and Analysis:

For this analysis, we used two different text files to test the efficiency of our program. This would give us an idea of how the network could potential effect the statics of our program and using different file sizes. The first text file (we will refer to this as File 1) we used was 6 Kilobytes and the second (we will refer to this as File 2) was 11 Kilobytes, almost twice as big as the first file. The data we collected from executions of the code were outputted to a excel file and contained the values: Connection Time, Transmission Out Time, Average Transmission In Time, Average Transmission Time, and Average Efficiency. For our analysis of the data we recorded, we calculated ranges, averages, and modes (which value occurred the most frequently) of the data values.

The connection time is the first statistic recorded, since connecting is the first action that takes place in the program. For File 1, the range on the data for connection time was from 9991 to 10137 milliseconds. The mode was 10003ms and average was approximately 10014ms. For File 2, the range on the data for connection time was from 20080 to 23025 milliseconds. The mode was 20083ms and the average was approximately 20194ms. We believe the greatest influence on the differences of connection times was due to the network. During the testing for File 1, all the clients were on the same subnet. While, for File 2, the clients were on different subnets.

Next, we will look at Transmission Times. For File 1, the Average Transmission In and Average Transmission Out Sizes were static at 267 characters. The Transmission Time ranged from 3 to 55 milliseconds. The mode was 3ms and the average was approximately 7ms. For File 2, the Average Transmission In and Average Transmission Out Sizes were static at 535 characters. The Transmission Time ranged from 60 to 126 milliseconds. The mode was 63ms and the average was approximately 75ms. The Transmission Sizes for both files were static due to the nature of the program, since the returned transmission is just a conversion to all capital letters of the client’s data. The differences in Transmission Time is most likely related to the differences in character/ file sizes, with the possibility of minor network constraints.

Lastly, we analyzed the efficiency of the program. For File 1, the efficiency ranged from 248 to 267 characters/ milliseconds. The mode was 267chars/ms and the average was approximately 266chars/ms. For File 2, the efficiency ranged from 523 to 535 characters/millisecond. The mode was 535chars/ms and the average was approximately 535chars/ms. We ran into an error here with calculating our efficiency for our execution. The code was executing so fast that all the data was being send in less than a second, which resulted in a divide by zero error. This explains the differences in efficiency compared to transmission times for both files. The differences in the efficiency of the program of the two files is most likely due to the differences in character/ file sizes, with the possibility of minor network constraints as well.

In conclusion, the majority of the differences in statistical data for the two file is largely due to the differences in file sizes and possible network constraints. There were correlation in amongst the data as well, such as a jump in the connection time resulted in a dip in the average efficiency during one execution. Another obvious correlation was the jump across all data when the larger File 2 was used for the input file instead of the smaller File 1. There appeared to be very few outliers among all of data as well. We believe most of the spikes in our data, besides when the input files were changed, is largely due to network constraints.

# Conclusion:

In Phase 2 of this project we implemented and analyzed a Peer-to-Peer distributed system that allows multiple pairs of nodes to exchange messages over a server-router bridge. By incorporating our programs from Phase 1, we were able to see the similarities and differences between the Client-Server system and the Peer-to-Peer system. The completion of this phase of the project has provided us with more knowledge and a better understanding of the Peer-to-Peer paradigm.