Time Efficiency in a Distributed Chat Application

Xenofon Georgitsaros
Mediterranean College
University of Derby
Thessaloniki, Greece
x.georgitsaros@mc-class.gr

***Abstract*— TCP is one of the major protocols of the whole Internet protocol set. TCP is connection-oriented, and a client-server connection is established before data is sent. TCP delivers dependable, organized and error-checked streams between applications running on hosts communicating through a network. Socket programming helps in the connection of two nodes to interact with each other over a network. In this paper, I propose a method to create a chat application for testing purposes the elapse time of message delivery using the TCP protocol and socket programming. In this application, there is a Graphical User Interface (GUI), using Java Swing and AWT technologies. Server and Client code is written in Java programming language. The aim of this chat application is to send a message to server and then, server broadcast this message to its clients and measure the elapse time.**

Keywords—Client-Server architecture, Distributed, Chat application, Elapsed time, Message broadcasting, Multiple clients, Multiple servers, TCP sockets, Java Swing

I. INTRODUCTION

Computer systems has been dramatically changed since the middle 80s and they continue to change. At the beginning of the modern computer era, computers where big and costly. Also, these machines operated separately from each other because there was not a way to connect them.

However, two technological improvements started to change that circumstance. The first one was the invention of microprocessors. Most common were the 16, 32 and 64 bit CPUs. With multicore CPUs, we have the capability to create programs using parallelism. In any scenario, the latest computer generation has the mainframe computing power deployed in the past 30 or 40 years, but for 1/1000th or less of the cost [1].

The second innovation was the development of high velocity computer networks. Local-area networks or LANs enable thousands of computers to be linked within a building or a university so that small amounts of data can be shared in just a few milliseconds. It is possible to move larger amounts of data between computers at rates of billions of bits per second (bps). Wide-area networks or WANs make it possible to connect hundreds of millions of computers across the globe at different speeds [2].

Alongside with the advancement of increasingly powerful and connected computers, we have, also, seen the diminution of computer systems with the most magnificent result being the smartphone. Equipped with sensors, plenty of memory space and a strong processor are making these machines equivalent to fully developed computers. Obviously, they are also capable of connecting to network.

As a result, the above technologies made it very simple to set up a computing system consisting of many networked computers. These computers are globally scattered, which is why they are commonly said to make up a distributed system. A distributed system's scale can range from a few machines to millions of computers. A distributed system's scale can range from a few machines to millions of computers. The linking network can be wired, cellular, or both. Also, distributed systems are in many cases highly flexible, in a manner that computers have the capability to enter and exit, with the underlying network's topology and efficiency almost constantly changing [1].

What is a distributed system? Generally, there have been multiple definitions of distributed systems. But the most suitable definition is the following:

"A distributed system is a network that consists of autonomous computers that are connected using a distribution middleware. They help in sharing different resources and capabilities to provide users with a single and integrated coherent network [3]."

The above description applies to two distinguishing features of distributed systems. The first is that a distributed system is a set of computer components that can function separately from each other. A computing element, usually referred to as a node, may be either a hardware device or a software process. The second is that users believe they have a single system to deal with. It means that the autonomous nodes need to cooperate by some means [1].

Distributed computing has led many programmers to develop various chat applications based on different architectures. One widely-known architecture is Client-Server. Client-Server architecture is a distributed application system that divides function or workload between resource or service providers, named servers, and data requesters called clients. In the client server model, the server acknowledges the requesting process and transfers the required data packets back to the client when the client device sends a request for data to the server. A server provides its service for multiple clients not just for one. Thus, it can be assumed that the client-server meets the many-to-one arrangement. Several clients can use one server's service. However, clients do not allow others to use any of their resources. Facebook, World Wide Web, etc., are some instances of the Client-Server model.

The aim of this paper is to develop a testing chat application that user will be able to broadcast a message to other clients. User will have the capability to change the number of servers and clients in order to find the most time efficient case, concerning the delivery of message to clients.

In this paper there are four sections. In section II, I present the related work of other researchers. In section III, I present the design of my system. In section IV, I provide a table with my testing results and in section V, I analyze the results of the previous section.

II. RELATED WORK

In this section, they are analysed some other examples of chat applications, which methods they chose to implement them and if they encountered any failures.

A. *Examples of Other Chat Applications*

Malhotra et al. implemented a chat application using sockets based on UDP protocol that allows the acknowledgement feature after the sending of each message. They did not select the TCP protocol because it does not provide the feature of multicasting. Malhotra et al. liken their application to a dedicated chat server with a server and multiple clients. Upon client and server configuration to connect they could accomplish many devices to interact through Peer-to-Peer (P2P) communication, multicasting and file sharing. Finally, they experienced different types of deficiencies, during their implementation, for which they suggested a solution [4].

Sabah et al. introduced a chat program that offers end-to-end protection and allows private details to be shared securely with each other without caring about privacy. In addition to data security. This paper provides a list of requirements for creating a safe chat program and the framework was developed based on these specifications. Client- Server is the suggested architecture for the implementation of their chat application. On the client side, as soon as, the user installs the program, they have the option either to register or log-in. The chat server on the server side consists of two servers. A server for users and a server for messages. The user's server is to manage the data of each user of the application. Message server uses Firebase Cloud Messaging (FCM) to manage messages between users. If the receiver is unavailable, the messages will be retained on the FCM list for a specific period of time and when the recipient is online, these messages will be transmitted to the user, and removed from the queue [5].

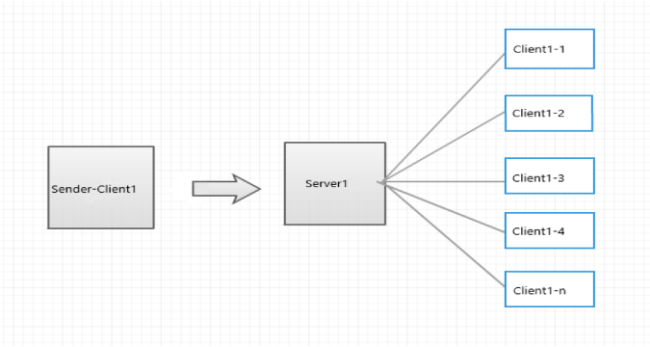
In this study, Mohamed et al., suggested a chat program based on peer-to-peer architecture that entirely excludes centralized or third-party components. Users control the system, and the communicating parties handle its protection autonomously. Before exchanging messages, each user will have their own database for peer's profiles and communication parties authenticate with each other. Finally, this chat application has completely been designed with emphasis on security measures [6].

III. SYSTEM DESIGN

A. *Requirements*

- *Scalability:* System should be able to increase the number of both clients and servers.
- *Communication:* Clients should be able to connect to servers.
- *Send message:* Sender-client should be able to send message to servers
- *Broadcast message:* Servers should be able broadcast the sender-client's message to clients
- *Show Time:* System should be able to measure the elapse time that message made to reach clients

B. *Depiction of Proposed System's Architecture.*



C. *Technologies*

- *Java programming language:* Java is an object oriented language that allows user to create classes and objects.
- Java Swing tutorial is a component of Java Foundation Classes (JFC) for developing window-based applications. It is set up at the top of the AWT (Abstract Windowing Toolkit) API and written entirely in java. Java Swing offers components that are lightweight and platform independent.
- TCP socket programming in Java. Socket programming is a method to connect two nodes to communicate with each other over a network.

D. *Proposed System*

Based on research that I have done, I considered Client-Server architecture is the most appropriate model to implement the chat application. The development of the whole project was based on Java language and socket programming. The sockets used in the application are TCP sockets. The development of GUI uses Java Swing and AWT technologies.

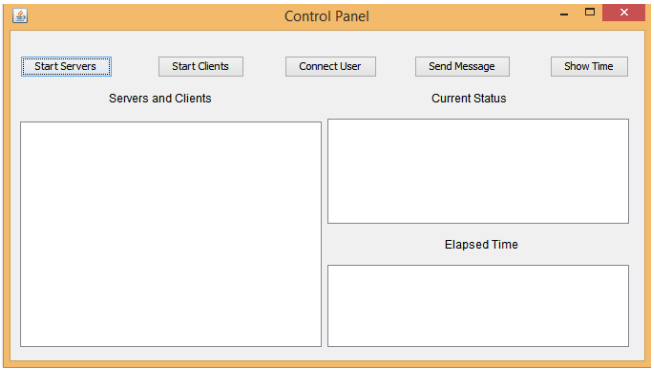


Fig. 1. This is the control panel of the application.

In Fig. 1, there is a form where the user of the application can select the number of servers. After that, in Fig. 2, another form shows up that asks from user to enter the number of clients they want to connect to each server. In Fig. 3, user can check the number of servers and the number of clients that is standing by to run will their test. Then, user can start the servers through the "Start Servers" button in control panel. Servers are listening and waiting for a connection with clients, as shown in Fig. 4. In Fig. 5, user can check if the connection among servers and clients is established. After that, if clients have connected to servers successfully, user connects the "sender-user" to each one of the servers in order to send the same string message to them, as it appears in Fig. 5 again.

Then, servers will broadcast this message to clients. Finally, in Fig. 6 user can press the "Show Time" button to see the elapsed time that message made to reach the clients.

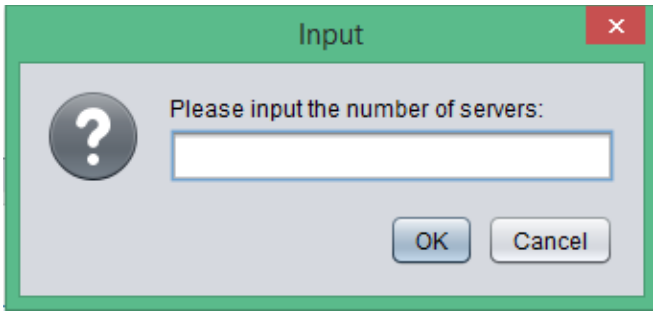


Fig. 1. Windows form to enter the number of servers

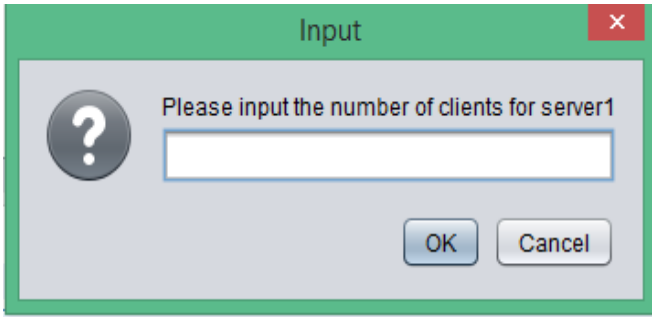


Fig. 2. Windows form to enter the number of clients for each server.

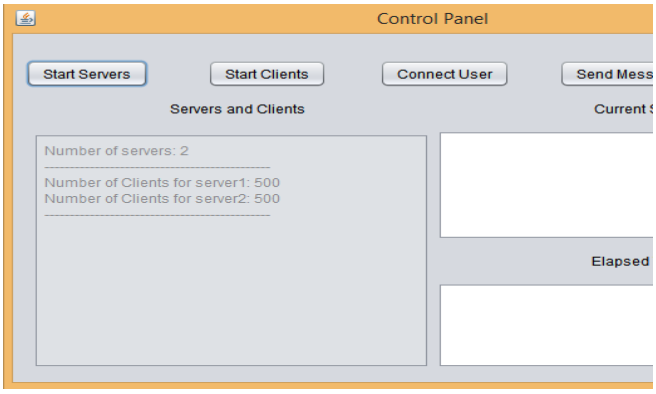


Fig. 3. This figure shows that user added to system 2 servers and 500 clients on each server.

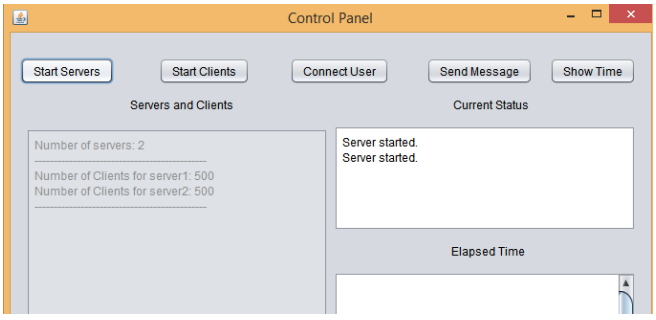


Fig. 4. This figure shows that user started the servers and servers waiting for connection.

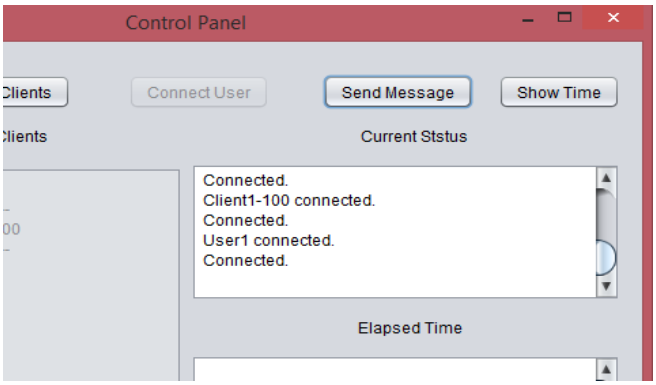


Fig. 5. This figure shows that clients and sender-user has connected successfully to server.

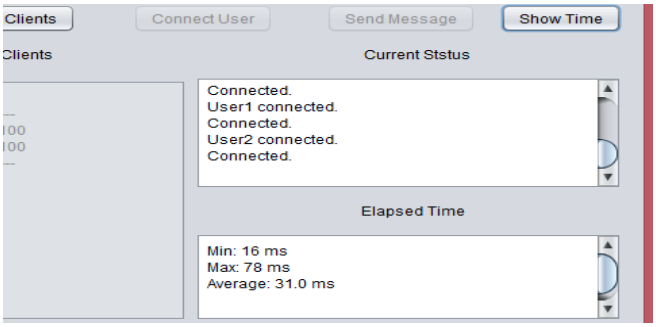


Fig. 6. This figure shows the minimum, maximum and average elapsed time of the message.

a) Implementation of server side: First of all, in order to establish a connection, server socket object is initialized, and a socket object constantly accepts incoming connection within a while loop, Fig. 7. For the stream obtaining, the inputstream object and the outputstream object are retrieved from the socket object of the ongoing requests, as it appears in Fig. 8. After that, in Fig. 9, a new clientHandler object is created, after the streams and port number have been obtained, and the method start() is called on this newly created object. In Fig. 10, ClientHandler class is responsible for handling various client requests and it holds name of the client that is connected to the server.

```
public void run()
{
    printOutStreams = new ArrayList<PrintWriter>();
    users = new ArrayList<String>();

    try {
        ServerSocket serverSoc = new ServerSocket(inputPort);
        sockets.add(serverSoc);

        frame.broadProcess("Server started. \n");

        while (true) {
            Socket clientSoc = serverSoc.accept();
            PrintWriter writer = new PrintWriter(clientSoc.getOutputStream());
            printOutStreams.add(writer);

            Thread tListener = new Thread(new ClientHandler(clientSoc, writer));
            tListener.start();
            frame.broadProcess("Connected. \n");
        }
    }
    catch (Exception ex) {
```

Fig. 7. This figure shows the initialization of server socket object and a while loop where a server socket object continuously accepts incoming connection.

```
Socket clientSoc = serverSoc.accept();
PrintWriter writer = new PrintWriter(clientSoc.getOutputStream());
printOutStreams.add(writer);

socket = clientSocket;
InputStreamReader inReader = new InputStreamReader(socket.getInputStream());
buffReader = new BufferedReader(inReader);
```

Fig. 8. This figure shows the InputStream object and the OutputStream object for stream obtaining.

```
Thread tListener = new Thread(new ClientHandler(clientSoc, writer));
tListener.start();
frame.broadProcess("Connected. \n");
```

Fig. 9. This figure shows the ClientHandler object and the call of start() method.

```
public class ClientHandler implements Runnable{
    BufferedReader buffReader;
    Socket socket;
    PrintWriter client;

    public ClientHandler(Socket clientSocket, PrintWriter user){
        client = user;
        try {
            socket = clientSocket;
            InputStreamReader inReader = new InputStreamReader(socket.getInputStream());
            buffReader = new BufferedReader(inReader);
        }
        catch (Exception ex){
            frame.broadProcess("Exception error. \n");
        }
    }
```

Fig. 10. This figure shows the implementation of ClientHandler class.

b) Implementation of client side: Firstly, a socket connection is established. Communication happens with the use of BufferReader and PrintWriter, as it appears in Fig. 11.

```
public void Connect(){
    if (connected == false) {
        try {
            sock = new Socket(address, port);
            InputStreamReader streamreader = new InputStreamReader(sock.getInputStream());
            buffReader = new BufferedReader(streamreader);
            pWriter = new PrintWriter(sock.getOutputStream());
            frame.broadProcess(username + " connected.\n");
            connected = true;
        }
        catch (Exception ex) {
            frame.broadProcess("Unable to connect! Try Again. \n");
        }
        ListenThread();

    } else if (connected == true) {
        frame.broadProcess("You are already connected. \n");
    }
}
```

Fig. 11. This figure shows the connect() method in client class.

c) How these classes work together: Every time a client sends a request to connect to a server, the server assigns a new thread to manage the request. Access to streams is given to the newly assigned thread for communication with the client. Once the new thread has been assigned the server comes back into accepting state through its while loop. When another request comes when the first one is still in progress, this request is accepted by the server and a new thread is assigned for processing. This enables the handling of multiple requests even when some requests are in process.

Research question #1: How elapsed time of a message delivery is affected and how can we achieve time efficiency in a chat application?

Research question #2: How many clients does a server can host?

IV. System Analysis

A. *Results of My Application.*

TABLE I.

	Clients[1]		
Servers	500	1000	2000
1	Min: 16 ms Max: 172 ms Avg: 150.0 ms	Min: 16 ms Max: 250 ms Avg: 201.0 ms	Min: 31 ms Max: 500 ms Avg: 390.0 ms
2	Min: 47 ms Max: 204 ms Avg: 166.0 ms	Min: 47 ms Max: 347 ms Avg: 260.0 ms	Min: 94 ms Max: 3219 ms Avg: 1379.0 ms
4	Min: 100 ms Max: 532 ms Avg: 490.0 ms	Min: 100 ms Max: 4297 ms Avg: 1320.0 ms	Min: - Max: - Avg: -

[1]Number of clients on each server.

The above table, Table I, is referred to the testing results of my chat application. In Servers column, there are 3 numbers. These numbers show how many servers were added in each testing run. Below Clients line, there are another 3 numbers. These numbers show the number of clients connected on each server. So, from this table, arise 9 different testing runs. For example, my first testing run was with 1 server and 500 clients connected to it. The second testing run was with 2 servers and 500 clients and so on. The variables min, max and avg stands for the minimum, maximum and average time, in milliseconds, that message made to reach clients from sending. Finally, in the last cell there are no numbers, because my pc crashed in that testing run.

V. Critical Review

First, I want to mention that this application was tested on a medium-priced computer, so the results may differ on other PCs.

Research question #1: By examining the results' table, in the System Analysis section, I observed that keeping the number of servers steady and increasing the number of clients on each server, there was a steady increase in the elapse time of message delivery, as it appears in Table I. That was an expected result, because server had to send the message to more clients, as I was increasing the number of them. However, I noticed that you can broadcast a message faster to the same number of clients, by adding one more server. For example, with 2 servers and 500 clients connected to each one (1000 clients in total), the message delivered faster than having 1 server with 1000 clients connected on it. The same improvement in time occurs with 2 server and 1000 clients (2000 clients in total) connected to each one, instead of 1 server with 2000 clients connected to it. The explanation behind this, is that the server's workload, to broadcast a message to a specific number of clients, is shared with another one server, so the performance is better [7]. However,

connecting too many servers to few number of clients, led to negative results. For instance, 2 servers with 1000 clients connected to each one (2000 clients in total) performed better that 4 servers with 500 clients connected to each one of them.

Research question #2: I made a testing run with 4 servers and 2000 clients connected on each one of them (8000 client in total) and my computer-server crashed. This occurred, because of memory lack of my computer-server. Thus, I concluded that hardware plays a major role for a server and determines the maximum number of connections that it can host [8].

REFERENCES

[1] Tanenbaum, A. and van Steen, M. (2016). *A brief introduction to distributed systems.* [online] Available at: https://link.springer.com/article/10.1007/s00607-016-0508-7 [Accessed 3 Dec. 2019].

[2] University of Indiana. (2018). *What is the difference between a LAN, a MAN, and a WAN?.* [online] Kb.iu.edu. Available at: https://kb.iu.edu/d/agki [Accessed 3 Dec. 2019].

[3] Techopedia.com. (2020). *What is a Distributed System? - Definition from Techopedia.* [online] Available at: https://www.techopedia.com/definition/18909/distributed-system [Accessed 4 Dec. 2019].

[4] Malhotra, A., Sharma, V., Gandhi, P. and Purohit, N. (2010). UDP based chat application. *2010 2nd International Conference on Computer Engineering and Technology*.

[5] Sabah, N., Kadhim, J. and Dhannoon, B. (2017). Developing an End-to-End Secure Chat Application. *IJCSNS Internation Journal of Computer Science and Network Security*, vol. 17, no. 11, pp.108-112.

[6] Mohamed, M., Muhammed, A. and Man, M. (2015). A Secure Chat Application Based on Pure Peer-to-Peer Architecture. *Journal of Computer Science*, vol. 11, no. 5, pp.723-729.

[7] IBM. (2019). *Load-balancing*. [online] Available at: https://www.ibm.com/cloud/learn/load-balancing [Accessed 22 Dec. 2019].

[8] Schroeder, B. and Harchol-Balter, M. (2002). *Web servers under overload: How scheduling can help*. [online] Available at: http://www.cs.toronto.edu/~bianca/papers/submosdi.pdf [Accessed 22 Dec. 2019].