



COMPUTER NETWORKS

UDP-USER DATAGRAM PROTOCOL

| Theo Madikgetla – MDKTHE015 |
| Cameron Broomfield – BRMCAM003 |
| CSC3002F | 10/03/2022

DECLARATION

We declare that this project report **Computer Networks: UDP- User Datagram Protocol** is based on our original work that was conducted during the projects given timeline. We further certify that:

1. Our work has not been shared with anyone who was not part of the group as we understand that it risks being plagiarized
2. Where we used materials (data, theoretical analysis and text) from other sources, we have given credit to the, in the text of the report and given the details in the reference list.

ABSTRACT

This project report outlines a networks client server application that is developed using Python's socket library. The objective is to design an application layer protocol and use it to develop a chat application that uses one of the Transport Layer's protocol User Datagram Protocol (UDP), which is known to be unreliable when exchanging data between hosts and make it a reliable protocol like the Transmission Control Protocol (TCP).

The chat application will have the group chat feature and an option of a one-to-one private chat. It is developed using the client-server architecture.

Table of Contents

1. Introduction	2
2. Systems Architecture	2
3. System Functionality	2
• Description	2
• Features.....	2
4. Specification	3
• gENERAL USAGE INSTRUCTIONS.....	3
• Protocol Design	4
➤ Sockets:.....	4
➤ Packet/message format/structure	4
• UML Diagrams.....	6
➤ Use case diagram.....	6
5. Result & Discussion.....	7
6. Conclusion.....	9

1. Introduction

The chat application is developed to support the client-server architecture in that there will be a server that will be responsible for managing communication between two or more clients that are connected to it. It is responsible for authenticating users, meaning a random client is not able to start sending data without first being registered. The application layer protocol is designed to use the User Datagram Protocol, UDP, and so data/messages that are sent, are encapsulated, and sent as packets that use the UDP packet format/structure.

The client interface is a Command Line Interface, CLI, and the user enters commands to connect to a server and interact with other clients. The protocol is designed in such a way that it achieves reliability. It will be able to tell when a packet is successfully sent and received and be able to detect errors from faulty packets that are received. If for some reason the packets get lost, the packets will be retransmitted by the sender.

2. Systems Architecture

- *Client-server model:*

The client-server architecture is a computer network architecture where there is a centralized host, which is the server, and many clients can request and receive services. The server is an always-on host, actively listening for requests and provides the requested services. The clients do not interact directly with each other.

3. System Functionality

- Description

The chat application allows multiple pairs/groups of users to communicate over a network. They can only send texts to each other using the CLI.

- FEATURES

1. User Authentication

- No two clients can have the same user identification and if one when registers the server finds a client with a similar user identification, it asks the client to enter a different username.
- 2. Automatic retransmission
 - A message that hasn't received an acknowledgement is automatically retransmitted after 5 seconds.
- 3. Chat rooms
 - The ability for more than one pair/group to communicate in separate chat rooms on the same server. This allows for compartmentalization of messages. Only members of the same chat room can communicate with each other.
- 4. Group Chat
 - The ability for more than just two users to communicate at the same time.

4. Specification

- GENERAL USAGE INSTRUCTIONS
 - First the **serverApp** is started.
 - When the **clientApp** is started, it requests the user's name. It then automatically establishes a connection with the **serverApp** through the ***/knock knock*** command, ensuring that the name is unique. If the name is not unique, then the client will be asked repeatedly until a unique name is chosen.
 - After the connection is established, the client is welcomed to the waiting room with the message: **"Welcome, " + name + ", to the waiting room. "Type /list, /create [room name] or /join [room name]"**.
 - From here, the client may use the following three commands (commands are denoted by the first character being a forward slash ("/")):
 - **/list** will print a list of the current chatrooms.
 - **/create + [room name]** will create a new chatroom with that name, but the client will not automatically join it.
 - **/join + [room name]** will allow the client to join that chatroom. The client will be welcomed to the chatroom with the message: **"Welcome to " + roomName + "Type /exit to go back to waiting room"**. From this point, the client may send messages to the chatroom. These messages will be broadcast to all members of the chatroom.
 - **/exit** will take the client out of the current chatroom and put it back in the waiting room.

- PROTOCOL DESIGN

- *Sockets:*

The socket library is used for exchanging messages across a network by providing a form of inter-process communication (IPC). The messages travel as packets and must use the internet protocol. This means they must have a source IP address; destination address and a port number.

- Packet/message format/structure

Since UDP is connectionless network when two hosts are communicating, they do not initialize a connection, instead they exchange messages by targeting a specific address. Because of this, packaged data that is received by a client/host might end up being corrupt. The solution to this problem is using a checksum.

Checksum is a value used to verify the integrity of a file or data transfer and is part of the UDP header. When a client/host receives a packet, it unpacks it and does its own checksum calculation which is then compared with the checksum in the packets' header. If they are equal, then the data is not corrupt whereas if they are not then an error is detected. The file "***Error_detection.py***" has the all the functions that help with error detection and is then used by both the client and server

The UDP header has four fields; a source port number, a destination port number, data length and a checksum and they are defined by the technical specifications of the protocol. Each of these fields are 2 bytes (16 bits) in size.

The pictured below is the generic UDP packet structure.



The 'Data' segment consists of the message being sent, with the date and time appended to the end of the message in the format YYYY-MM-DD hh-mm-ss. This is a 19-character string. Throughout the program you will see code such as:

message = data [-19] → this extracts the message from the data

timestamp = data [-19:] → this extracts the timestamp from the data

This is done to separate the message from the timestamp. I felt it was important to include the timestamp within the Data and not in the Header so that it would be covered by the checksum error detection.

- **PROTOCOL ROBUSTNESS**

- **Loss detection:**

When the client or server sends a message, that message is appended to a list (a list of unacknowledged messages, variable name "*unackmsg*"). When the message arrives at its destination, the destination user will automatically send an acknowledgment. The acknowledgement of a message consists of the code "**ACK**" to denote that this is an acknowledgment message, together with the timestamp of the message. The timestamp, together with the source address of the acknowledgement is sufficient to uniquely identify any given message (i.e., these two attributes together serve as a "packet ID" of sorts.) In the client's case, the source will always be the server. In the server's case, it will need to use the clients' address to identify messages.

When an acknowledgement is received, that message is removed from the list of unacknowledged messages.

In both the server and client apps, there is a loss detection thread running. This thread iterates over the list of unacknowledged messages and checks what time that message was sent. If it has been longer that 5 seconds and an acknowledgment is still not received, the message is resent to the destination.

Loss detection is simulated by having the sender have a random chance of not actually sending the message (A message not being sent is akin to a message lost along the way). The server and clients both have an attribute called '*lossRate*'. '*lossRate*' is the percentage chance that a message will not be sent out (*lossRate* = 10 means 10% of messages will be lost).

- **Error detection**

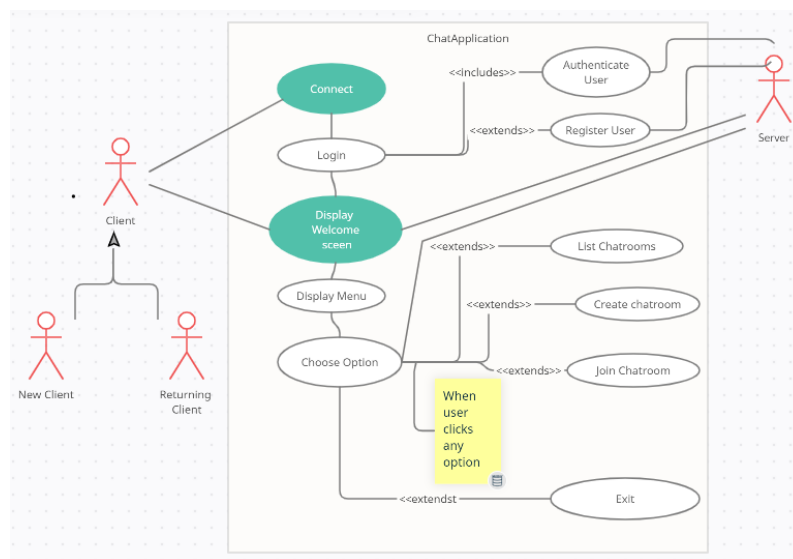
The header of the packet contains a checksum in order to detect corruption within the data. The method of handling packet corruption

could certainly have been improved had there been more input from other team members. At this stage, if a packet is corrupted (i.e., checksums do not match), the destination user will simply not send an acknowledgement. It will rely on the lossDetection thread of the sender to resend the message. I had hoped to be able to implement retransmit request functionality where, if the destination detects packet corruption it then requests the source to resend the message. Unfortunately, there was not enough time to implement such functionality.

Unfortunately, there was also not enough time to simulate packet corruption. However, because the error detection mechanism is essentially the same as loss detection mechanism, error detection is sufficiently accounted for and handled.

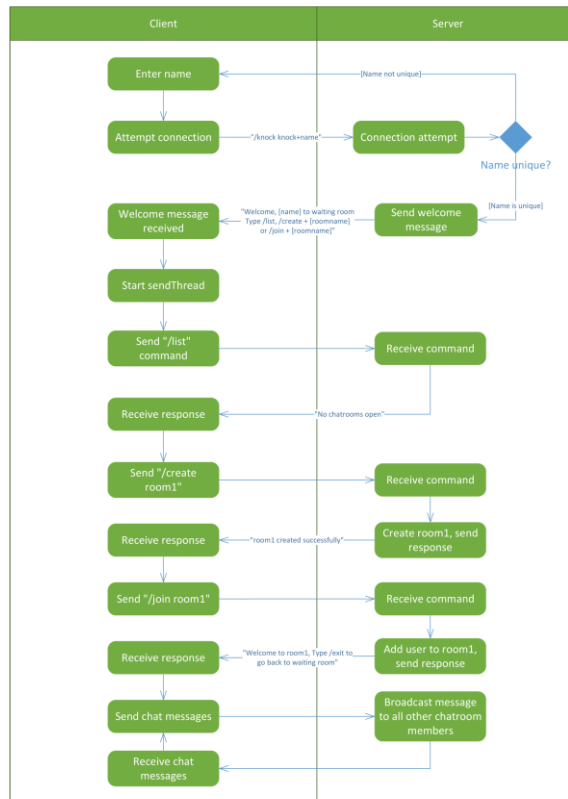
- UML DIAGRAMS

- *Use case diagram*



The use case diagram shows a broad view of how the client will interact with the chat application.

- *Sequence Diagram*



5. Result & Discussion

➤ Screenshots:

Client Interaction:

```

C:\Windows\py.exe
Enter name: cam
Welcome, cam, to the waiting room
Type /list, /create [room name] or /join [room name]
/list

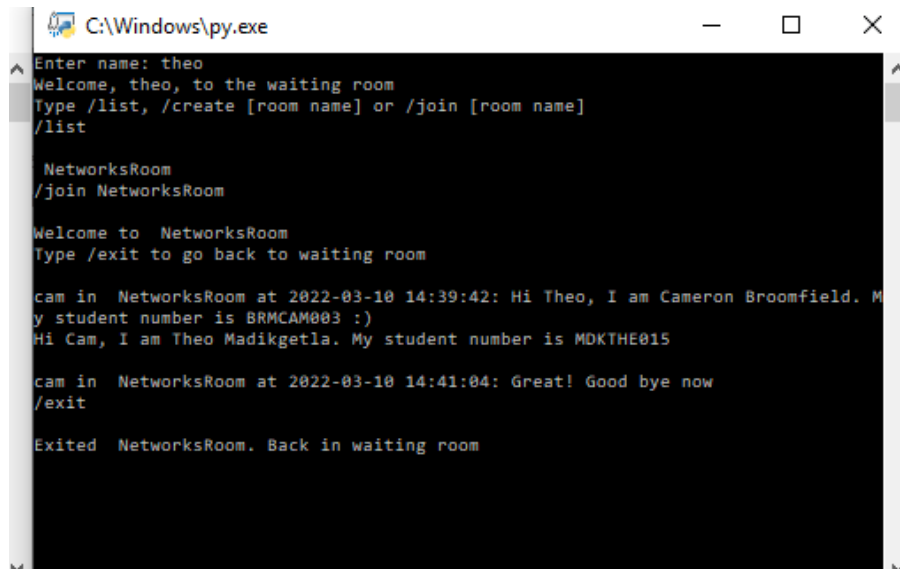
No chatrooms open
/create NetworksRoom

NetworksRoom created successfully
/join NetworksRoom

Welcome to NetworksRoom
Type /exit to go back to waiting room
Hi Theo, I am Cameron Broomfield. My student number is BRMCAM003 :)

theo in NetworksRoom at 2022-03-10 14:40:31: Hi Cam, I am Theo Madikgetla. My s
tudent number is MDKTHE015
Great! Good bye now
/exit

Exited NetworksRoom. Back in waiting room
  
```


A screenshot of a Windows command prompt window titled 'C:\Windows\py.exe'. The terminal shows a sequence of commands and responses for a chat application. The user 'theo' enters their name, is welcomed to a waiting room, and lists chatrooms. They then create a room named 'NetworksRoom' and join it. Another user, 'cam', joins the room and greets 'theo'. 'theo' responds with a greeting and student number. 'cam' then says goodbye and exits the room. The terminal text is as follows:

```
Enter name: theo
Welcome, theo, to the waiting room
Type /list, /create [room name] or /join [room name]
/list

NetworksRoom
/join NetworksRoom

Welcome to NetworksRoom
Type /exit to go back to waiting room

cam in NetworksRoom at 2022-03-10 14:39:42: Hi Theo, I am Cameron Broomfield. My student number is BRMCAM003 :)
Hi Cam, I am Theo Madikgetla. My student number is MDKTHE015

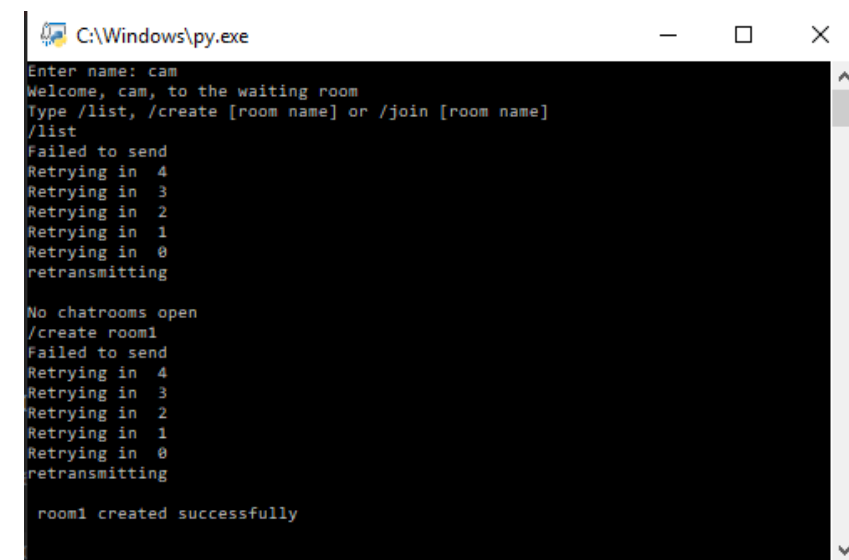
cam in NetworksRoom at 2022-03-10 14:41:04: Great! Good bye now
/exit

Exited NetworksRoom. Back in waiting room
```

The above two screenshots show an interaction between two clients. In the first screenshot, “cam” joined the server, listed the chatrooms and saw there were none open, created a chatroom called “NetworksRoom” and join that chatroom. In the second screenshot, “theo” then joined the server, listed the chatrooms (of which NetworksRoom was the only one), and joined networks room.

At 14:39:42, cam greeted theo and stated his full name and student number. At 14:40:31, Theo greeted cam back and stated his full name and student number. At 14:41:04, cam said good bye and exited the chatroom. Theo then also exited the chatroom.

Loss detection:

A screenshot of a Windows command prompt window titled 'C:\Windows\py.exe'. The terminal shows a user 'cam' entering their name and being welcomed to a waiting room. They attempt to list chatrooms but receive a 'Failed to send' error. This is followed by a series of 'Retrying in' messages with counts from 4 down to 0, and a 'retransmitting' message. After several more retries, the message 'room1 created successfully' appears. The terminal text is as follows:

```
Enter name: cam
Welcome, cam, to the waiting room
Type /list, /create [room name] or /join [room name]
/list
Failed to send
Retrying in 4
Retrying in 3
Retrying in 2
Retrying in 1
Retrying in 0
retransmitting

No chatrooms open
/create room1
Failed to send
Retrying in 4
Retrying in 3
Retrying in 2
Retrying in 1
Retrying in 0
retransmitting

room1 created successfully
```

In the above screenshot, cam joined the server. He requested the list of open chatrooms. This failed to send. You can see the loss detection thread counting for 5 seconds before retransmitting. The server responds with “*No chatrooms open*”. Cam then tries to create a chatroom called “room1”, which also fails to send. When it is retransmitted, the server responds with “room1 created successfully”

6. Conclusion

As shown above, we were able to develop a chat application and design a protocol that uses the User Datagram Protocol (UDP). Even though the UDP protocol is known to be unreliable, we designed our protocol in such a way that it will be able to achieve reliability when there is an exchange of packets through error detection mechanisms.

Reference:

1. [client-server architecture | Definition, Characteristics, & Advantages | Britannica](#)
2. [Computer network options - wired and wireless solutions for home and business \(rdcs.com\)](#)
3. Kuros, J & Ross, K 2017, *Computer Networking: A Top-down approach*, 7 edition
4. [UDP - Client and Server example programs in Python | Pythontic.com](#)