Application Layer Network Protocol

University of cape town |

CSC3002F – Assignment 1

Theo Madikgetla (MDKTHE015), Rethabile Modise (MDSRET001), Saadiq GEDULDT (GDLMOE001)

2021

Table of Contents

[ Abstract: 2](#_Toc69196045)

[1. Introduction: 2](#_Toc69196046)

[2. Using UDP: 2](#_Toc69196047)

[3. Protocol Design: 3](#_Toc69196048)

[4. Implementation: 5](#_Toc69196049)

[o Features 5](#_Toc69196050)

[ Client Class: 5](#_Toc69196051)

[ Additional features: 6](#_Toc69196052)

[5. Screenshots: 6](#_Toc69196053)

[6. Conclusion: 6](#_Toc69196054)

[Reference List: 7](#_Toc69196055)

# Abstract:

Client-server Architecture is a distributed communication framework of networks and it denotes a relationship between associative programs in an application that consists of clients requesting services and servers providing that service to them. In this Assignment, we are designing a client-server chat application using User Datagram Protocol (UDP) at the transport layer.

# Introduction:

Client-server Architecture is a distributed communication framework of networks and it denotes a relationship between associative programs in an application that consists of clients requesting services and servers providing that service to them. In this Assignment, we are designing a client-server chat application using User Datagram Protocol (UDP) at the transport layer.

# Using UDP:

UDP is a lightweight transport protocol. Due to being connectionless, they are used for establishing low-latency environments for sending information between applications over the internet. Data can be transferred much faster as no agreement is given by the receiving application as in TCP. Therefore, UDP would be more beneficial to use in a simple chat application. TCP would be better for sending large amounts of data, but the application only requires sending text data in small amounts and thus the faster speed of UDP would be preferred. This does come with the cost of unreliability as there is no guarantee that data will be delivered, and data could also be received in a completely different order to what they were sent in. To counteract this, threads will be used to parse through incoming data accordingly and maintain an order of the data sent and received.

* Server Configuration

The server will be responsible for handling all requests from clients and this will also include passing on the text data being sent from one client to another. It will first have to successfully connect to a specific port. Thereafter it will constantly be receiving and appropriately handling requests from multiple clients. To help with multiple requests being received at any given point, the server will make use of a helper thread that starts whenever a request is received. This thread will receive the data from any one client, process it and send the appropriate response to the client and after it has completed this process, it is safely killed.

* Client

Like the server, the client will have to successfully connect to a socket. The client will behave similarly to the server with regards to handling incoming messages, with the only exception being that a user will be sending requests or responses as well. The client will make use of a helper thread as well to help parse through user input that needs to be sent out.

# Protocol Design:

Both the Server and Client will make use of the same *DatagramSocket* methods for sending and receiving data:

1. **DatagramSocket.receive**()

* E.g.:
* *DatagramPacket packet = new DatagramPacket(received,received.length);*
* *socket.receive(packet);*
* This will accept any incoming packets and will therefore need to be checked.

1. **DatagramSocket.send**()

* E.g:
* *String message = text;*
* *byte[] sendData = new byte[1024];*
* *sendData = message.getBytes();*
* *DatagramPacket send = new DatagramPacket(sendData, sendData.length, serverIP,serverPort);*
* *socket.send(send);*
* This will send any inputted text from the user or any response from the server

The following two functions form the basis of our protocol.

* **Messages:**

Messages will all have a standard format being [source] <message>. This will indicate to both client and server where the message is coming from. In the case of the clients, it will indicate if the message has been sent by the server or if it is chat text from another client. As for the server it will indicate what client has sent a request to the server.

* **Commands:**

Clients are given a list of command options of which they can type in and send. These commands will be typed alongside text in the chat, so they will need an indication that they are commands. We decided to use “//” as a prefix to indicate that whatever follows is a command statement. The client commands are:

* //*contact*: requests a list of online clients connected to the server at the time of the request
* //*exit*: safely exits the client program, as well as indicate to server that they have disconnected
* //*startchat <name>:* requests to start a chat with another client. The server will indicate to both clients that they are in a chat together
* //*end*: ends a chat with another client and indicates to them that the chat has been terminated
* **Clients**

After successfully connecting to a port, will be required to provide a name that will be used to be registered on the server. Thereafter, the client will continuously be waiting for user input in a send thread which will parse through inputs and prevent sending incorrect commands to the server. The client class will act as a main receiving thread that parses through incoming messages and prints them out accordingly.

* **Server**

The server class makes use of server threads. Whenever a packet is received the details of that packet are used to instantiate a new server thread. The thread saves the information of the client that sent the packet and check what information is in the packet data and responds accordingly. The thread will then be stopped. This will allow multiple users to send to the server socket and make sure responses are sent to the correct client that made the request.

Sequence Diagram

Diagram

Description automatically generated

# Implementation:

# Features

## Client Class:

* *addToLogs*:

This function receives a string input as an argument from the client. It stores the messages of the client while interacting with other clients to the server, and the date they were added. This method allows the client to be able to retrieve messages in the future for reference or to ensure that she received all the messages.

* *Start*:

This function receives no arguments but is called in the constructor. it includes the *DatagramSocket* which implements a connectionless data communication by creating a socket that is bound to port number 1235. The *DatagramPacket* class is then instantiated to facilitate the transfer of messages from one client to another.

This function also allows multiple users to send messages to another instead of just two clients by creating a separate thread for each client that is connecting.

* *Stop*:

This function checks if the *DatagramSocket* is closed. If it is not, it closes it.

* *Listener class*:

This class implements the runnable interface to allow multiple messages to be added into the database at the same time.

* *Server class*:

The server creates a *ClientHandler* object. This object is responsible for communicating with a particular class.

* *ClientHandler class*:

This class represents the Server in the client- server framework. for each client connected to the server and it implements runnable interface to allow multiple clients to be connected to the server.

Features include:

* **Run**

It first checks the arguments provided by the client to connect. If the name of the client is already connected, it informs the client that the username is invalid.

Else the client connects successfully.

Because of the threading, this class also includes a feature for creating a group chat. One several clients are connected to the server they can all communicate. This is done by creating an object of the *DatagramSocket* which implements the wireless connection and the *DatagramPacket* which receives multiple arguments to facilitate the connection.

## Additional features:

1. N/A

# Screenshots:

# Conclusion:

User Datagram Protocol and Transmission Control Protocol are two Protocols used by the transport layer to allow applications to exchange messages. There are many differences that distinguish them but the main one is that TCP is connection orientated while UDP is not. In this Chat application, we utilized UDP socket. It is faster than TCP to send messages because there are no attempts of error recovery and we do not need to track the connections or order messages.

# Reference List: