Ben Totten

CS594

Fall 2020

b-IRC RFC

1. INTRODUCTION

b-IRC is a python-based protocol that uses asynchronous named pipes to communicate and python sockets to connect. Internet Relay Chat (IRC) has been used by system administrators at Portland State University (PSU) since Janaka Jayawardena helped bring internet to the college in the late 70s and early 80s. Many new students who joined the Computer Action Team (CAT) would learn their first bash lessons by writing bots for PSUs irc network. Due to the necessity for a more beginner-friendly method of communication, the CAT has moved to more modern chat applications for day-to-day duties, but a small contingent of students still exist on freenode. This RFC aims to outline a new (and more simplified) implementation of the old IRC protocol. All traffic will use port of the users choosing, easily set at the top of the client.py/server.py files. The traditional port for irc is 6667/TCP to avoid needing root access in order to function, however the default is set to ‘localhost’:5000.

1.1 Servers

As specified in the original IRC RFC, b-IRC servers provide a way for both clients and other servers to connect and send messages to each other, thus forming the IRC network. The server allows multiple clients to connect to it by creating a new thread with each connection. These threads are set up daemons, and so end when the program ends, ensuring a graceful closing. The server can directly be communicated with via the named FIFO pipe called server.io. You can echo commands into it in order to shutdown. The server handles client crashes and disconnects using the python socket library and error try and catch blocks.

1.2 Clients

A client is a non-server connection that connects to a server to send network traffic and messages. On first connection, the client provides its nickname/handle, the ip address its connecting from, and the port its connecting from. Clients can connect to a server, create a channel, list all channels, join a single channel and multiple channels, leave a channel, list all members in a channel, and send different messages to different channels. Using try and catch blocks, the client gracefully handle server crashes and attempt to reconnect. The user can send commands to the client with the ‘_’ leading character, or can send messages using a provided script that writes to a named pipe called client.io. This pipe allows the user to send messages from the command line.

1.3 Channels

Just like the original IRC protocol, a channel is a group of one or more clients which all receive messages addressed to that channel. All channels begin with a leading ‘#’. Users are by default

joined to the ‘#’ channel when they are connected. This protocol has no special channel modes or elevated users. The channels will be stored in a dictionary on the server, and each channel entry will contain a dictionary of client tuple keys for clients in that channel consisting of the clients IP address, and the clients real port number. This client key resolves to the clients nickname. A channel is automatically deleted when the last user leaves it.

1.4 Messages

All messages are specially formatted to inform the server what type of message they are, and what they’re meant to cause the server to do. Messages contain several parts: prefix (*optional), channel header, command (*optional), and finally the message. They are structured as such

:*<Nick>**!*** *<IP>/<Port>* PRIVMSG #*<chan>: /<Command> <Message>*

2. SPECIFICATION

The protocols outlined here can be used for all connection types.

2.1 Character set

This protocol, like the original RFC, is based on codes composed of eight bit octets and is compatible with the Unicode standard.

2.2 Messages

Messages between clients and servers are asynchronous and do not wait for a reply.. Each IRC message consists of three parts: an optional prefix (for server forwarded messages or initial joins containing the users nick), the channel, the command, and the command parameters or the message.

Like the original protocol, prefixes are indicated with a single leading colon ‘:’ with no whitespace between the colon and the prefix, and are used to indicate the origin of the protocol.

All IRC messages are terminated with a New Line. Messages will not exceed 280 characters. This protocol uses a similar Backus-Naur Form (BNF) as the original protocol

2.2.1 The BNF representation for this is:

<message> ::= [':' <prefix> <SPACE>] PRIVMSG #<channel>: /<command> <SPACE> <params> <nl>

<prefix>   ::= [<nick>! <ip>/<port>]

<command> ::= /<letter> { <letter> } | <number> <number> <number>

<SPACE>   ::= ' ' { ' ' }

<params>   ::= <SPACE> [':' <trailing> | <middle> <params>]

<middle>   ::= <Any *non-empty* sequence of octets not including SPACE or NUL or CR or LF, the first of which may not be ':'>

<trailing> ::= <Any, possibly *empty*, sequence of octets not including

NUL, CR or LF, or NL>

<crlf> ::= CR LF or NL

2.2.3 Prefixes

Messages with prefixes indicate that either the client is connecting for the very first time, or the message has been forwarded from another server. If the message has a prefix, the server throws out the connecting clients IP and port number, as these are from a different server, not the true client, and reads the IP, port number, and nickname from the prefix instead. A prefix is indicated by a ':' leading character followed immediately by the nickname of the client.

2.2.4 Private Messages

If instead of a leading # before a channel, there are only alphanumeric characters, the server will know this message should be directed to a specific user and not a channel. The server then reverse-looks up the handle, and forwards the remaining message to the client IP/Port tuple it fetched from the rooms.

2.2.5 Commands

Commands are indicated by a forward slash '/' after the channel portion of the message. If this '/' is missing, the remaining bytes are assumed to be entirely message, and are forwarded to the appropriate channel or user.

2.2.5.1 Join

A user can use /join to either join a room or create one if it does not exist yet.

2.2.5.2 Part

/part #<channel> will cause a client to leave a channel.

2.2.5.3 List #chan

/list #chan will list all members in a channel

2.2.5.4 List

/list will list all channels

2.2.5.5 Quit

/quit will cause the server to disconnect the client.

3. CONCEPTS

This outlines the functionality of this IRC protocol

3.1 Client to Group Communication

Clients will be able to send messages to whichever channel they have joined. Channels allow the dynamic joining and parting of clients with the actual conversation only being sent to servers which have users on a given channel. Just like the original RFC, If there are multiple users on a server in the same channel, the message text is sent only once to that server and then sent to each client on the channel. This action is then repeated for each client-server combination until the original message has fanned out and reached each member of the channel.

4. Functionality

This describes the different functions that are addressed by this protocol

4.1 Server

The server is able to disconnect from clients and gracefully handle client crashes. The servers are configured to allow multiple clients to connect to them and host multiple channels. Servers will forward traffic according to its destination, outlined in the octet attached to each message.

4.2 Client

Clients are able to connect to individual servers and channels within them. Clients can connect to multiple channels simultaneously, and can create channels. They can list all already created channels, and can leave channels. They can send messages to channels and receive messages, as long as their session is alive. They can disconnect from a server and gracefully handle server crashes. They can also list all members in a channel. Clients can also direct message other clients, as long as the sender has the nickname of the client they wish to reach and the receiver is connected to the server.

5 Security

5.1 Secure Socket Layer Protocol

The server automatically generates, if not already in existence, SSL certificates, should the client connection request them.

5.2 Storage

The server does not retain copies or logs of the messages being sent between clients and rooms. In the future, this irc implementation will incorporate end-to-end encryption.

6. Network Protocol

IRC is implemented on top of TCP, like the original protocol, due to the reliability of TCP and its assurance that message packets will make it through, leaving complete, coherent messages.

7. Message viewing

Clients will write out received messages into a chat log stored in a channel directory.

8. The Future

The creator of this program would like to see a separate data structure to hold client information, instead of searching through the room lists until a match is made. There are some algorithmic changes that should be made instead of $O(n^2)$ or $O(n^3)$ time iterative loops for the retrieval functions. The creator would also like to incorporate semaphores for thread synchronization, instead of the current loops present in main. The creator would also like to make more rugged the ability for server-to-server communication to take place.