CS494P
Internet Draft
Intended status: IRC Project
Expires: June 2019

John Lorenz IV
Portland State University
June 10th, 2019

Internet Relay Chat Project
RFC_Document_Lorenz.pdf

Abstract

This is an Internet Relay Chat program developed for a course at Portland State University. It is designed to demonstrate the interactions between a server and client, and to gain experience with Internet Protocols.

Status of this Memo

This document is submitted as an assignment for a project to develop an Internet Relay Chat application. As such, it is formed as an Internet Draft with a finite lifespan. After completion of this project, this draft is marked to expire by the end of June.

Table of Contents

1. Introduction

This is an IRC program for the CS494p class at Portland State University. There are implementations for both the server and client side of this application. Server hosts multiple users and users have the ability to join, create, display, rooms as well as send messages to rooms. The way users may communicate is by joining a room, and all messages directed towards that room will be relayed to all users that have chosen to join that room.

2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119, and should only be interpreted as such if in full capitalization.

3. Additional Information

All commands to join, send, create, and so on are preceding by a control indication character (#). For example, a user intending to send a message to a specified room is required to follow the formatting guidelines of constructing the command as such: #send <room number> <message>. Messages that are outside of the parameters as specified by the #list command will be rejected by the server and discarded.

As this is a program for an academic environment, I ensure no security, reliability, or otherwise. The server and client application programs for this project have arbitrary limitations placed upon them, such as buffer size limitations, maximum user capacity, or otherwise.

Although efforts were made to purge and account for specific cases of bad input and other forms of edge cases, there is no guarantee that this project is bug-free, and the committed tester will likely find that statement to be true. The intention of this project is to demonstrate understanding of

internet protocols, and socket programming, and so time spent on pedantic cases was limited to bare functionality.

4. Server and Client Application Structure

Significant effort was made to separate the server environment from the server operations. Because the client program is very simple in comparison, I will iterate on the server application first.

Firstly, the server follows the traditional format of constructing a file descriptor set and uses an asynchronous environment. That is, it accepts multiple users up to a maximum capacity and appears to process their commands concurrently. However, the server application doesn’t implement true multi-threaded programming, and is therefore not truly concurrent. This means that it is obviously not scalable to extremely large user sets.

The port used by the client and server can be changed by modifying the indicated #define values in the respective source files. The server and client addresses are specified as “INADDR_ANY”, and as such are intended to be run on the same machine.

Message payload between the server and client applications are intentionally very simplified strings of characters. Processing on the client side is trivial, whereas the server must perform deep string manipulation to extract meaningful information from the messages without modifying the overall intention of the message itself. This is done with extensive care through a series of classless general methods listed at the end of chat.h

User and Chat room functionality is specific to the server application, and is separated into two classes with entirely public methods. The two classes that are in charge of the chat room and user system are appropriately titled, “class room” and “class user” respectively.

5. Handling Invalid Input/Errors

If a user decides to attempt to join a room beyond the maximum capacity, or attempts to create a room beyond capacity, there are bound checks in place to prevent trivial overflow and crashing of the server. Moreover, as the server application data structure deploys pointer arrays, efforts are made to normalize and re-structure arrays when the list is modified. This is done to prevent rightward drift from normal use.

If a user inputs an invalid command, or a command that seems nonsensical, the server MAY make efforts to identify the problem for cases like empty #send messages, or invalid #joins. However, there is no guarantee that all edge cases are accounted for. In the event that the server detects a problem with supplied user input, a “SEND_DATA” error opcode will be returned back to the main function, indicating that a server message MUST be sent to the offending user to notify them that their request could not happen, or that their input was invalid.

6. Detecting User Input and Requests

User input is constantly filtered through the “screenInput” method. That is, each time the client sends a message to the server, the server will analyze the contents of the message by first checking to see if the first character in the string sequence is “#”. If it isn’t, then message is ignored. Otherwise, the message is then parsed to identify the command being sent. Based on the command, an operation code in the form of a flag is returned to main where it is then used by the “executeCommand” method. This is the central hub of all server operation. To perform any command related to the IRC, it must pass through this method.

6a. Join Functionality (#join)

When requesting to join a channel, the client MUST send a 2048 ASCII character text string to the server. The appropriate message format is enforced: #join <room number>. This will subscribe the user to the output messaging stream of that room. A user may leave a room using the same syntactic logic. Behind the scenes, the server application receives the message payload, and it begins to parse the string for relevant context.

6b. Leave Functionality (#leave)

When requesting to leave a channel, the client MUST send a 2048 ASCII character text string to the server. The proper encoding for this command is #leave <room number>. When leaving a room, the user requesting the operation MUST be removed from both the room listings, as well as have the chat room removed from their personal information account.

6c. Create Channel (#crt)

When requesting to create a channel, the client MUST send a 2048 ASCII character text string to the server in the following format: #crt <room name>. The server will automatically find a 'free' location and rename that room. The server does not implement dynamic rooms, so if all of the rooms are assigned to a name, then this function will return a FALSE, and mimic that response back to the user to let them know the operation failed. The basic logic behind the IRC chat program is that no news is typically good news.

6d. Delete Channel (#dlt)

When requesting to delete a channel, the client MUST send a 2048 ASCII character text string to the server in the following format: #dlt <room name>. The name input must be exact to the name displayed in the chat room listings. If there is no match to that name, then the server MUST return a FALSE code to notify the user that this operation failed. The convention of this command is similar to #join in the way that they both iterate over an array of a fixed size. Only one instance of a room may be deleted at any time, and the room deleted will be

the lowest room number given a circumstance with multiple room instances.

6e. Sending a message (#send)

When sending a message, the client MUST send a 2048 ASCII character text string to the server in the following format: #send <room number> <message>. Upon receiving this request, the server will iterate through the list of users that are connected and subscribed to that channel stream. Each user has a socket descriptor number, and an index associated with their character profile, and so a list of pointers is appended to each chat room. When a chat room has a message sent to it, the server will iterate through the pointer list and send the received message to each user in that feed using the socket descriptor number for each user.

7. Displaying Information

The server has a variety of display functions. Specifically, the server supports, display all rooms (#dsp), display all users (#dspu), and display my user info (#dspme), and list all commands (#list). When a user requests a display action, the server will ignore the remainder of the text that follows the command string. Display functionality operates in exactly the way one might anticipate.

In addition to displaying all rooms, the server will display the user count in that chat room and the users inside of it.

When displaying your personal user information, the server will show you the chat rooms you are a part of in addition to your user name.

8. Message Handling

The server and client MUST handle messages of length 2048. The messages are sent as a single ASCII character string. Currently, the server or client applications don't perform string bounds checking. The client application program is minimalistic in that it plays very little in the overall IRC chat application. Client applications, consequently, merely act as senders and receivers for messages.

The server, however, performs a significant amount of string parsing to extract meaningful semantics from the messages sent by the clients, and rejects messages that are out of syntactic bounds. The specific handling of messages will be described in the following paragraphs.

Firstly, when a client sends a message to the server application, the server MUST analyze the first character of the message. If the first character of the message is a # symbol, then the server proceeds to parse the string. Otherwise, the server MAY ignore or reject the message. When parsing the string, the server analyzes the characters following the # symbol, and constructs a command string locally. When it encounters a space character, the parsing halts and the resulting command string is analyzed for semantics. If it is a valid command, a flag is set and returned to the caller function to be used in command execution.

Secondly, now that the server knows what command is desired and is in the execute command method, it will use a switch statement that will perform different operations specific to the flag that was returned. If the user requested to send a message to a room, the received message is then parsed again, this time ignoring the command string: "#<some command>", and will extract first the room number that the message is being sent to. Once the room number has been extracted, it will be represented in two ways locally. The first representation is a character representation, and the second representation will be a numerical representation. We will return the numerical representation, but the character representation is used to then perform a substring search for the first occurrence of the number in the received message string. This will return a pointer to exactly the beginning location where the number

occurs in the received message string. This is important to prevent the need to re-parse a user's message.
Thirdly, now that a pointer to the substring exists, the server increments that pointer in memory to bypass the room number. The resulting string will be the message. So, in conclusion, a user requests a message to be sent using #send <room number> <message to send>. By doing the above three steps, the only thing that will be left is <message to send> with "#send" and <room number> omitted from the message. Understanding the string manipulation in this application is important to understanding the application as a whole.

9. Security Considerations

This application is not designed to be secure, and WILL NOT provide a safe environment for communication. The program MAY be used by anyone that wishes to do so, but it WILL NOT be updated to improve security.

10. Normative References

[RFC 2119] Brader, S., "Key words for use in RFCS to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17.487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.