IRC
Internet-Draft
Intended status: Standards Track
Expires: November 16, 2018

B. Creeley
Portland State University

May 15, 2018

Internet Relay Chat Client/Server Protocol
IRC RFC

Abstract

This rfc describes an Internet Relay Chat (IRC) protocol that was created for a Networking Protocols class project at Portland State University. The protocol allows for clients to connect to the chat server, joining and creating rooms, sending and receiving chat messages based on subscribed rooms, and leaving chat rooms.

The server is implemented using TCP/IP sockets and assumes that the client application does this as well.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on November 16, 2018.

Table of Contents

1. Introduction

The purpose of this rfc is to explain how the interaction between the client and server for the IRC protocol being presented. This is done using TCP/IP sockets for both the client and server.

For this IRC protocol there is a central server which can consist of multiple chat rooms called channels. Each channel has a list of users that can communicate through this channel. When one user messages the channel the server will forward this message to every other user subscribed to this channel.

The client can also request the available channels (chat rooms) and the users in a specified channel. The server will receive this request and determine whether or not it is valid and relay the information back to the client who requested it.

2. Basic Information

Clients can connect to the server using TCP/IP sockets as long as the server is available. The server is always listening for new TCP connections on port 5013. As soon as the client successfully connects it can send any of the protocol’s available messages.

The client will use a predifined message structure to send all requests. In all cases the server will send a response back to the client. The message structure is explained in subsequent sections.

In some cases the server will respond to the sending client, but as noted above other messages don't require a response from the server. When clients disconnect from the server, the server will remove the client from any of the channels they were subscribed to. It is up to the user on reconnection to reconnect to these channels.

For simplicity each message has the same length. The first member of a message is the message type. This is followed by a union that defines the available messages. The message structure is described below.

3. Message Infrastructure

3.1. Message Format

```
struct message {
        uint8_t type;
        uint32_t length;

#define RESP_INVALID                    0
#define RESP_SUCCESS                    BIT(1)
#define RESP_INVALID_LOGIN              BIT(2)
#define RESP_INVALID_CHANNEL_NAME       BIT(3)
#define RESP_NOT_IN_CHANNEL             BIT(4)
#define RESP_ALREADY_IN_CHANNEL         BIT(5)
#define RESP_SERVER_HAS_NO_CHANNELS     BIT(6)
#define RESP_CANNOT_GET_USERS           BIT(7)
/* Don't add any defines greater than BIT(31) */
#define MAX_MSG_RESP_NUM                BIT(31)
        uint8_t resp_code;

        void *payload;
}

/* payload for LOGIN message type */
struct server_login {
        char src_user[USER_NAME_MAX_LEN];
        char password[PW_MAX_LEN];
}
```

```
/* payload for JOIN message type */
struct channel_join {
        char src_user[USER_NAME_MAX_LEN];
        char channel_name[CHANNEL_NAME_MAX_LEN];
}

/* payload for LEAVE message type */
struct channel_leave {
        char src_user[USER_NAME_MAX_LEN];
        char channel_name[CHANNEL_NAME_MAX_LEN];
}

/* payload for CHAT message type */
struct channel_chat {
        char src_user[USER_NAME_MAX_LEN];
        char channel_name[CHANNEL_NAME_MAX_LEN];
        char text[CHAT_MSG_MAX_LEN];
}

/* payload for LIST_CHANNELS message type */
struct channel_list {
        char channels[MAX_CHANNEL_LIST_LEN];
}

/* payload for LIST_USERS message type */
struct channel_user_list {
        char users[MAX_CHANNEL_USER_LIST_LEN];
};
```

3.1.1. Message Definitions

- type - 8-bit value holding the message type.
- length - 32-bit value holding the length of the entire message.
- resp_code - 32-bit value holding the result of a client request.
- payload - depending on the message type this will be any of the above listed message specific structures.
- src_user - the name of the user that the message originated from.
- channel_name - name of the channel that this message applies to.
- result - server uses this field to send a result back to the client
- text - text of a chat message.

3.1.2. Request Usage

The message structure used was defined in preceding sections. This will be used along with the message types to build a request message. This is done by flattening all of the data into an array of bytes. The requester (most likely the client) will need to set the type, length, and in some cases the payload field for the message to be valid. Also only some of the messages require a response which is shown above.

3.2. Message Types

```
MSG_TYPE_INVALID = 0,
LOGIN            = 1,
JOIN             = 2,
LEAVE            = 3,
CHAT             = 4,
LIST_CHANNELS    = 5,
LIST_USERS       = 6,
MAX_MSG_NUM      = 255
```

3.2.1. Message Types Definition

- MSG_TYPE_INVALID(client) - If no message type is specified then default of 0 is considered invalid. This makes it so the client doesn't accidentally send a valid default message type.

- MSG_TYPE_INVALID(server) - If no message type is specified then the default of 0 is considered invalid. This makes it so the server can't accidentally send back a success/fail (depending on which is 0).

- JOIN(client) - How the client either joins and/or creates a channel depending on whether or not it has already been created. The user does this by specifying their username and the channel they wish to join/create. If the user is already a member of the channel they get the response RESP_ALREADY_IN_CHANNEL. On success the server will send back RESP_SUCCESS in the resp_code.

- JOIN(server) - How the server sends the response back to the sending client. The server will fill the payload with the same data that was sent from the client. If the user is already in the channel the server will return RESP_ALREADY_IN_CHANNEL to the user. On success it will return RESP_SUCCESS.

- LEAVE(client) - How the client tells the server it wants to leave the channel specified in the payload. On success the client will

receive a response from the server of RESP_SUCCESS. On failure the client will receive RESP_NOT_IN_CHANNEL.

- LEAVE(server) - The server will send a response of RESP_SUCCESS if the client was successfully removed from the specified channel. If the client was not in the channel the server will respond with RESP_ALREADY_IN_CHANNEL. The server will fill the payload with that data the client sent on the LEAVE request.

- CHAT(client) - This is how the client sends messages to a specific channel they are subscribed to. If the user tries to send a message to a non-existent channel or a channel they are not part of then they will get a RESP_NOT_IN_CHANNEL from the server. On success the client will receive the payload it send along with a response of RESP_SUCCESS.

- CHAT(server) - The server uses the message to either forward the message to all users in the channel specified or to send a RESP_NOT_IN_CHANNEL to the client who supplied the CHAT message. The payload will be filled with the payload that was sent from the source client.

- LIST_CHANNELS(client) - The client is requesting the current list of channels from the server. The client should not fill the payload for this request. This request will either return RESP_SUCCESS and a list of the channel names separated by a ":" or a RESP_SERVER_HAS_NO_CHANNELS in the case that no channels have been created.

- LIST_CHANNELS(server) - If the server has no channels it will not fill the payload and the resp_code will be RESP_SERVER_HAS_NO_CHANNELS. Otherwise the server will fill the payload with all of the channels separated by a ":" and set the resp_code to RESP_SUCCESS.

- LIST_USERS(client) - The client uses this message to list the users of the channel specified. If the channel does not exist or if the user is not in the channel the server will respond with a resp_code of RESP_CANNOT_GET_USERS. Otherwise the server fills the payload with the channel’s users separated by a ":" and sets the resp_code to RESP_SUCCESS.

- LIST_USERS(server) - The server will determine if the channel list specified is valid and return no payload and the resp_code RESP_CANNOT_GET_USERS if the request is invalid. Otherwise the server will fill the payload with the channel’s users separated by a ":" and set resp_code to RESP_SUCCESS.

3.3. Response Codes

```
#define RESP_INVALID                    0
#define RESP_SUCCESS                    BIT(1)
#define RESP_INVALID_LOGIN              BIT(2)
#define RESP_INVALID_CHANNEL_NAME       BIT(3)
#define RESP_NOT_IN_CHANNEL             BIT(4)
#define RESP_ALREADY_IN_CHANNEL         BIT(5)
#define RESP_SERVER_HAS_NO_CHANNELS     BIT(6)
#define RESP_CANNOT_GET_USERS           BIT(7)
uint32_t resp_code;
```

3.3.1. Response Code Definitions

- RESP_INVALID - When the client or server doesn’t specifically set the response this will be the default (assuming all message fields were set to 0 initially).

- RESP_SUCCESS - The server sets the resp_code to this when sending a successful response back to the client.

- RESP_INVALID_LOGIN - The server sets the resp_code to this when a client sends a LOGIN with invalid username and/or password.

- RESP_INVALID_CHANNEL_NAME - The server sets the resp_code to this when a client tries to send a CHAT, LIST_USERS, or LEAVE message with a non-existent channel.

- RESP_NOT_IN_CHANNEL - The server sets the resp_code to this when a client tries to send a CHAT, LIST_USERS, or LEAVE message to a channel that they are not subscribed to.

- RESP_ALREADY_IN_CHANNEL - The server sets the resp_code to this when a client tries to JOIN a channel they are already subscribed to.

- RESP_SERVER_HAS_NO_CHANNELS - The server sets this when there are currently no channels created if the client tries to LIST_CHANNELS.

- RESP_CANNOT_GET_USERS - The server sets this when the client tries to LIST_USERS in a channel, but is not currently part of this channel.

4. Disconnecting

If a client disconnects from a server, the server will remove the user from any subscribed channels. If the client wants to reconnect to all of the channels it was previously connected to it needs to keep track and handle this on the client side. The only information the server maintains about a user is their username and password.

5. Security

This protocol is extremely insecure. The server stores usernames and passwords in plain text and none of the messages are encrypted. Any of the messages could be monitored to steal this information.

6. References

CS494 example RFC

7. Full Copyright Statement

8. Acknowledgments

This document was prepared using xml2rfc.tools.ietf.org

Author’s Address

Brett Creeley
Portland State University
1825 SW Broadway
Portland, OR 97201

Email: bcreeley@pdx.edu