|  |
| --- |
| MTU |
| Distributed Computing |
| Assessment |

|  |
| --- |
| Ivan Segade Carou T Number T00219357  24-03-2024 |

Contents

[Introduction 1](#_Toc162116947)

[Implementation 2](#_Toc162116948)

[Objectives and overview 2](#_Toc162116949)

[Demonstration of the use 3](#_Toc162116950)

[Application layer 6](#_Toc162116952)

[Presentation layer 8](#_Toc162116953)

[Server layer 11](#_Toc162116954)

[Documenting protocol 15](#_Toc162116955)

[Introduction + objectives 15](#_Toc162116956)

[Message definition 16](#_Toc162116957)

[Conclusion 19](#_Toc162116958)

[References 20](#_Toc162116959)

# Introduction

This document explains the development of the creation of a message system protocol between machines by using sockets in the Java programming language. The same machine is used by setting the IP address as localhost, but it may be used in a LAN by changing the localhost parameter for an IP address. The program uses sockets with the TCP protocol, which guarantees the reliability of the message being sent.

The code can be found in the following link:

<https://github.com/Segade/DistributedComputing>

The message system contains a server that centralises the sending of the messages. Then, clients connect to the server and can send messages to the server that stores them. The project accepts concurrent clients at the same time by using the thread technique. Moreover, the project implements SSL in order to encrypt the communication and as a result a secure connection.

The document explains how the project is designed internally by describing how messages are treated depending on the message name. Moreover, the document explains how the server handles concurrent clients and the SSL implementation. There are sections that describe the correct use of the program along with screenshots for a better understanding of the utilisation.

# Implementation

## Objectives and overview

The project consists of a message system between a server program and one or more client users. The server runs and is constantly listening in the 1981 port in the localhost machine waiting for receiving a request from a client at that port. When a client wants to connect to the server, must run the client program, which tries to connect with the server through the 1981 port. The server also logs all the actions occurred while this is running for supervision purposes.

The command required to run the client program is:

java -cp . -Djdk.tls.client.protocols=TLSv1.2 -Djavax.net.ssl.trustStore=public.jks -Djavax.net.ssl.trustStorePassword=password EchoClient2

The command to run the server program is:

java -Djdk.tls.server.protocols=TLSv1.2 EchoServer3

The server code must be run first to be listening to requests. A presentation message is displayed once the program starts. During the process, the server displays the text “Waiting for a connection”. Then the client code must be run and a presentation message is displayed. Then, users must enter their username first and their password later. If the credentials are correct, the server code allows users to access the system and a message is displayed on its screen with the text “Connection accepted”.

For this project, there are two usernames with their corresponding passwords:

Username: ivan Password: aaa

Username: peter Password: 1234

Once the connection is established and users are logged in, users have access the entire system and the main menu comes up on their screen. On this menu, users can send a message, retrieve all the sent messages, retrieve one particular message or log out.

In the case of entering wrong credentials, the server sends a message with the text “Wrong credentials” and the client adds the text “Wrong username or password. The server displays on its screen the text “fail/Wrong username or password”. Finally, the server denies the access of the user to the system. This automatically finishes the session and closes the connection and the client program ends while the server keeps listening on.

Apart from the main menu, there is a submenu in the “retrieve one message” option. On this submenu, users type the message number that they want to retrieve. Once the message is retrieved, users come back to the main menu. In the case of that the server cannot find the requested message, it sends an error message with the text “Not such message found”. Likewise, when users try to retrieve the complete list of messages and there is no messages in the list, the server sends an error message with the text “Empty list”

This project is developed based on an example of an Echo Java project created by Peter Given. The code belongs to the Echo3 example from one of his labs. The classes were refactored in order to adapt them to the needs of this project.

### Demonstration of the use

Following, there are some screenshots showing how the project works:

A screenshot of a computer

Description automatically generated

Figure 1. Example 1 of use of the client.

## A computer screen with a black screen Description automatically generated

Figure 2. Example 2 of the use of the client.

A computer screen with a black background

Description automatically generated

Figure 3. Example 3 of the use of the client.

A screenshot of a computer

Description automatically generated

Figure 4. Example 1 of the use of the server.

## Application layer

For creating this project, the programming language chosen is Java along with the Sockets technique and more specifically, stream sockets. Therefore, the project works on the TCP protocol to insure data cannot be lost.

The server can provide service for several concurrent users. For this purpose, the server uses threads. With this, each user can interchange messages with the server independently from the rest of the users connected.

In order to handle the concurrent server via threads, a class called EchoServerThread separated from the server class is created. This class implements the Runnable interface to be able to add it to the Thread class. Each time a new user accesses the system, the server class instantiates a new Thread and adds the runnable class to the Thread. The Thread class manages all the actions for each user independently. Therefore, user actions can not interfere among them.

This program implements the SSL technology to increment the security in the communication between the clients and the server. As the code which this program is based does not use SSL, the code was refactored in the client and server classes to achieve this. Moreover, OpenSSL was used to generate the public and private keys.

For this purpose, new libraires were imported in the EchoServer3 class, such as, java.security and javax.net.ssl. Moreover, new code was added to read the security key from the file and new classes were added, such as SSLContext, along with changing the regular server sockets to the SSL socket class, i.e. SSLServerSocketFactory and SSLServerSocket. Also when the MyStreamSocket class is instantiates a new object the socket passed as a parameter is casted to SSLSocket class.

\*\*\*\*\*\*\*\*\*\* SSL code

… rest of the code …

char ksPass[] = "secaiv".toCharArray();

char ctPass[] = "secaiv".toCharArray();

… rest of the code …

try {

KeyStore ks = KeyStore.getInstance("JKS");

ks.load(new FileInputStream(ksName), ksPass);

KeyManagerFactory kmf =

KeyManagerFactory.getInstance("SunX509");

kmf.init(ks, ctPass);

SSLContext sc = SSLContext.getInstance("TLS");

sc.init(kmf.getKeyManagers(), null, null);

SSLServerSocketFactory ssf = sc.getServerSocketFactory();

SSLServerSocket s

= (SSLServerSocket) ssf.createServerSocket(1981);

… rest of the code…

MyStreamSocket myDataSocket = new MyStreamSocket

( (SSLSocket)s.accept() );

… rest of the code …

\*\*\*\*\*\*\*\*\*\* End of new SSL code

Also the MyStreamSocket class was modified to adapt it to the SSL technology. A new library called javax.net.ssl was imported. Moreover, the regular Socket class was replaced by the SSLSocket class. Then, the constructor that is called by the client was modified adding the required classes to use the SSL technology. The constructor used by the server just receives a Socket object parameter. Therefore, this declaration was modified to SSLSocket to store it in the local SSLSocket object.

\*\*\*\*\*\*\*\*\*\* New SSL code

… rest of the code …

private SSLSocket socket;

… rest of the code …

MyStreamSocket(InetAddress acceptorHost,

int acceptorPort ) throws SocketException,

IOException{

SSLSocketFactory f =

(SSLSocketFactory) SSLSocketFactory.getDefault();

this.socket =

(SSLSocket) f.createSocket(acceptorHost, acceptorPort);

socket.startHandshake();

setStreams( );

}

… rest of the code …

\*\*\*\*\*\*\*\*\*\* End of new SSL code

## Presentation layer

The user presentation layer resides on separated file called EchoClient2 from the code that process the messages. In the case of the server side, the presentation layer resides on the two server files called EchoServer3 and EchoServerThread. The first one displays the presentation message and connection requests made. The second file displays the action made n by each user.

\*\*\*\*\*\*\*\*\*\* EchoServer3 code.

… rest of the code …

/\*\*/ System.out.println("Echo server ready.");

while (true) { // forever loop

// wait to accept a connection

/\*\*/ System.out.println("Waiting for a connection.");

MyStreamSocket myDataSocket = new MyStreamSocket

(myConnectionSocket.accept( ));

/\*\*/ System.out.println("connection accepted");

… rest of the code…

\*\*\*\*\*\*\*\*\*\* End of EchoServer3 code

\*\*\*\*\*\*\*\*\*\* EchoServerThread code.

switch(header){

case "login/":

if (!findUser(message.trim()) ){

System.out.println("fail/Wrong username or password");

myDataSocket.sendMessage("fail/Wrong credentials");

myDataSocket.close( );

done = true;

} else {

username = Functionality.getUsername(message);

System.out.println("The user : " + username + " is logged in");

myDataSocket.sendMessage("loggedin/You are logged in");

}

break;

case "message/" :

message = Functionality.getMessage(message);

allMessages.add(message);

System.out.println("Message received from " + username + "\n" + message);

myDataSocket.sendMessage("received/Message " + allMessages.size() + " sent" );

break;

case "logout/":

System.out.println("The user " + username + " is logged out");

myDataSocket.close();

done = true;

break;

case "retrieveall/":

String messages = sendAllMessages();

System.out.println("The list of the messages of" + username + " is: " + messages);

myDataSocket.sendMessage( messages);

break;

case "retrieveone/" :

int position = Integer.parseInt(Functionality.getMessage(message));

if (position >= allMessages.size() )

message = "Not such message found";

else

message = allMessages.get(position);

myDataSocket.sendMessage("onemessage/" + message);

System.out.println("The user " + username + " requested the message:\n" + position + ". " + message );

} // end switch

\*\*\*\*\*\*\*\*\*\* End of EchoServerThread code.

This project is designed in a console environment Whether the server code and the client code must be run on a command line. Once the server is running displays all the action triggered by all different users.

Once the client code is executed, a prompt asks for an username followed by a second prompt that askes for a password. The menu is used by entering the number of the chosen option, likewise the submenu for retrieving one message.

## Server layer

This layer contains two classes: EchoServer3.java and EchoServerThread.java.

\*\*\*\*\*\*\*\*\*\* EchoServer3 code.

import java.io.\*;

import java.net.\*;

/\*\*

\* This module contains the application logic of an echo server

\* which uses a stream-mode socket for interprocess communication.

\* Unlike EchoServer2, this server services clients concurrently.

\* A command-line argument is required to specify the server port.

\* @author M. L. Liu

\*/

public class EchoServer3 {

public static void main(String[] args) {

int serverPort = 7; // default port

String message;

if (args.length == 1 )

serverPort = Integer.parseInt(args[0]);

try {

// instantiates a stream socket for accepting

// connections

ServerSocket myConnectionSocket =

new ServerSocket(serverPort);

/\*\*/ System.out.println("Echo server ready.");

while (true) { // forever loop

// wait to accept a connection

/\*\*/ System.out.println("Waiting for a connection.");

MyStreamSocket myDataSocket = new MyStreamSocket

(myConnectionSocket.accept( ));

/\*\*/ System.out.println("connection accepted");

// Start a thread to handle this client's sesson

Thread theThread =

new Thread(new EchoServerThread(myDataSocket));

theThread.start();

// and go on to the next client

} //end while forever

} // end try

catch (Exception ex) {

ex.printStackTrace( );

} // end catch

} //end main

} // end class

\*\*\*\*\*\*\*\*\*\* End of EchoServer3 code.

\*\*\*\*\*\*\*\*\*\* EchoServerThread code.

import java.io.\*;

/\*\*

\* This module is to be used with a concurrent Echo server.

\* Its run method carries out the logic of a client session.

\* @author M. L. Liu

\*/

class EchoServerThread implements Runnable {

static final String endMessage = ".";

MyStreamSocket myDataSocket;

EchoServerThread(MyStreamSocket myDataSocket) {

this.myDataSocket = myDataSocket;

}

public void run( ) {

boolean done = false;

String message;

try {

while (!done) {

message = myDataSocket.receiveMessage( );

/\*\*/ System.out.println("message received: "+ message);

if ((message.trim()).equals (endMessage)){

//Session over; close the data socket.

/\*\*/ System.out.println("Session over.");

myDataSocket.close( );

done = true;

} //end if

else {

// Now send the echo to the requestor

myDataSocket.sendMessage(message);

} //end else

} //end while !done

}// end try

catch (Exception ex) {

System.out.println("Exception caught in thread: " + ex);

} // end catch

} //end run

} //end class

\*\*\*\*\*\*\*\*\*\* End of EchoServerThread code.

The server layer holds two files. The first one is the main one that is in charge of processing all the logic. This file implements the server Socket code and is constantly listening on. Also, the server manages the creation of the threads that hold the session for each user.

The second file manages user accounts and user actions. It stores the list of registered usernames along with their corresponding password in a String array for authentication purposes. This file processes the requests made by the corresponding user and sends the answer with the required information.

# Documenting protocol

## Introduction + objectives

The application consists of a message program, where users can log in to the system through a valid user name and password. Then, users have access to the menu where they can select between: send a message, retrieve all the messages, retrieve one message and log out.

Users send their messages to the server that stores the messages and communicates with the client to confirm if it received the messages correctly. Also, users can retrieve one particular message from all the messages sent previously. For this purpose, users must pick this option from the menu and then enter the number of the required message. Moreover, users can retrieve all the messages sent previously. For this purpose, users must pick this option form the menu and the server will send all user messages.

When users want to finish their session, they must pick the log out option. This option will finish the session and closes the connection with the server.

## Message definition

Message: Login.

Description: The user can log into the server with a username and password.

Header: login/

Message Parameters: username, password (both text). With the format “username:<username>&password:<password>”

Response Message

Header: loggedin/

Text: You are logged in

Response Message

Header: fail/

Text: Wrong credentials

Message: Send a message.

Description: The user can send a message to the server, which stores the message in the user session.

Header: message/

Message Parameters: The message of the text (text)

Response Message

Header: received/

Text: Message <message number> sent

Message: Retrieve one message.

Description: The user can retrieve one specific message picked from the list of previous sent messages during the current session by the message number.

Header: retrieveone/

Message Parameters: Message number (Number)

Response Message

Header: onemessage/

Text: <Message text>

Response Message

Header: onemessage/

Text: No such message found

Message: Retrieve all the messages

Description: The user can retrieve all the messages sent during the current session.

Header: retrieveall/

Message Parameters: <empty message>

Response Message

Header: amessage/

Header: lastmessage/ (For the last message of the message list)

Text: <Message number>. <Message text>

Response Message

Header: lastmessage/ (No messages)

Text: Empty list

Message: Log out

Description: The user can log out of the current session and close the connection with the server

Header: logout/

Message Parameters: <empty message>

Response Message

<No response message>

Describe how functionality of each message is implemented

User name and password are sent in the same message as one string. “usrname:<username>&password:<password>”. The credentials are stored on the server with the same format in a string array. A find User function compares the string sent by the user with all the strings in the array looking for a match.

If the user credentials are valid, he receives the message “loggedin/You are logged in”. In the case of the credentials are not valid, the user receives the message “fail/Wrong credentials” and the program ends automatically.

A message is sent once the user is logged in. The user choose the option from the menu “Press enter to send a message) and sends the message with the format “message/<Message text>” to the server that stores it in an Array List. Once the message is sent, the user receives a confirmation message with the format “received/Message <Message number> sent”.

The retrieve all messages is sent in order to see all the messages sent during the current session. The user chooses the option from the menu (Press 1 to retrieve all the messages). Then, the user sends a message with the format “retrieveall/. As a result, the server retrieves all the messages stored in the ArrayList for that user and sends all back to the user with the format “amessage/<Message number>. <Message text>”. There is an exception for the last message. To differentiate this message and say to the user that that is the end of the list, the format of the message is “lastmessage/<Message number>. <Message text>”. In the case of there is not messages in the ArrayList the message sent back to the user has the format “lastmessage/Empty list”.

The retrieve one message is sent in order to see a message picked for the user by the message number. The user chooses the option from the menu (Press 2 to retrieve one message). Then, a second menu comes up asking for the message number to be retrieved. The user enters the selected number and the server will send back the required message with the format “one message/<Message text>”. In the case the number does not belong to any message, the user will receive a message with the format “onemessage/No such message found”.

The log out message is sent in order to allow the user to end the session and close the connection with the server. The user chooses the option from the menu (Press 3 to log out). Then, the user sends a message with the format “logout/”. Once the server receives this message, it closes the connection with the user. Finally, the user see the message “You are logged out” and the program ends.

# Conclusion

This project uses Sockets to set up a connection between a server program and one or more client program in order to create a message storage system. Sockets facilitates the task of setting up connection between two end points by avoiding to code in a low level. Sockets are cross-platform, this means that Sockets can create a connection regardless the operative system or device type, since the most known programming languages provide Sockets and they are compatible among them.

Socket perform at a low level, therefore, the communication is done in an efficient way. Moreover, Sockets are suitable for real time communications that need instant transfer of information, such as, chats, messengers or even video games. Also, Sockets are used for data transfer or updates, where a constant connection is required (PubNub, 2024) , although Sockets can establish non-blocking communication as well.

Sockets by themselves do not encrypt the data transmitted. Therefore, additional implementations, such as SSH (Loshin & Cobb, 2024)must be added to provide confidentiality. The connection is establish between only two end points, therefore, the connection is 1-1. Even though, the server can handle several connections, but these can not interact among them (GeeksforGeeks, 2024). In concurrent connections Sockets are not efficient, since server has to maintain the communication between the server and the client and that consumes resources. This fact reduces the scalability. Moreover, Sockets perform through ports, this can create generate conflicts with firewalls (PubNub, 2024).

Alternatives to Sockets may be solutions related to Remote Procedure Call (RPC). Different solutions are GRPC, THRIFT or SOAP. These perform at higher level than can define more variety of methods or services. This abstracts developers from working at a low level and a result these can focus on the application level. With Sockets, developers must define the conventions and error handling in the message system, unlike RPCs solutions (IBM, 2024).

Another option is Rest. Rest performs through the http protocol. This protocol is stateless. This means that there is no a direct connection between the server and client, therefore the messages must contain all the required information of the server and client, as the server does not store any information of the connection with the client. This is more scalable, since sever is not dedicated to specific connections. On the other side, developers may need to maintain the connection state between the machines, therefore, additional code must be implemented. Rest uses JSON or XML formats, therefore, developers do not have to define their our conventions when sending messages (ably, 2024).

As a conclusion, if developers want to create a program with its own protocol definition, with a big control of the sending system and how messages must be interpretated and manipulated depending on different situations, Sockets is a good solution. However, this program must not handle a large number of connections, since Sockets are stateful. This means that server maintains the information of each connection with each end point and this requires a large consume of resources. If this is not the case, developers have other solutions that may meet their needs.

# References

ably. (2024, 03 10). *WebSocket vs REST: Key differences and which to use*. Retrieved from ably.com: https://ably.com/topic/websocket-vs-rest

GeeksforGeeks. (2024, 03 10). *Advantages and Disadvantages of Java Sockets - GeeksforGeeks*. Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/advantages-and-disadvantages-of-java-sockets/

IBM. (2024, 03 10). *Remote Procedure Call - IBM Documentation*. Retrieved from ibm.com: https://www.ibm.com/docs/en/aix/7.1?topic=concepts-remote-procedure-call

Loshin, P., & Cobb, M. (2024, 03 10). *What is SSH (Secure Shell) and How Does it Work? Definition from TechTarget.* Retrieved from techtarget.com: https://www.techtarget.com/searchsecurity/definition/Secure-Shell

PubNub. (2024, 03 10). *What are WebSockets? | PubNub*. Retrieved from pubnub.com: https://www.pubnub.com/guides/websockets/