CSC3002F

Assignment 1:
Client-Server Chat Application

Luke Slater, Ohio Imevbore, Erik Polzin

Introduction

Chat applications allow us to communicate with anyone in the world in real time through the internet. Most of us use them every day, be it Whatsapp, Facebook Chat, or Microsoft Teams. Each chat application requires its own protocol to govern the transfer of data between the various clients and servers. These protocols are designed according to the requirements and features of their chat application.

Usually these 'application layer' protocols use one of two protocols: UDP (User Datagram Protocol) or TCP (Transmission Control Protocol). TCP is a connection-oriented protocol that verifies reception of its data at the destination. It also allows for retransmission of lost packets. UDP is a connectionless Datagram-oriented protocol that does not guarantee the delivery of data to its destination, but is typically faster than TCP.

Our task is to design a Python chat application along with an application layer protocol to support it. The application must use UDP at the transport layer, as well as the client-server architecture.

Because UDP provides an unreliable service, our chat application has to implement an application layer protocol to achieve reliability and cohesion between the server and the "connected" clients.

Application architecture

The application is designed to follow a conventional server/client architecture. The client and server run in separate processes, on potentially different machines in the local network. There is only one server running at any given point in time, and its IP address must be static and known to the clients. The clients, on the other hand, can run at arbitrary addresses on the network.

They have different roles: the server waits for UDP datagrams (*'messages'*) from its clients, processes them, updates a database and then sends a response datagram back. Multiple GUI or CLI clients can request data from the server in this way, or send requests to update the server's database. This setup allows each client to create or log in to an account, broadcast or receive chat messages from specific 'rooms' and fetch already sent chat messages, among other things.

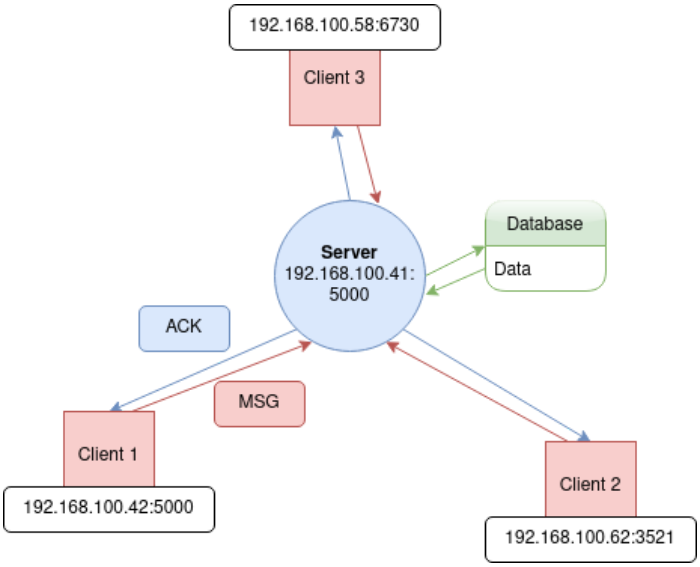

Figure 1: Client-Server Architecture

In the above diagram, each connection between server and client is facilitated by a UDP socket pair. Unlike short-term HTTP requests, these connections stay open for the duration of their client's lifespan once established, allowing UDP packets to be sent both to and from the server or client at any point during this time. This way, messages sent from a client to the server can be pushed to multiple other clients almost instantaneously. There is no need

for the recipient clients to constantly query the server for new messages the way email clients periodically check for incoming mail. The exact details of how client and server interact is described in the following subsection, 'Protocol Design'.

Protocol Design

As mentioned, the custom chat protocol utilizes the User Datagram Protocol, which allows datagrams to be delivered from one socket to another. These datagrams contain headers describing the sender's IP address, as well as a checksum for the recipient to validate the received data. These features are powerful, but not sufficient for a reliable chat application, which also needs verification of packet arrival and retransmission of any lost packets.

So for this assignment, datagram acknowledgement and timeout-based retransmission have been reimplemented using UDP: after each datagram is sent, the recipient responds with an 'ACK' message (for 'acknowledgement'). Meanwhile the sender starts a timer, estimating the amount of time it will take for the recipient to respond. If this timer runs out before an acknowledgement is received, the sender assumes the packet did not arrive and retransmits it. These 'lost packets' seldom occur over a local network, so we've simulated lost packets programmatically by dropping 20% of server-received packets.

Additionally, each datagram has a unique SEQN (sequence number) header entry, so that ACK responses can be traced to the request they are acknowledging. This is necessary since the client can send multiple datagrams concurrently and may receive acknowledgements in a different sequence than the one it sent the original requests in.

Message Format

This extended UDP protocol adds data to the default UDP header. This is where it stores some of the above-mentioned flags as binary data: SYN, ACK, FIN & SEQN.

The data body is stored in plain, unencrypted text. Its data is encoded in JSON, so that simple data structures may be sent to and from client and server. Each message must contain a 'type' listed in the table below. The format of the message in question will vary depending on this type.

Command Messages	Control Messages	Data Transfer Messages
SYN: Initialize connection **FIN**: Terminate connection **USR_LOGIN**: log user in	**ACK**: Acknowledge packet **MSG_RBA**: message delivered to all members	**CHT**: broadcast message to group **GRP_SUB**: subscribe user to group **GRP_ADD**: create a new group **GRP_HST**: list groups for member **USR_ADD**: create user **USR_LST**: list all users

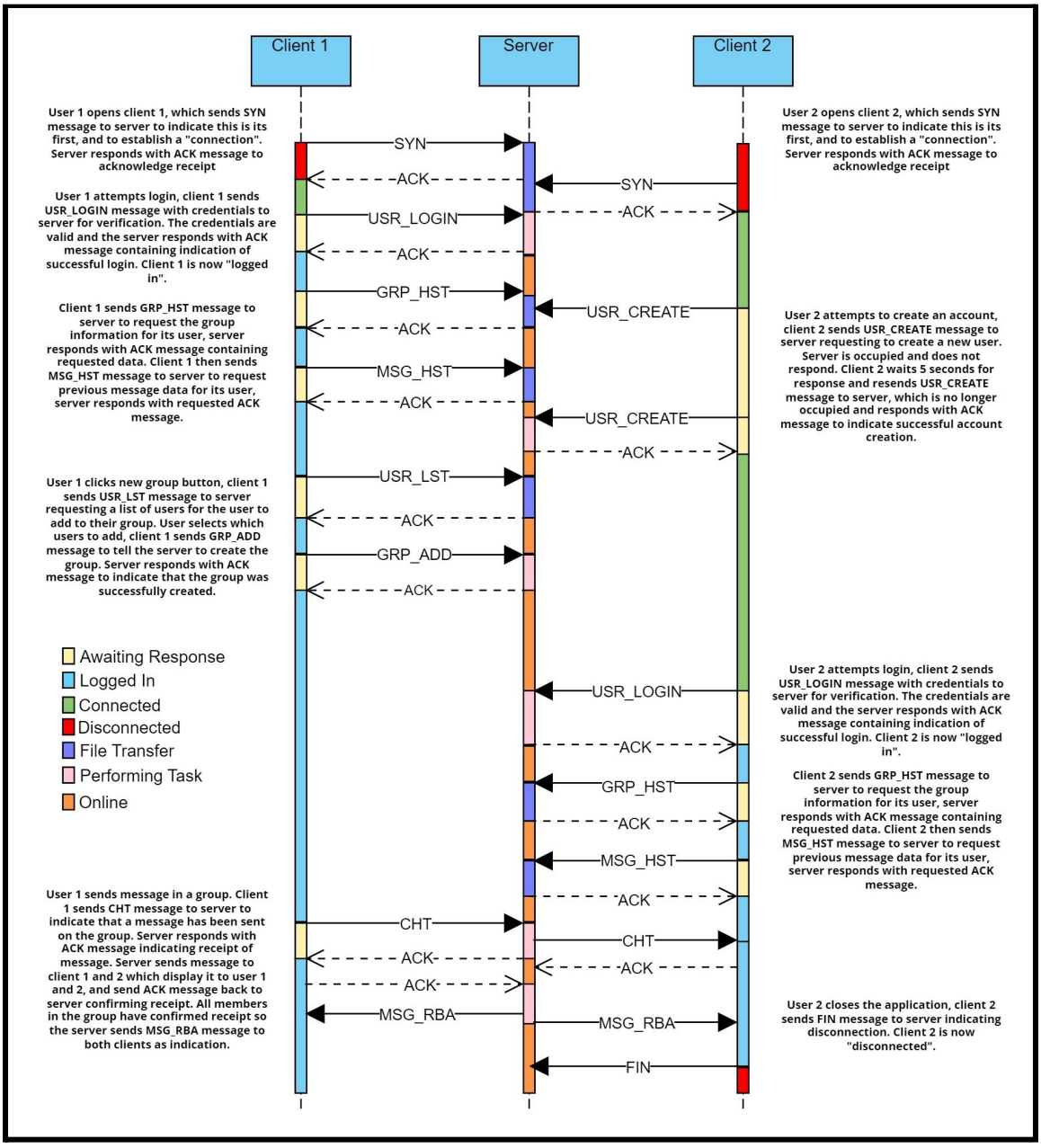

Figure 2: Sequence diagram representing Server/Client communication

Application Features

User Authentication

When starting the application, and after initializing a connection to the server, the GUI enables the server to validate a client's username and password. Passwords are stored as PDKDF2 hashed strings, which can be verified but not easily decrypted.

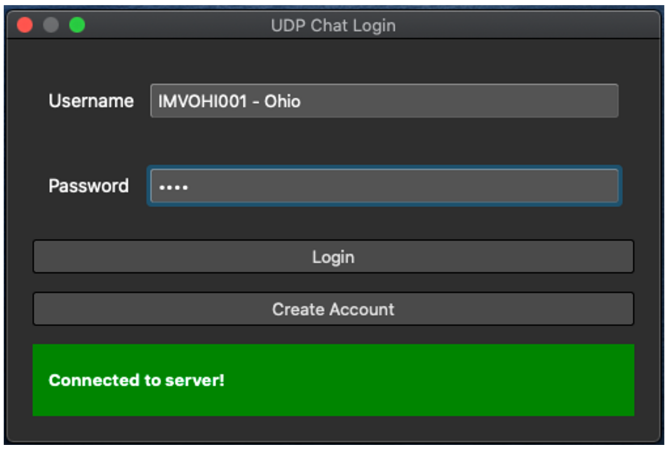

Figure 3: Client/User login window

By clicking the 'Login' button, the client sends a login message to the server. The server sends an acknowledgement message back to the client to confirm or refute the request if the password doesn't match. Behind the scenes, it also stores the user's last-accessed IP address & port number so that it can route any incoming chat messages to their address.

Real-Time Messaging

The application allows for a client to send and receive messages in real time. This means that multiple clients can communicate with each other in a live environment.

The client sends a chat message to the server to indicate a message has been sent. The server then sends a chat message to the other clients and the message is displayed in real time.

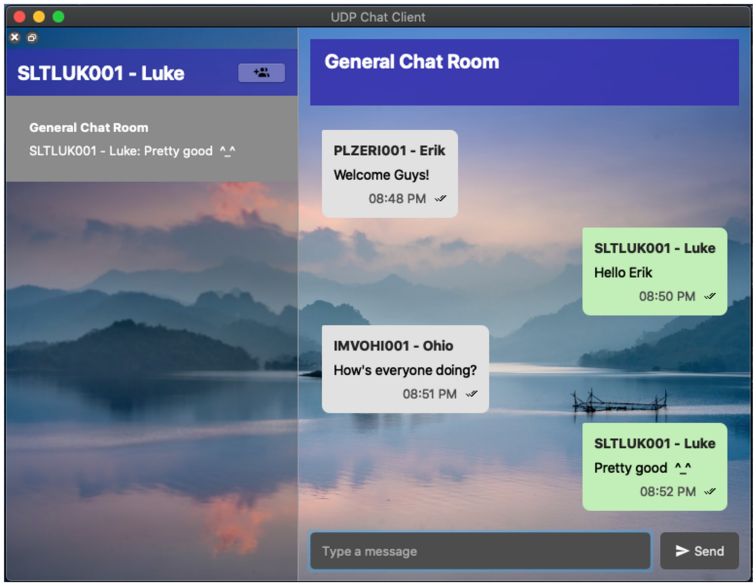

Figure 4: Real time messaging

Pair/Group Communication

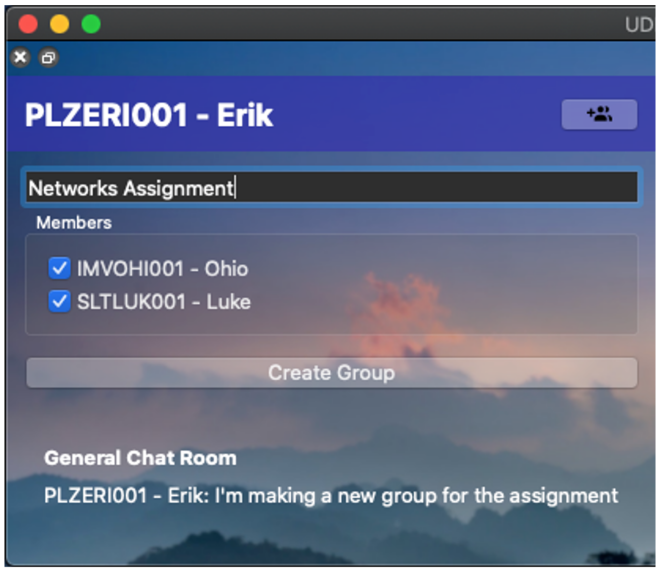

Figure 5: New group creation

The application allows a client to create a private communication channel with other clients through the server.

The user creates a new group by clicking the group icon in the sidebar, and selects the users they want to be part of this group. When the client sends chat messages in this group window, a message with a group identifier is sent to the server and the server then broadcasts the message to the corresponding clients.

Message Verification/Loss Detection

The application gives the client a confirmation tick that their message has been delivered. This ensures that messages are not lost and improves the reliability of communication.

Before the message has been delivered to the server, it is marked with an hourglass. Once it has been persisted to the database it is marked with a single tick. A double tick verifies that the message has been delivered to all members in a given group.

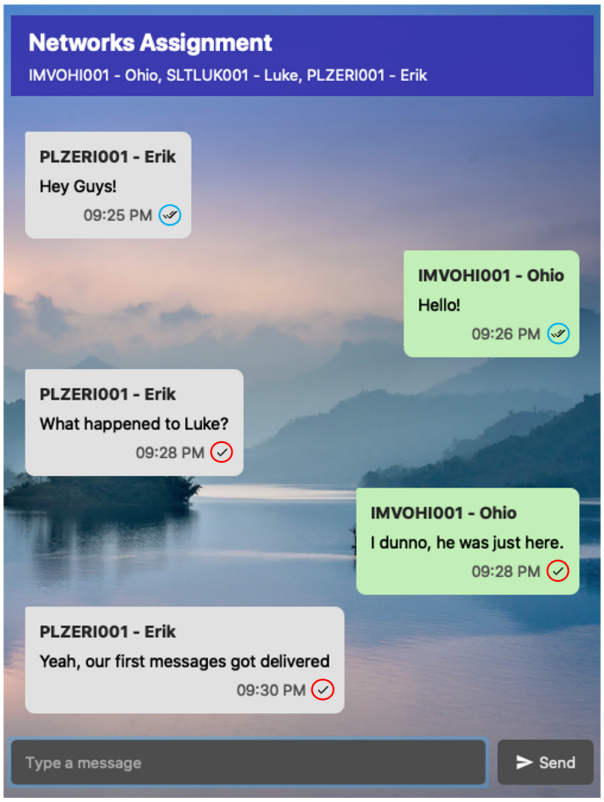

Figure 6: Message confirmation ticks

Client Re-connection

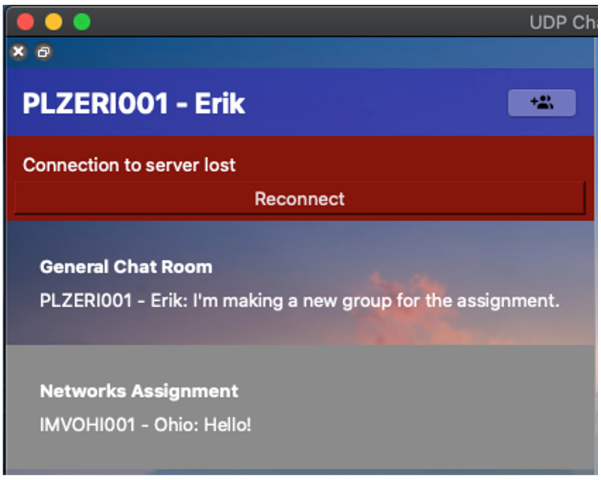

Figure 7: Client reconnection

The application allows the client to easily reconnect to the server if they disconnect, without losing any messages.

By default, clients give the server 5 seconds in total to respond, before deciding the server is unreachable.

Retrieval of Historical Messages

The application stores successful client chat messages in a database which allows the server to broadcast historical messages to the clients. This means that a client will have access to the chat messages that were sent to them while they were offline.

When clients send chat messages to a group, the server receives a message with the text data and the destination group. The server broadcasts the message to clients that are currently connected and disconnected clients receive these messages once they establish a connection.

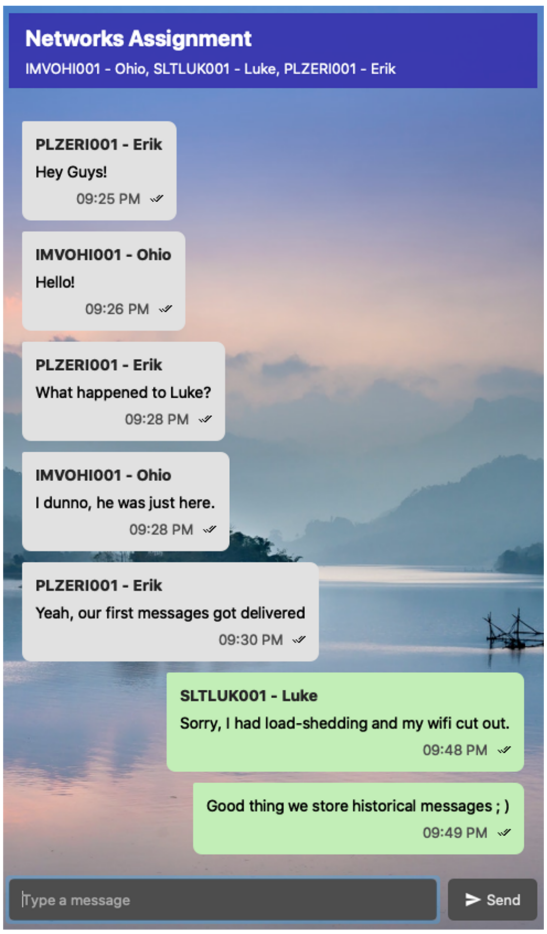

Figure 8: Retrieve historical messages