User Datagram Protocol (UDP) Messenger Application

[GitHub ReadMe]

Zongo Maqutu – MQTZON001

April 13, 2021

Introduction

This report will serve to give insight on the inner workings of a chat application that uses UDP at the transport layer. The application enables a client user to send a text message to another client 'connected' to the same server. The User Datagram Protocol is an inherently unreliable protocol, this report therefore delves into various features that were implemented at the application layer to mitigate UDP's unreliability. Sequence diagrams will be provided to show the workflow of the entire application.

Application Architecture

OOP and Class Description

In order to effectively talk about the network related aspects of the application it is important to first understand the file structure and the overall Object-Oriented Programing Design of the messenger application.

Client OOP Design:

The client has two files named:

`clientDriver.java` and `Client.java`

This driver file creates a new instance of a client class which extends the JFrame class as shown in the code snippet below.

`Client udpClient = new Client("localhost");` and `public class Client extends JFrame{`

The class constructor creates a graphical user interface (GUI) that the user can then engage with to chat to another user or with multiple users in a group chat. A basic layout of the client GUI is shown later in the report.

Once the user enters their username and that of the individual they wish to communicate with, a "startRunning()" method is called. This method creates and runs a single messegeHandler thread whose job is to receive incoming messages. The messageHandler thread is created like this: `new Thread(new messageHandler()).start();` . messageHandler is a Runnable subclass of Client that decodes information about messages as they come in and runs the necessary functions to process the message.

Server OOP Design:

The server's structure is very similar to that of the client. The main difference is in the tasks both programs perform. Like with the client, the server has files named:

`serverDriver.java` and `Server.java`

This also creates a new instance of a Server class in the manner shown below:

`Server udpServer = new Server();` and `public class Server extends JFrame{`

A GUI for the server will be created where the server administrator can engage with the graphical user interface to start and stop the server. Once more, when the start button is pressed the server runs a method named startRunning(). This method creates multiple instances of a subclass called clientHandler. These clientHandlers implement Runnable and create a new thread each time a new user requests to connect to a server. This means that the number of threads created should equal the number of clients using the application. The clientHandler thread is created like this: `new Thread(new clientHandler(username,recipientName,ip,portNo)).start();`

Each thread class is responsible for receiving messages from a client and redirecting them to the intended recipient.

A more detailed description of the network architecture will be provided in the next section.

Protocol Specification

In this section a detailed description of the application layer protocol specification is provided. Furthermore, a sequence diagram that will detail the transmission of packets in and around the system is provided.

Message format and structure:

Each message is sent and received as a byte array which is then converted to a string for processing. For example, for a typical chat message received, the string message may have the following format:

```
message + "\n" + sender + "\n" + recipient + "\n" + checksum;
```

The split("\n") method is then called on the string and each element in the resulting array is assigned a variable name for further processing. The server can determine the type of message received based on the length of the array that is returned when we split the data by a newline character ("\n").

Message types and their array lengths:

Length of array	Signature	meaning
2	message + "\n" + "checksum"	The incoming data is a old chat retrieval and will be printed to client screen
3	name + "\n" + recipient + "\n" + localPort	This is a login request
4	message + "\n" + sender + "\n" + recipient + "\n" + checksum	This is a data transfer message where a message is sent to recipient with information of who sent it

Messages of array length 2 comprise of a message and a checksum only. This message structure is used primarily retrieving old chat messages. After a login request has been made the server

looks through its chat logs for any previously existing chats and sends them exactly as they are to the requesting client using this format. Each message is checked for validity using the checksum field. When the message has been checked it is then displayed in the GUI's chat window. Information like IP address and Port Number is not necessary since we only need to receive the message and write it to screen. An example of chat retrieval using this message format can be seen in the image below.

Messages of array length 3 are categorized as login requests to the server. This packet is often the very first packet sent by the client with a structure like this:

name + "\n" + recipient + "\n" + localPort

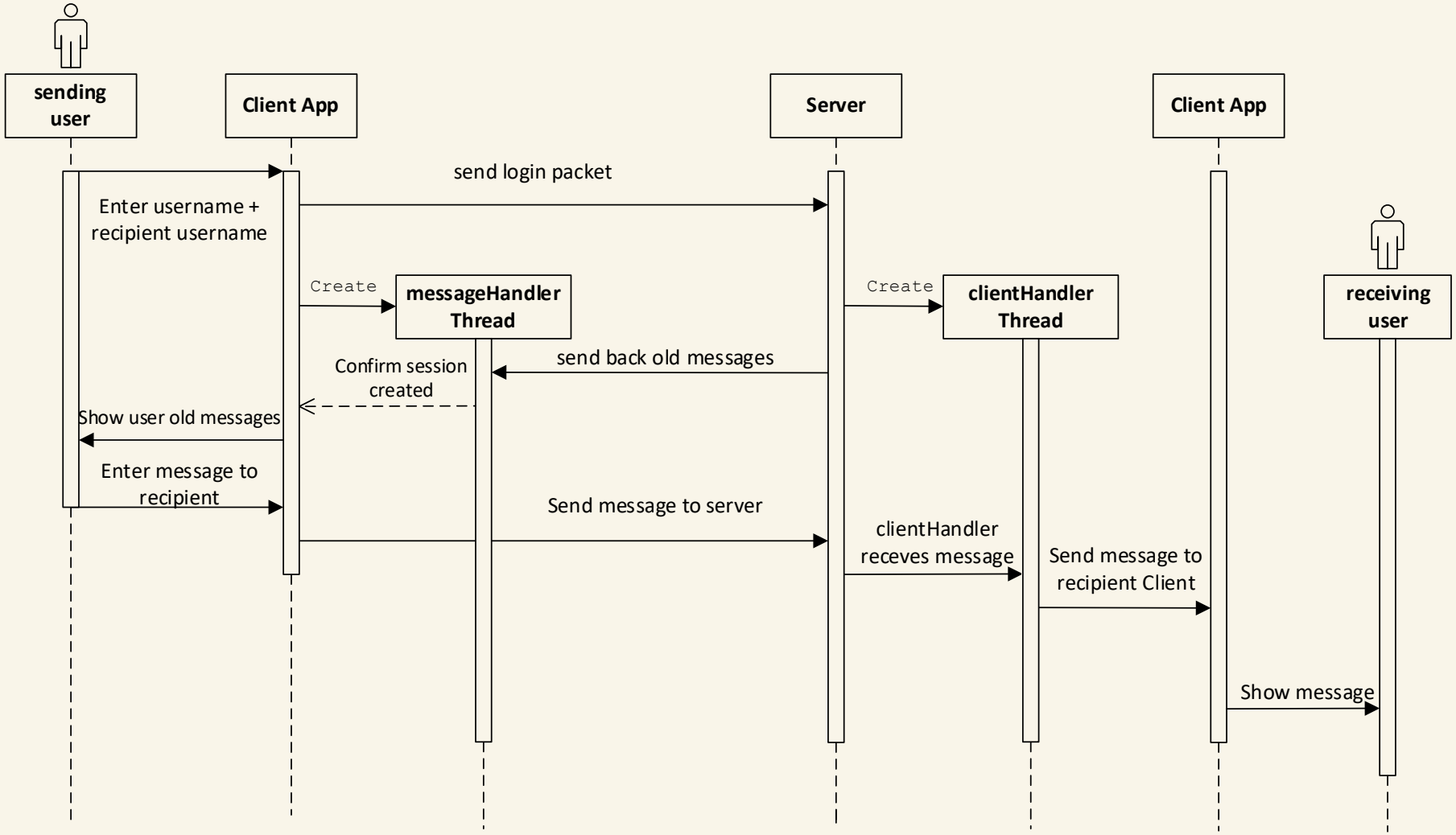
The client sending a request to create a session attaches a randomly assigned local port that will be connected to our sockets.

Messages of array length 4 are simply your data transfer messages that facilitate the chatting. They hold the message, the names of the people in the conversation and the checksum to validate that the message is correct.

Sequence Diagrams message journey description:

The sequence diagram on the next page describes, step by step, how a login session is initiated by the sending client and a message is sent to a receiving client. The sender client first enters their username and the username of the user they wish to chat with. A login packet is then sent to the server where the packet data will initiate the process of retrieving messages if they exist. A messegeHandler thread is created in the client and listens for incoming packets. The server reads from a file and sends the chat logs directly to the sender so they can show their old messages on a chat window. At this point the user can now enter the message they wish to send in the GUI's text field and send it back to the server. The message is received by the server's clientHandler and sent off to the receiving client. The sequence is the same for all clients connected to the server.

UDP Messenger App Sequence Diagram:



Sequence Diagram for a client connection to the server and sending a message to a recipient

Features

There are several features that have been implemented to mitigate UPD's unreliability. Other features were implemented for convenience and a more favorable user experience. This section will detail each feature's implementation and the justification behind adding that feature.

Message Retrieval:

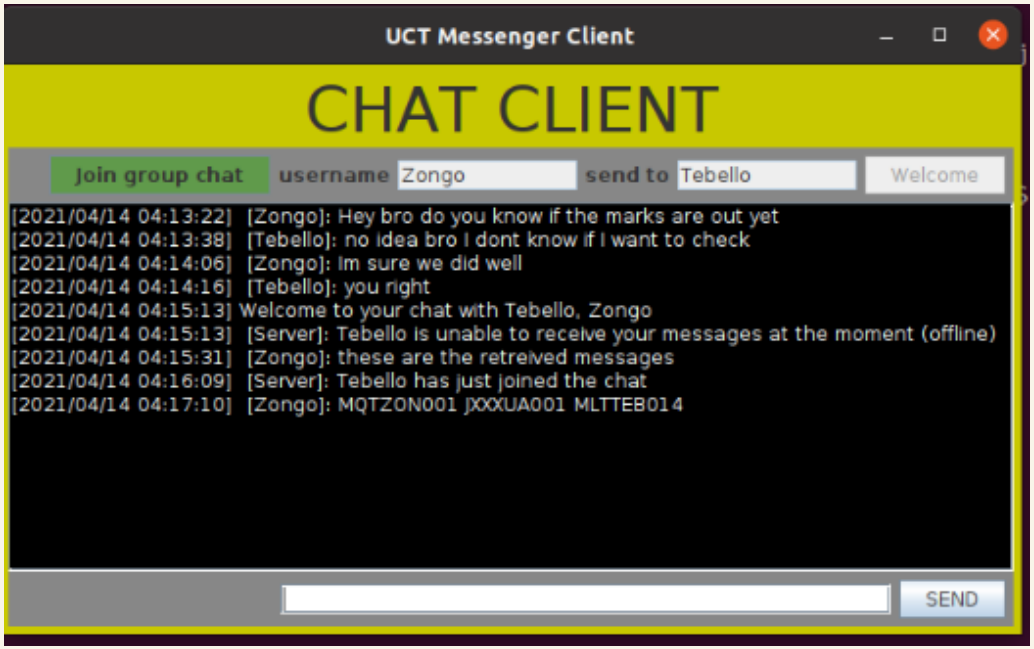
The ability for a user to recover old messages even after a new session has been created is very important to increasing our application's reliability. UDP does not guarantee that a message will be received, this is especially true when a connection with the recipient cannot be established. In our design, every message that the server receives is appended to a file just before the packet is sent to its intended recipient. Every user who logs in for a chat has a directory with their name created. This directory contains txt files of all conversations the user has had. When a user logs in and specifies who they wish to communicate with, the server goes into the directory with the format

chatlogs/username/recipient.txt

and sends back all the contents of that file to the client. It is important to note that messages are still appended to this text file even if the recipient is not available to receive the message. This means that will be able to receive those messages once they get back online.

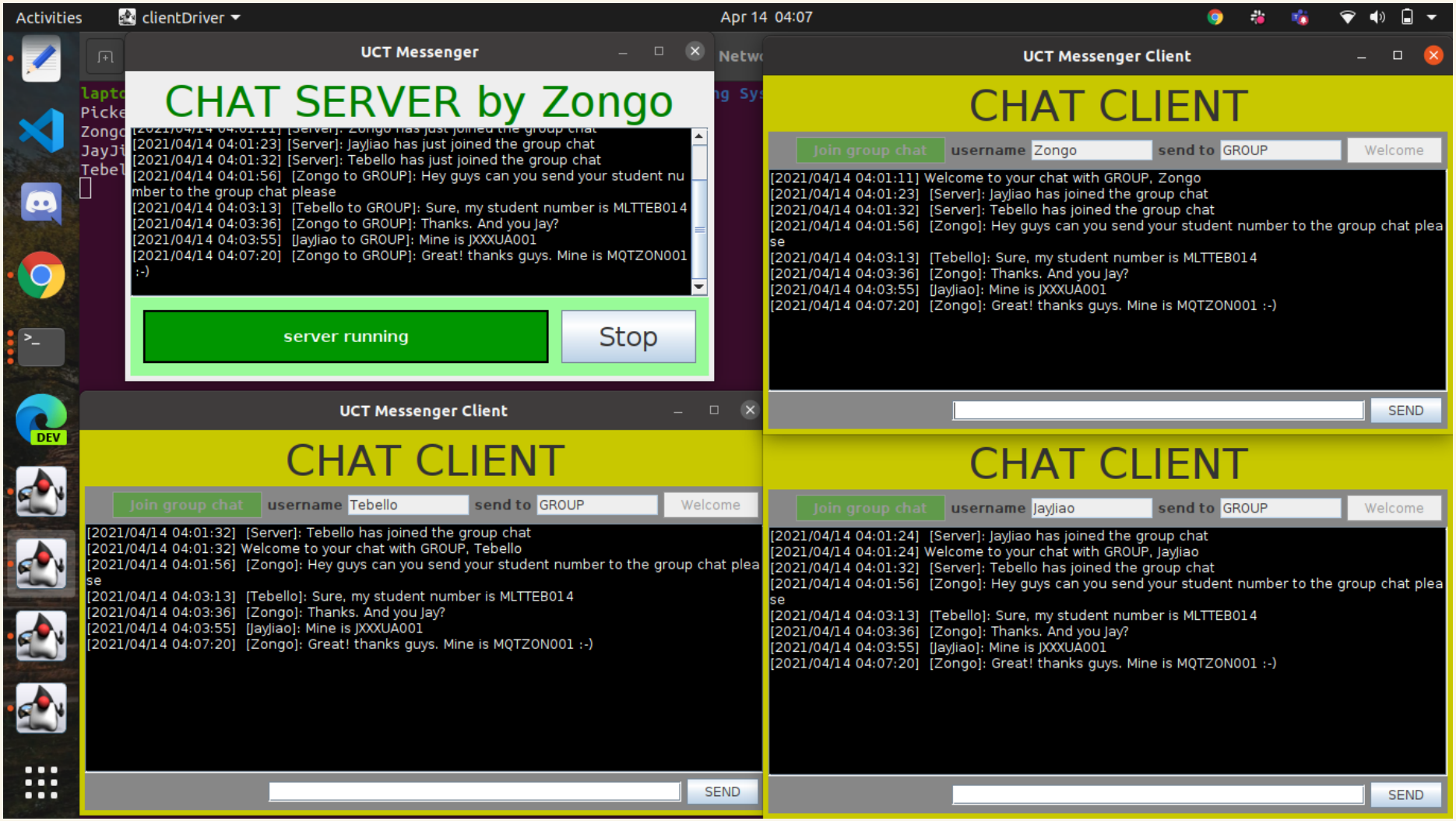
Adding a feature that enables users to receive messages even if they are not available mitigates much of UDP's unreliability problem. It means that users can always retrieve the true copy of messages sent to them.

In the screenshot below, the user Zongo tries to chat with Tebello. Their previous messages are retrieved and a notification of Tebello's current online status is displayed.



Group chat:
The group chat feature allows each client to broadcast their message to other clients in a group chat. When a user presses the 'join group chat' button we set the recipientName value in our packet to 'GR\0UP'. The server then adds this user to a HashMap of group chat users. When the server receives a message with a recipient field labeled 'GR\0UP' the server knows not to send it to a singe user but to rather call the broadcastMessage() method, which sends the message to all users in our group chat HashMap.

The screenshot below shows a group chat between three users. Each client uses their first name as their user name.


Activities
clientDriver
Apr 14 04:07
UCT Messenger
CHAT SERVER by Zongo
[2021/04/14 04:01:23] [Server]: Jayjiao has just joined the group chat
[2021/04/14 04:01:32] [Server]: Tebello has just joined the group chat
[2021/04/14 04:01:56] [Zongo to GROUP]: Hey guys can you send your student number to the group chat please
[2021/04/14 04:03:13] [Tebello to GROUP]: Sure, my student number is MLTTEB014
[2021/04/14 04:03:36] [Zongo to GROUP]: Thanks. And you Jay?
[2021/04/14 04:03:55] [JayJiao to GROUP]: Mine is JXXXUA001
[2021/04/14 04:07:20] [Zongo to GROUP]: Great! thanks guys. Mine is MQTZON001 :-)
server running
Stop
UCT Messenger Client
CHAT CLIENT
Join group chat
username Zongo
send to GROUP
Welcome
[2021/04/14 04:01:11] Welcome to your chat with GROUP, Zongo
[2021/04/14 04:01:23] [Server]: Jayjiao has joined the group chat
[2021/04/14 04:01:32] [Server]: Tebello has joined the group chat
[2021/04/14 04:01:56] [Zongo]: Hey guys can you send your student number to the group chat please
[2021/04/14 04:03:13] [Tebello]: Sure, my student number is MLTTEB014
[2021/04/14 04:03:36] [Zongo]: Thanks. And you Jay?
[2021/04/14 04:03:55] [JayJiao]: Mine is JXXXUA001
[2021/04/14 04:07:20] [Zongo]: Great! thanks guys. Mine is MQTZON001 :-)
SEND
UCT Messenger Client
CHAT CLIENT
Join group chat
username Tebello
send to GROUP
Welcome
[2021/04/14 04:01:32] [Server]: Tebello has joined the group chat
[2021/04/14 04:01:32] Welcome to your chat with GROUP, Tebello
[2021/04/14 04:01:56] [Zongo]: Hey guys can you send your student number to the group chat please
[2021/04/14 04:03:13] [Tebello]: Sure, my student number is MLTTEB014
[2021/04/14 04:03:36] [Zongo]: Thanks. And you Jay?
[2021/04/14 04:03:55] [JayJiao]: Mine is JXXXUA001
[2021/04/14 04:07:20] [Zongo]: Great! thanks guys. Mine is MQTZON001 :-)
SEND
CHAT CLIENT
Join group chat
username JayJiao
send to GROUP
Welcome
[2021/04/14 04:01:24] [Server]: Jayjiao has joined the group chat
[2021/04/14 04:01:24] Welcome to your chat with GROUP, Jayjiao
[2021/04/14 04:01:32] [Server]: Tebello has joined the group chat
[2021/04/14 04:01:56] [Zongo]: Hey guys can you send your student number to the group chat please
[2021/04/14 04:03:13] [Tebello]: Sure, my student number is MLTTEB014
[2021/04/14 04:03:36] [Zongo]: Thanks. And you Jay?
[2021/04/14 04:03:55] [JayJiao]: Mine is JXXXUA001
[2021/04/14 04:07:20] [Zongo]: Great! thanks guys. Mine is MQTZON001 :-)
SEND

Message check sum:
Each message that is sent has a checksum added as the last item to arrive. The checksum is used to verify that the message was not corrupted during transmission and is used to mitigate/prevent packet loss. Messages have the format

message + "\n" + sender + "\n" + recipient + "\n" + checksum

When a client or server receives a packet it takes the message, send, and recipient fields and passes them through a checksum function. The return value of this checksum is then compared with the checksum value in the packet. A new checksum is generated when a message is sent. This

Confirmation message has been received:
The client-server application was built to allow users to know if their messages have been received or not. From the very beginning, the server will notify a client user if the person they are sending message to is offline or not. The application will also notify them once that person is online to alert them that their messages are being received in real time. Furthermore, when a message is sent by a client, it is not shown on the screen once we receive an identical message from the server. This acts as an acknowledgement that the message has been received by the server which writes the message to a text file, guaranteeing the message will be received by another client. This feature vastly increases our application's reliability.

Additional features:
Additional features include a GUI for both client and server administrators.

Conclusion

There are several conclusions one might draw from implementing this application. It seems like there are many tradeoffs to consider when selecting a transport layer protocol. UDP is faster since time is not wasted establishing a connection. This makes it suitable for instances where you just want to send a packet and do not care about creating/establishing a connection. We had to implement many features on the application layer to overcome UPD's shortcomings. It may be worth considering another protocol for an app like this. TCP seems as though it would be perfect considering it is far more reliable.