**Kennesaw State University**

**College of Computer Science and Software Engineering**

Course : CS 4504 – Distributed Computing

Fall Semester 2017

Instructor : Dr. Patrick Bobbie

Due Date : December 3rd, 2017

***Term Project Phase Two:***

***A Study of Message Passing Within a Peer-To-Peer Paradigm Implementation Using Java Socket API***

Contributors:

Michael Bourgault | [mbourgau@students.kennesaw.edu](about:blank)

Erica Pantoja | [epantoj2@students.kennesaw.edu](mailto:epantoj2@students.kennesaw.edu)

Colby Dozier | [cdozier3@students.kennesaw.edu](about:blank)

Abstract

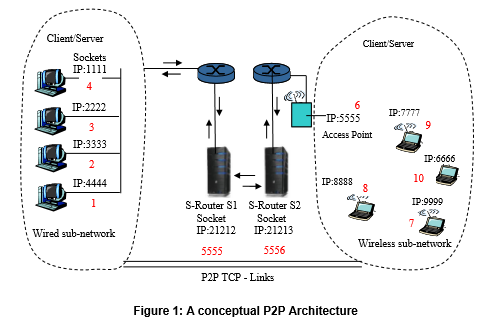
Compared to the client-server system, the peer-to-peer system presents many advantages that make it preferable in certain types of communication. In this study, we have been asked to implement a peer-to-peer system using the Java Socket API. Implementing this design presents many challenges including I/O management, passing IP addresses, and simply collecting data correctly. At a glance, our design is a small simulation of the peer-to-peer paradigm uses 4 separate terminals, two to hold the peers, and two to hold the respective server routers. By testing our design and gathering data on message size, routing table lookup time, transmission time, and average dropped messages, we determined that, within the peer-to-peer paradigm, there are correlations between some of these values. We further explain the challenges presented in implementing this design, the differences between the peer-to-peer and client-server models, and the overall results of our study.

Introduction

In order to build a peer-to-peer paradigm “from scratch,” first the design had to be drawn out and made very clear. The peer-to-peer paradigm and client-server paradigm are two fundamentally different distributed computing paradigms that carry with them some simple, yet stark differences. Early on in research, we came across some of the largest design differences between the two, which later became some of the greatest challenges associated with implementing this design. Our first challenge came from the concept of peers.

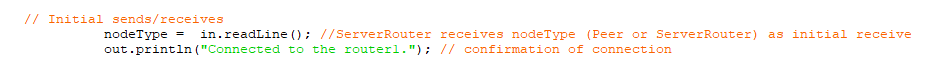
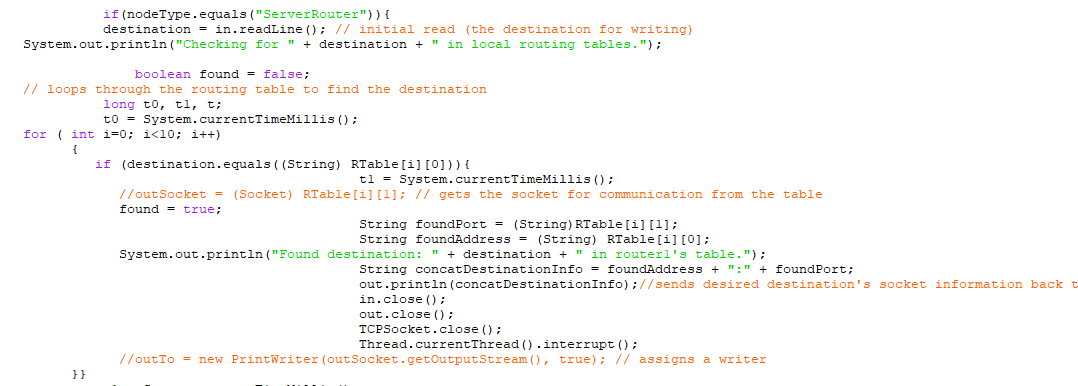
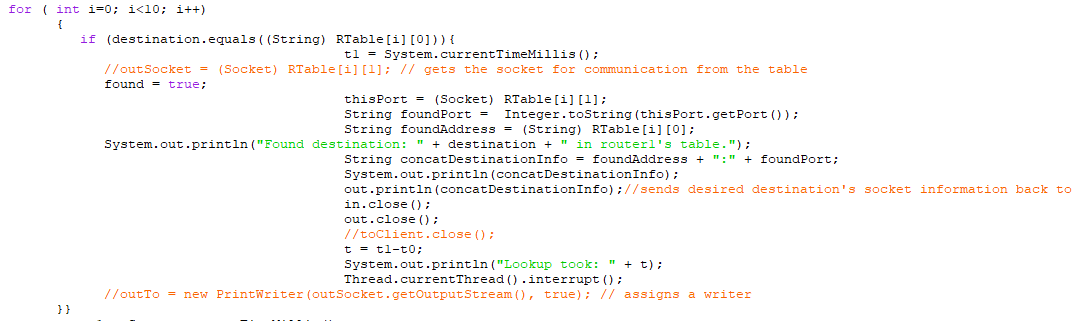
In looking closer at the applications of peer-to-peer versus client-server design, a very big difference becomes clear. In client-server, the participating processes play separate roles from one another. The client, or sender, issues requests and awaits responses while the server, or receiver, listens passively until a request comes in, processes, it and sends required output (Liu, 79). So, why is this so different from peer-to-peer? In peer-to-peer, there is no difference between the client and server. Instead they are referred to as “peers.” A peer is a term for a node that can both initiate requests and provide responses to requests. In this implementation, one of our first big challenges was to design a “peer” using the Java Socket API from the already established TCPClient and TCPServer files. We decided that designing this peer as an amalgamation of the client and server files would be the simplest way to simulate a true peer.

Another big difference in the peer-to-peer design is the communication between server routers. In a more practical sense, the internet is comprised of server routers, servers, and peers. In this study, we needed to design a method of having the server routers send vital information about the connection the two peers are trying to create, but also know when to include information from their own routing tables in this information. Establishing a socket connection between the two routers was relatively simple, but designing them so that they know when to search their routing table or “pass it off” to the other router was a bit trickier.

One more big challenge presented with the peers comes from the fact that the peer-to-peer paradigm is based on establishing direct connections with a peer in another sub-network. The initial request message sent by one peer must have the necessary information and be sent through the correct channels (routing tables in this case) to gather the info about a peer they want to connect with. The peers on the receiving end must also understand the information being sent to them and by whom it is being sent to establish a connection and transfer the message. This was perhaps our biggest challenging in implementing the peer-to-peer paradigm.

For reference, **Figure 1** was the goal design of this study. In our design we have implemented this with only a few minor changes. The major points of this design were to include a different socket for each client, and have four separate IP addresses (terminals) running each sub-network (including multiple nodes) and each server router. At the bottom, the figure illustrates the direct TCP links established after the routing information has been furnished for the requesting peer. For the remainder of this study, we will refer to this design as our “goal design.”

Design Approach

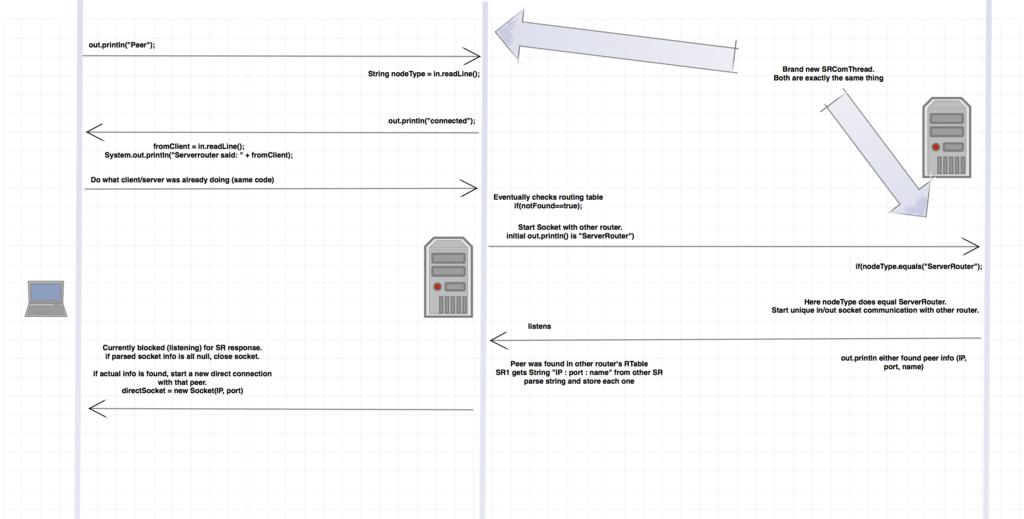
With our goal solidified, designing the approach was an important next step. The three major challenges described in the introduction were our focal points in our design approach because the rest of the code was provided for us. Within, we will discuss these three challenges as the server router connection problem, the peer design problem, and the initial request problem. Overall, this study required an improved understanding of the Socket API most of all, but also a logical plan behind each class that we designed. We started with the server router connection problem, but it’s important that we first discuss the initial request problem. This was pertinent because these two challenges ended up going hand in hand. When a peer wants to establish a connection with another peer, it must first send a request to the server routers to gather that peer’s IP address and socket. In our design, this request carries with it an established name of the peer it is trying to connect with. For instance, a peer might be known as B1, which insinuates that it is on the B router and it is the “first” node. As we discussed earlier, the server router had to be modified so that when it receives a request from a peer, it knows to search its own table, but in the event that it does not find this name that we have just described from the initial request problem, it will send the message on to the other router, and that router will search its table. The initial request problem is mostly handled by attaching names to each of the peers on either side, but what about the server router connection problem? Our solution is to have the server routers distinguish between different types of connection based on what we have called “node type.” **Figure 2** shows the code that is responsible for making sure this occurs. Within the two separate cases, “client” or “server router,” there are two vital tasks that must be performed, also illustrated in **Figure 2**. For more understanding on this implementation, refer to the section titled Simulation.

**Figure 3:** Recognizing “nodeType”

After solving these two problems, the only challenge left to deal with was the peer design problem. As stated previously, to achieve the goal design, the peers are required to not only receive information, but also request information. In this study, the other goal is for the peers to each be able to take in input from another peer through a direct TCP connection and return it as the server did in Phase 1, fully capitalized. We decided to include many classes in each “peer” to give it the ability to not only transmit like a client, but also process requests like a server. Naturally, we used our already designed Phase 1 implementation for Client-server, but added some modifications. Our solution was to include the files illustrated in **Figure 5** in our peer nodes. Some of these files could have been combined, but we thought the abstraction would prove to be helpful in understanding the implementation further. The Node Setup classes would create threads and assign sockets for each of the nodes to be created. Client and server serve the functionality purpose that they did in Phase 1, in that client sends requests and server waits to receive code, but in this case they work with the different threads that represent the nodes of the peers. This is also where the Server Thread class comes into play, as well. For more on the design of each class itself, refer to the Architecture section. With a clear design approach for each of the problems described in the Introduction, we were able to begin our implementation.

**Figure 2:** nodeType Recognition Code

**Figure 3:** Peer-side Structure

Architecture | Diagrams

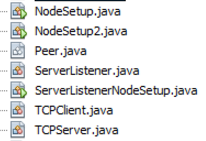
**Figure 4:** Peer to Server Router to Server Router Architecture

**Figure 4:** Peer-Router-Router connections

This section is comprised of some of the design diagrams that allow us to better illustrate the architecture of our attempt at the goal design. **Figure 4** shows the basic design of our node structure. As is apparent, the peer initializes a request by sending an initial message including its “nodeType” to the server router. The server router then establishes a connection and lets the peer know that it is connected. From here the peer then uses the TCPClient class to send the information regarding the destination to the server router. Once the server router receives this information, it will check its routing table because it knows that it is connected with a peer “nodeType.” If the server router fails to find the desired destination within its routing table, it triggers a separate socket by which to send the information to the second server router. In this case, the second server router is prepared to read in the “nodeType” from the first router and it will therefore know to check its routing table for the name information being passed in through the peer. In the case that it doesn’t find the name, it sends a string back through the server router and to the client that says there is no address for this name. In the case that it does find the name and address, the server router returns this information in much the same way it would return the error, and the peer is now knowledgeable about the IP address of the peer it is trying to connect to.

Given this information the peer is now able to establish a direct TCP connection in much the same way that the client did with the router in Phase 1. The difference here is that the TCPClient class of one peer will be connecting directly to the TCPServer class of another peer, without going through a server router. Once we understood how to retrieve the IP address from the routing tables, the rest of the architecture was already laid out for us. For more information on the modules associated with the implementation, refer to the Implementation Modules section.

Implementation Modules

 For this design, there are two distinct sets of classes that we used to create a server router and a set of peers. **Figure 5** lays out the sets of modules we used, divided into their overall purpose, server router or peer. Within the section titled Design Approach, we broke down the modules within the Peer. In this section we will discuss the design of the Server Router and how each of these classes work together to complete the overall goal design.

**Server Router Classes**

**Figure 5:** Our Modules

**Server Router Classes**

**Peer Classes**

Threads were a major part of completing this design. We used the Server Thread in the Peer class to create separate connections to each of the various nodes on each of the ports. In practice, this served its purpose very well and saved a lot of time in adding new separate classes with different port numbers. In the case of the server router, a threading process was used to establish communication with both the servers (peers) and the other server router in SRThread. By designing this so that each time a connection needed to be made anywhere the SRThread is responsible for opening a socket and then later closing it. The SRThread also communicates directly with the routing table for the server router. Without this design, the implementation would not work for multiple peers on each side because it would not be dynamic.

It is important to remember that each module we used served a specific purpose in the total system. In our system, it was only necessary to create two separate entities, a sub-system of peers, and a server router with a routing table. From there, since each of those were built with the goal design in mind, the entire design is capable of being replicated. In this case, the server routers and clients mirror themselves. In the case of the peers, they are made up of three true modules (TCPClient, TCPServer, and ServerListener). The purpose of the remaining classes is either to instantiate the main threads or to get data (The Peer class). NodeSetup instantiates TCPClient, NodeSetup2 instantiates TCPServer, and ServerListenerNodeSetup instantiates ServerListener. For more information regarding the specific implementation within these modules, read into the Simulation section.

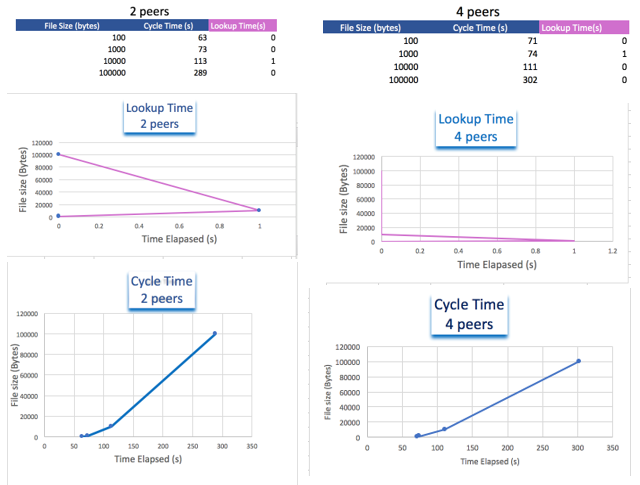
Simulation

The simulation begins with the server router. Once the server router is running, it actively listens on port 5901 for any incoming connections with the usage of ServerSocket. As it is waiting, the TCPServer is started. This starts a new socket with the server router. Once the server router accepts the connection it is handed off to SRThread and goes back to listening for another incoming connection. So with the TCPServer instance connected with SRThread as of now, the SRThread currently awaits an initial send from the instance. The initial send from the peer’s server instance is,”Server”. This is stored in String nodeType after being read in. The server router then sends a connection confirmation back to the peer’s instance. After sending the confirmation, the nodeType that was read in is compared to,”Client”,”Server”, or “ServerRouter”. The logic that is executed next depends on what initiated a socket with the server router. At this point the type that connected was the server instance of the peer. So naturally, the conditional with “nodeType.equals(“Server”)” is true and the block of code under it is stepped into. This results in “while(true){}” to be started. This leaves the socket between the server instance of the peer and the server routed indefinitely connected until one of them ends the connection. This leaves the peer that is connected to the server router in the server router’s routing tables until the connected is ended.

Next, the ServerListener is started. This mimics the TCPServerRouter code to an extent as it waits for any incoming connection with a server socket. As of now the ServerListener is blocked until a connection is accepted. This leads us to the final instance that has to be started for this network architecture to be fully used. The client is instantiated with it knowing its own server router’s information. The user is asked to input what destination they want to connect to on the client side. As with the other instances used, Client also sends what type of interface it is. It sends,”Client” as the initial send to the connected server router. After receiving a connection confirmation the Client instance sends the address the user inputted to the server router. This is compared to each peer currently connected to the server router through a simple for loop. If the destination is found within the current server router’s routing table, the address and port of the peer are gathered from the stored socket and address in the routing table. Once the client receives the peer’s information from the server router the connection with the server router is closed alongside in and out streams. With the new peer information stored in memory, the Client tries to establish a new socket with the peer directly. This is where the ServerListener interface comes into play. This architecture has an established protocol of the ServerListeners listening for incoming connections on port 6000. Kind of similar to how SSH is handled through port 22. So the Client instance attempts to connect to the ServerListener instance of the peer it wished to connect to with the information it gained from the server router. Once a connection is established, the client starts sending messages directly to the listener to have converted into uppercase characters. The converted characters are then sent back to the client. This is done by line. After the conclusion of the client and listener conversation, relevant sockets are closed and the listener goes back to waiting for an accepted connection.

The above piece is if the destination is found in the first server router’s routing table. If the destination is not found in the first server router’s routing table, the server router attempts to establish a new socket with the other known server router. The same applies before. The standard is for all server routers to listen on port 5901. So both server routers know the port of the other server router. Once connected the server router that initiated the connected sends,”ServerRouter” as the initial send. This tells the other router that a server router has connected to it and the logic within the block of code belonging to the conditional of “nodeType.equals(“ServerRouter”)” is stepped into. This results in the server router checking its routing tables and responding with either the peer’s information or that it was not found. The original server router then uses this received information to inform the Client instance of the peer’s information or that it was not found. If the peer was not found the Client instance closes all sockets and any readers or writers.

Data Results and Concept Relation



**Figure 6:** *Top-row*: 1 and 2 pair of peers (2 and 4 peer) test with 4 different message sizes

*Middle-row*: Routing Table Lookup Time By Number of Peers

*Bottom-row*: Message Transmission Time By Number of Peers

By testing the system with different message sizes, we tested for a possible relationship between message size and cycle time. In the peer-to-peer paradigm, there are only a few singular differences that might affect the overall cycle time for a message to be transmitted as compared to the standard set by the Client-server paradigm. In our design, which was very close if not indistinguishable from the goal design, the routing table lookup times are almost negligible at values from 0 to 1 ms. As well, in sending 5 messages of different sizes, we noticed a slight uptick in transmission time based on the size of the message, but no difference in the routing table lookup time.

A healthy skepticism of our results is important, so we tested a few separate times to see if there are any errors in the system, or if we would have any lost messages. Since we chose not to work with larger files like images and videos, our data does not show some issues that may be a result of larger, encoded files. In terms of major differences between the client-server paradigm and the peer-to-peer, we noticed that, based on our results from Phase 1, the peer-to-peer is a slightly faster method of sending messages on average. Once a direct connection is established, it is very easy for the peers to send and receive data with not much transmission time in between. Also, they do not have to consult the server router each time they want to send a message, but rather they are already directly connected and can send and receive at will.

The practical uses of peer-to-peer are numerous, but in our eyes, this is a very efficient way to download and “stream” large video files to a single peer. As opposed to client-server, however, the peer-to-peer connection is a bit more complicated to re-establish if it gets lost. In a more practical case, one peer must search at minimum through two separate routers’ routing tables to find the destination IP address for a message. While this is no big hassle for some data, very large files can become a nuisance. In this case, peer-to-peer is not necessarily the solution.

Conclusions

Given our results, the goal design we were looking for was achieved. While we did not get a chance to test our implementation with more than 4 peers, our design allows us to create peers dynamically and the results that we gathered from the tests we did gave us good insight into the trends of the peer-to-peer paradigm in terms of transmission time and message size.

We now accept that the larger a single message being sent in this paradigm, the larger the transmission time. Based on our results, we have not seen any correlation between routing table lookup time and message size, which leads us to conclude that there is no correlation between these two values. We did not lose any messages in our testing, so the average dropped message was 0. This is a good sign meaning that we are addressing the messages correctly, and that our internet connection is not losing any message. It is important to note that since we did not collect any incredibly large number of peers, this conclusion is not necessarily indicative of the efficacy of our system.

This study was a success in that it produced a successful peer-to-peer implementation that works over any group of 2 or 4 terminals with only a small amount of adjustment. Comparing our implementation in Phase 1 to Phase 2, we see various design flaws in the original system that we designed. With a broader understanding of the Java Socket API, we were able to better design this system with fewer challenges in the implementation itself. The challenges we faced in this study led to a greater understanding of the TCP connections, Distributed Computing Paradigms, and Peer-to-Peer more specifically.

Given more time to further study this implementation, our first move would be to tweak the design so that it allows a friendlier graphical user interface so that users can test 100’s and 1000’s of nodes without having to manually adjust the actual source code. Another part of the design that would be nice to change is giving the peer the ability to send and receive audio, image, and video data. A study concerning these data types might prove different results to our own, and could considerably change our conclusions about the P2P paradigm.

References

Liu, M. L. (2004). *Distributed computing: principles and applications*. Boston, Mass: Pearson.