Student Numbers: KNGELM003 - NDHCAR002 - MGTBAT001

Introduction

The aim of this project was to create a python-based client-server chat application that uses UDP at the transport layer.

However, since the UDP protocol provides an unreliable service, we implemented our own protocols on the application layer to provide reliability and correctness when sending and receiving messages. These protocols are RUDP (Reliable User Datagram Protocol) and our own Messaging protocol, these will be detailed below.

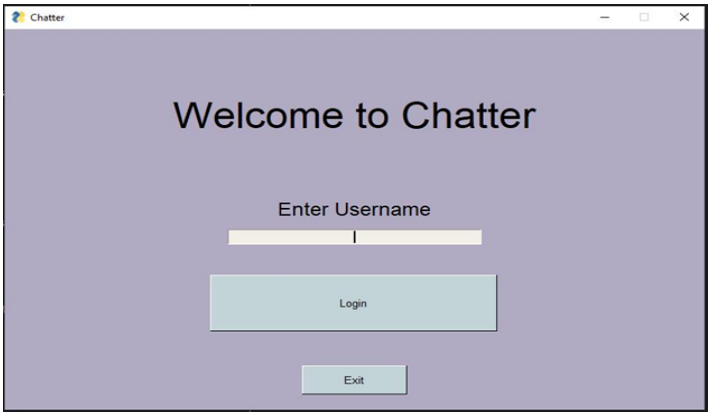
Client

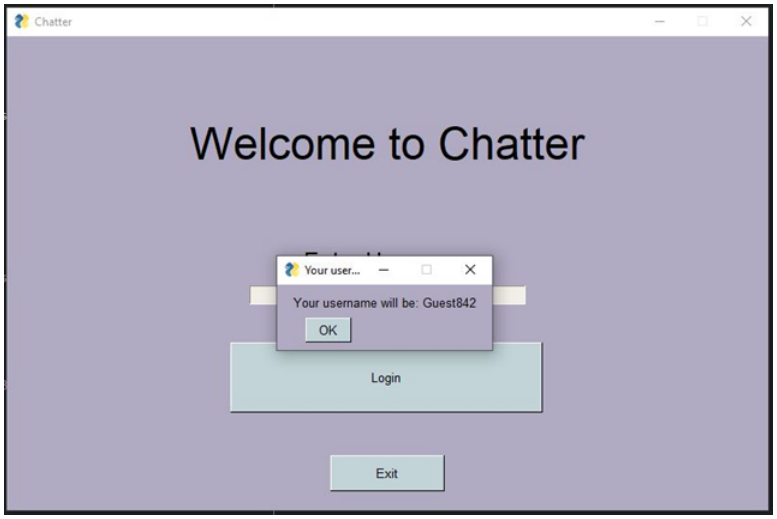
Chatter features a graphical user interface (GUI), created with the *tkinter* library in Python.

Login and Authentication:

The GUI opens with the *Login* screen, prompting for a username, this allows for users to connect to the chat room and send messages with an alias of their choice. The login button is bound to the return key on the user's keyboard, allowing for more convenient login.

This username is then used for authentication.





If the user does not enter a username and attempts to login, a username with a random number will be assigned to them.

Format: (Guest###).

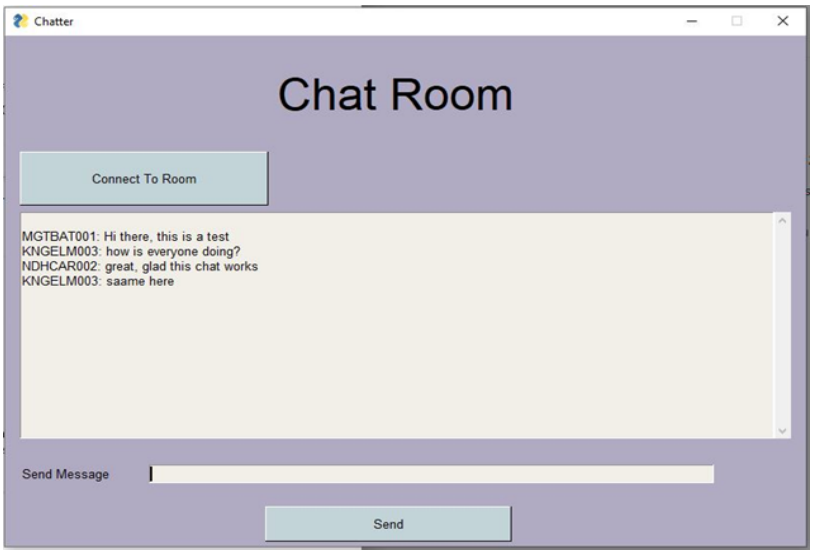
This will be displayed in a pop-up window.

Chatting

If the login is successful, the *Chat Room* screen will be displayed. The “Connect to Room” button activates the chat room. This means that all unread messages sent to the server from other clients are then retrieved and displayed to the chat block, as can be seen below.

The client GUI runs on a main/user thread. This is where the sending of messages occurs. Concurrent programming is used to run the *recieveMessages()* task on a low-priority, background (daemon) thread.

This method ensures that clients can receive and read messages while typing and sending their own messages.



Server

The server is responsible for mediating the exchange of messages between clients. The clients make requests to the server (e.g. to send/fetch messages) for which the server provides a response.

The messages and clients are stored using redis which is an in-memory database.[1] The messages stored contain a username, message content and timestamp while the clients have a username and timestamp for when they last performed the fetch message request. This timestamp is used to clear messages that have been viewed by all clients to prevent filling up memory with messages. This is done by getting the oldest timestamp out of all clients for when the fetch message request was performed. This timestamp is then used to clear all messages that are older than it.

Finally, the python event loop is utilised to allow the server to handle multiple different clients all continuously sending requests.

Protocols

- Please see Sequence Diagram, Appendix A below.

Chatter implements two protocols, namely: the messaging protocol and the RUDP protocol. The first thing we needed to achieve was reliability. We did this with an implementation of a protocol we call Reliable UDP (RUDP). RUDP is an application layer protocol that adds two things on top of UDP:

1. Loss detection - the sent packages are received by the receiver
2. Error detection - The received packages are not corrupted

[1] https://redis.io/

The messaging protocol is used for the application specific remote procedure calls. It works similar to HTTP.

Reliable UDP (RUDP)

RUDP works on a request-response basis. The sender sends a request packet with a unique id and a checksum. The receiver responds with a response package with a checksum, and whose id is that of the request package being replied to.

2 bytes checksum
36 bytes universally unique identifier (uuid)
data

The structure of an RUDP packet. The bold sections are covered by the checksum.

Loss detection

The loss detection algorithm is responsible for ensuring that the receiver (server) does receive the package being sent. RUDP is unique in the sense that it does not use separate signals to acknowledge a packet on the receiver side. The response for that packet is the acknowledgment that the packet has been received and is correct. If no response is received for a sent packet within 500ms, the sender retransmits that package and will keep retransmitting it every 500ms until it receives a response from the receiver. After 6 seconds of not having received a response from the receiver, the sender algorithm returns an error to the application signalling that the receiver cannot be reached.

The receiver algorithm works on a wait, process or bypass, and respond basis:

1. It waits for request packets from senders
2. When it receives a request it checks if it has received a request with the same id in the last 30 seconds. That is, is the packet a duplicate?
3. If the packet is NOT a duplicate, it passes the request to the server handler, caches the response against the request packet, and sends back the response packet to the sender
4. If the packet is a duplicate, it fetches the response it cached for that request and sends it back to the sender.

The decision to use a 30 seconds window to determine duplicates was rather random. Ideally, we want a window size big enough for the receiving algorithm to know if it has received a packet with a given id already. Due to device memory constraints, however, we need to limit this to be big enough to accommodate the longest delay we can expect from the network in moving a packet from the sender to the receiver. The devices we used for testing were connected to the same WIFI network. Therefore, this delay was very small - smaller than 1 second. We were being a little extra safe by going for 30 seconds in case we had to test the protocol on an extremely slow network. This means for even slower networks, the duplicate packet window will need to be increased which would result in increased memory usage.

Error detection

Both request and response packets have a 16 bit checksum value that hashes the uuid and data sections of the packet. The algorithm we are using for computing the checksum is the CRC-CCITT (xModem) algorithm. The checksum value is stored in Big Endian in the packet.

When an endpoint (sender or receiver) receives a packet it reads out the uuid and data section of the packet and computes the CRC-CCITT checksum value for these two sections. It discards the package as corrupted if, and only if, the checksum value it computes is not equal to that the package came in with.

Messaging protocol

The messaging protocol is an application specific protocol that runs on top of the reliability of RUDP, just as HTTP works on top of the reliability of TCP. The Chatter messaging protocol has methods, headers, and a body just like HTTP. The protocol implements the request-response model of RUDP in that every request is fulfilled with a response.

Request message

The request message contains a header line with a body line if applicable. The header line specifies an action that the client would like the server to perform. The body line provides additional information if required. The format is shown below:

```
Method: action
Data: '{"key":"value"}'
```

Possible values for the method are: FETCH, MESSAGE, LOGIN, EXIT. FETCH is used to fetch messages since the provided timestamp. MESSAGE lets the client send a message to the chat. LOGIN authorises a client to be able to send/receive messages. EXIT lets the server know the client has terminated.

The data is a serialised version of a python dictionary. This is done using the json library using the dumps (convert a python dictionary to a string) and loads (convert a string into a python dictionary). Example request formats for each method are provided below.

LOGIN format

```
Method: LOGIN
Data:'{"username": "john"}'
```

FETCH format

```
Method: FETCH
Data:'{"timestamp": 1646486140.689381}'
```

MESSAGE format

```
Method: MESSAGE
```

```
Data:'{"message": "Hello there!"}'
```

EXIT format

```
Method: EXIT
```

Response message

Response messages from the server contain two header lines with a body line. The header lines state the status name followed by the status message. The body line contains data if requested by the client. The format is as follows:

```
Status-name: name
Status-message: message
Data: serializedData
```

Possible values for the status name are: AUTHORIZATION-ERROR, DATA-REQUIRED, UNSUPPORTED-METHOD, FORMAT-ERROR, SUCCESS. AUTHORIZATION-ERROR specifies an error with either carrying out the LOGIN request or when performing other requests without having been authorised. DATA-REQUIRED is when the requested body line doesn't exist or value required is not within this body line. UNSUPPORTED-METHOD when the request method provided is not handled for. FORMAT-ERROR when the request message is not in a form recognizable to the server. SUCCESS when a request was completed with no errors.

The status message header line provides more information regarding the status name of the response.

The data body line will contain serialised data corresponding to the request method and is parsable using json loads method. Two example responses are provided below.

```
Status-name: AUTHORIZATION-ERROR
Status-message: Username is already taken
```

```
Status-name: SUCCESS
Status-message: Messages successfully fetched
Data: '[{"username":"John", "message":"Hi everyone",
"timestamp":1646486140.689381}]'
```

Testing (protocols and server)

Protocol testing

The RUDP protocol was tested against the server for the promised reliability. We created a module that encapsulated the python UDP API and in the send function, code was added to deliberately drop and corrupt packets at given intervals from both the client and server applications.

The protocol functioned as expected. Lost or corrupted packets are made up for by retransmission. If all packets are lost or corrupted, the sender algorithm time out to the user saying the server is unreachable - which is also the expected behaviour.

Server testing

The server was tested to see if it could successfully respond to all clients that are sending requests. To simulate different clients, multiple threads are created which all send requests to the server. Each thread is then responsible for listening to the server's response, then checking to see if it was a successful message. If successful, the standard output is only populated by response times tracking how long it took each thread to get a response from the server. If not, then that thread throws a ValueError which can be picked up on the main thread. This checking was done for the four different methods that a client can request of the server.

Appendix A : Protocol Sequence Diagram