Josh Evans

Term Project: *Internet Relay Reborn Chat*

Design Document

Table of Contents

1 Introduction

This document describes the design for the Internet Relay Reborn Chat (IRRC) application infrastructure.

1.1 Purpose and Scope

This document is to describe, in-depth, the design of the IRRC system. It is to be used as a general outline for the implementation of the software.

1.2 Target Audience

This document is to be used by the development team for the purposes of implementation. For general design requirements, please see the IRRC requirements document.

1.3 Terms and Definitions

IRRC stands for Internet Relay Reborn Chat, which is the name of the software to be developed. Similarly, IRC stands for Internet Relay Chat, which is the type of software.

2 Design Considerations

This section will outline the core design considerations from the requirements.

2.1 Constraints and Dependencies

This IRRC system must have a user login for the chat server. Therefore, there must be a way to store and authenticate user account data. There must be both server and client applications, with the server handling multiple client connections.

2.2 Methodology

The methodology used to develop this software will be a type of Agile development using rapid prototyping and feature-driven development. Due to minimal dependencies in the features of this software, it is safe to fully flesh out a feature before moving onto the next. This will be done in iterations, letting each feature become more complete every time.

3 System Overview

As previously stated, the IRRC infrastructure uses two main components – client and server. The client will communicate with the server using the java socket class. When the client attempts a connection to the server, user authentication is required. Once the user logs in or makes an account, the client will be able to read input from the user to send to the server, and display output back to the user from the server. The server will use channels as conduits of information – channels being the general chat and private chats. The server will keep a limited backscroll for each channel in a text file to display. If more than the specified number of lines are written to a channel, the first line of the channel text file will be removed and the next line will be written at the end. Each time a new message is written to a channel, the server updates the necessary text file and redisplays the file's contents in the channel history. Every time this happens, the server will resend the text data for the channel to all of the connected clients in the channel.

4 System Architecture

This section will describe the system architecture to be used for the IRRC infrastructure. The topics covered here include the underlying language, main classes, and some of the crucial methods in these classes.

4.1 Client-Server Communication

The server will communicate with the client using the java socket class. This class will allow sending and receiving of text data over an internet connection. The server application will be started with an argument listing the port number the server socket should listen on. When receiving connections, the server will queue the acceptable connections at the port, and then service them simultaneously once they are connected through the use of multiple threads. In each thread, the server will simultaneously listen for incoming data and send new data to the client. To send new data, the server will push the contents of the updated channel file to all connected clients, with a *Push()* method. When a client connects with a known username and password, the channels which they are connected to will be pushed out. This allows clients to read new messages and view their message history in their connected channels.

4.2 Server Core

To handle multiple connections, the server will use threads. When a new connection comes in, the server will accept it and give it to a ConnectionThread object to service it. The ConnectionThread objects will be processed in a linked list. The server will function as one large infinite loop until it is instructed to exit. Each thread will also function in a similar way.

Before each thread enters its service loop, the client will have to authenticate with a username and password. There are two types of authentication – new and existing. A new authentication means the user created a new account.

After each thread is authenticated, the ConnectionThread class will contain variables for the IP address and username of the connected client.

4.2.1 Server Files

The server will use data files to track message history and user information. These files will be stored in the server directory. There will be three types of files – channel files, the authentication file, and the log file. Channel files will contain message history for the given channel. Public channels will be marked with the '#' symbol in the client and files ('!' in front of the filename in the directory). These files will have the format:

```
!channel_name.txt:

User1: This is a message in a public channel.
User2: These messages will be stored in the channel file.
User3: All connected users in the channel can see these!
```

Private channels, aka direct message history, will have the format:

```
User1!User2.txt:

User1: This is a direct message to User2.
User2: Only we can see these messages.
```

These files will be updated by the server whenever new data designated for them is read. By default, these files will have a line limit of 100 lines. If the line limit is reached, each additional line will be added at the end of the file while the first line in the file will be removed.

The authentication file will contain a list of usernames and passwords, as well as a list of connected channels for each user. This file will have the format:

```
Auth.txt:
```

```
User1 password123 !allchat.txt User1!User2.txt
User2 password456 !allchat.txt User1!User2.txt
User3 password789 !allchat.txt !videogames.txt !cs300.txt
```

Users have the option to leave certain channels, or opt-out of seeing their direct message history with other users. If both users choose to do this for a direct message history, it will be deleted. Additionally, if all users leave a channel, the channel will be deleted. This is simply an optimization and upkeep decision.

The log file will be the master file for the server. It will contain all updates to other files and server output, including new user accounts, failed authentication attempts, direct messages, server commands, etc. All entries in this file will be in the raw input format received from the client, or the raw output format from the server. The console output for the server application will reflect the changes to the log file. This file will be named Log<space>[date server started]<space>[unique number].txt. Note that the log file will contain sensitive user information, and should not be shared with non-operators. The file will look something like:

```
Log 5-8-2017 1.txt:

Server started.
Listening for connections on port 4444.
Client 192.168.390.45 connected.
->192.168.390.45 AUTH1 User1 passwodr123
<-192.168.390.45 Username or password not recognized.
->192.268.390.45 AUTH1 User1 password123
Client 192.168.244.500 connected.
->192.168.244.500 AUTH0 User2 password456 password456
<-Auth User2 password456
->User2@192.168.244.500 JOIN #allchat
<-Server@localhost ANNOUNCE #allchat :User2 has joined the
server! Welcome!
->User1@192.168.390.45 MSG #allchat :Hello User2! Welcome!
<-User2@192.268.244.500 MSG #allchat :Hello! :D
```

The arrows indicated incoming and outgoing data from the server threads. Lines without these indicators are messages directly from the server.

4.2.2 Server Commands

Users can issue commands in their messages, such as “/join <channel>” or “/leave” or “/quit”. These command messages are not displayed in any channel, but make requests directly to the server. These commands are handled in the service loop for the ConnectionThread.

4.3 Client Core

The IRRC client will be a UI-based application which will connect to the server on a port and send/receive data on that port. Upon starting, the client will ask the user to input the server IP address and port to connect to. Once connected, it will open into authentication. When authentication is complete, the application will move into the main UI. The client application will use text boxes to capture and send all input to the server. This allows spontaneous and instant input to the server at any time, so that it can be handled correctly in the ConnectionThread loop. In the main UI for the client, there will be other elements which will be populated by the incoming server information. These elements will display the current channels, online users, and the message history for the current channel. To switch channels, the user can click between the open ones. To leave a channel, the user can issue a “/leave” command, or simply click the X button on the right-hand side of the channel element. The client will also have a loop which reads data in from the server in order to refresh the message histories and channel list.

5 Detailed System Design

5.1 Core Loop

As previously stated, the server will use a core loop to accept connections on the specified port. This loop will look like:

```
While(1) { // server
     // check for expired connections and remove them
     // check for new connections
     // if new connection, create ConnectionThread object
     // add ConnectionThread object to data structure
}
```

5.2 Authentication Loop

At the beginning of each thread, the client will need to authenticate in order to access any channels. This will be done in another loop in order to correctly define the behavior of the process. This loop will look like:

```
While(1) { // authentication
     // check for type
     // if type is new:
          // check if username contains invalid chars
          // if username is invalid, print error
          // check if username already exists
          // if username exists, print error
          // check if passwords match
          // if passwords don't match, print error
          // add username and password to auth file
          // add to channel #allchat
          // display welcome message in #allchat
     // if type is existing:
          // check if username exists
```

```
        // if username doesn't exist, print generic error
        // check if passwords match
        // if passwords don't match, print generic error
        // if username exists and password matches, break
}
```

5.3 Thread Loop

Each ConnectionThread will process authenticated client's requests in yet another loop. This loop will basically look like:

```
While(1) { // thread
     // listen for input with timeout of 1ms
     // if input, check for command
     // if command, send to process method
     // if not command, write to appropriate file
     // push updated channel files
}
```