Internet Relay Chat:

**Abstract:**

This document describes a communication protocol for an internet relay chat client-server system for the CS 494/594 Internetworking Protocols class at Portland State University.

**1. Introduction**

An Internet Relay Chat (IRC) Protocol is an application that provides a communication framework between clients connected to a server. Using the client/server networking model, IRC clients are application programs that users use to transfer messages to other clients. Communication is done through channels, which are simulated “rooms” that allow users to choose which group of other clients to send/receive messages. In addition to using channels, clients can also send personal messages, enabling one-on-one communication between clients.

1a. Server

The server is an application program that serves as the relay point, accepting and sending messages to connected users. The server allows multiple clients to connect with each other, forming the structure of the IRC network.

1b. Client

Each user must have their own instance of the IRC application program in order to connect to the server. Each client, in order to access the server, must pass the IP address and source port. This way, the client will know how to connect to the server and establish a communication link.

1c. Channels

Channels represent the “rooms” in the IRC framework. These channels allow clients to group together into shared discussion forums of their choosing. When a client first joins the server, a channel is created. All subsequent clients that join are placed into this channel until a new channel is specified. Clients are able to create or join multiple rooms.

**2. IRC Specifications**

2a. Overview

Communication on an internet relay chat is done through the unidirectional method from one user, through the server, to the other user.

2b. Create a server to host the IRC

In order to begin the IRC, the server must first be initialized to host the clients. To do this, a server host must be initialized:

New\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

In the above code, socket.AF\_INET tells us that we are going to send the information using ipv4 (as opposed to ipv6). The second argument, socket.SOCK\_STREAM, means we are using a TCP connection, rather than a UDP connection, in which case we would use .socket.SOCK\_DGRAM”.

The next step is to bind our socket to an address and a port. This is given from the tuple argument passed to the create\_socket() method within our Client class:

new\_socket.bind(server\_address)

Once the socket is bound, the socket will need to "listen" for new connection requests, with a given value for the maximum number of allowed connections. In this case, a maximum of 30 clients is given:

new\_socket.listen(MAX\_ClIENTS)

At this point the server is running and is listening for a new client to join. This is implemented using a while loop that continues to listen until a new client instance is initialized, or if a keyboard interrupt causes the server to exit.

2c. Clients joining the Server

When a new client begins running the client program, the client is initialized in a similar way to the host. The client is given the same port, ‘12345’ to pass to the new\_socket.connect() method so that the client can access the same port that the server is listening on. This way, the client will establish a connection with the server, enabling the client to also access any other client that connects to this server.

When the client(s) are connected, each runs a while loop (just as the server) so that the program will continually wait for a command by the client, in which the client can start using the internet relay chat as desired.

2d. Reading in Commands

read\_socket, write\_socket, error = select.select(soc\_list, [], [])

In the above code snippet, the select.select() method is used as a direct interface to the underlying operating system. This method’s role is to monitor the socket for an event when it becomes readable. This method is implemented on both the server side and client side. On the server side, its role is to wait until a new connection is ready or if it is receiving data from an existing connection. On the client side, this method is used when a message is being given to the server. When the client types in a command on the command line, the call is given to message the server, in which the server handles the command (see supp.py) and a response is given back to the server from the client. This code is important because it allows the client to communicate messages and commands to the server.

**3. Userspace program and interacting with the server**

Given that this project hinges on the ability to communicate between various clients through a shared server, it is expected that the design and implementation should handle basic Chat functionality. Our current application gives clients (users) the ability to create, join, delete, list, and exit chat rooms, as well as, check to see available users currently connected to the server and message both a specific user or a group.

3a. How to run the application

Our application was developed using Python 3 and ran using python 3.7.3

All the commands are triggered by the “$” sign followed by their abbreviation. Below is a list of the commands and how to run them:

3b. Commands

1. **JOIN**

This command takes in one argument, the name of the channel you would like to join

It can be executed by typing the following:

*$j [the name of the chat room you would like to join]*

Example:

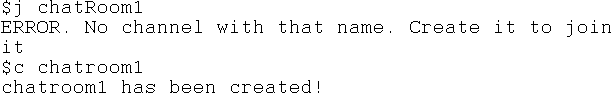


1. **CREATE CHAT ROOM**

This command takes in one argument, the name of the channel you would like to create. By default, when you create a chat room you are added to it. It can be executed by typing the following:

*$c [the name of the chat room you would like to create]*

Example:



1. **DELETE CHAT ROOM**

This command takes in one argument, the name of the channel you would like to delete

It can be executed by typing the following:

*$d [the name of the chat room you would like to delete]*

1. **EXIT CHAT ROOM**

This command takes in one argument, the name of the channel you would like to leave

It can be executed by typing the following:

*$e [the name of the chat room you would like to leave]*

1. **MESSAGE CHAT ROOM**

This command takes in two arguments, the name of the channel you would like to message and the message you would like to send. It can be executed by typing the following:

*$m [the name of the chat room] [the message you would like to send]*

1. **PERSONAL MESSAGE**

This command takes in two arguments, the name of the user you would like to message and the message you would like to send. It can be executed by typing the following:

*$m [the name of the user] [the message you would like to send]*

**LIST CHAT ROOMS**

This command takes no arguments and can be executed by typing the following:

*$lr*

**LIST CONNECTED USERS**

This command takes no arguments and can be executed by typing the following:

*$lu*

4. Error Handling

The clients and server must both be able to detect when the socket connection linking them together is terminated. When this occurs, the program will display a message notifying the client or user of a disconnection. When the server crashes/disconnects from the sockets, all clients receive an error message and their programs are subsequently terminated. When a client disconnects/crashes, the server is given a message stating which client has disconnected, and is still able to maintain functionality despite a client disconnection.

5. Conclusion and Future Works

Beyond the core requirements of this communication system, other features, such as a cloud based web server, P2P style infrastructure and security considerations were looked into. Some issues arose when setting up a web server due to problems establishing port forwarding for the server so that other devices could connect to the server. While a web server could be set up on a cloud server company such as AWS, there were problems getting other devices linked to this server, which again resulted in a local device communication system.

In terms of security, these messages have no protection against inspection, tampering or forgery. This is because the server will see all messages that are sent through the use of this service. If more time was given, functionality such as encrypting messages using an encryption key would be implemented. This would involve clients in a channel or two clients within a private message forum generating a shared encryption key that would be sent prior to messages been sent/received. An alternative would also be the generation of a unique key identifier for each client, and the client would only share this unique key with other clients they wished to open communication with.