

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.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Basic Information	2
3. Message Infrastructure	3
3.1. Message Format	3
3.1.1. Message Definitions	4
3.1.2. Request Usage	4
3.2. Message Types	4
3.2.1. Message Types Definition	5
3.3. Response Codes	6
3.3.1. Response Code Definitions	7
4. Disconnecting	7
5. Security	7
6. References	8
7. Full Copyright Statement	8
8. Acknowledgments	8
Author's Address	8

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 predefined 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

```
Message Type    - 1 byte
Message Length  - 4 bytes

/* One-Hot Response Codes */
RESP_INVALID           0
RESP_SUCCESS           BIT(0)
RESP_INVALID_LOGIN     BIT(1)
RESP_INVALID_CHANNEL_NAME BIT(2)
RESP_NOT_IN_CHANNEL    BIT(3)
RESP_ALREADY_IN_CHANNEL BIT(4)
RESP_SERVER_HAS_NO_CHANNELS BIT(5)
RESP_CANNOT_GET_USERS  BIT(6)
/* Don't add any defines greater than BIT(31) */
MAX_MSG_RESP_NUM      BIT(31)

Message Response - 4 bytes
Message Payload  - Size dependent on payload

/* payload for LOGIN message type */
Source User - 16 bytes
Password    - 16 bytes

/* payload for JOIN message type */
Source User - 16 bytes
Channel Name - 16 bytes

/* payload for LEAVE message type */
Source User - 16 bytes
Channel Name - 16 bytes
```

```
/* payload for CHAT message type */
Source User   - 16 bytes
Channel Name  - 16 bytes
Chat Text     - 256 bytes

/* payload for LIST_CHANNELS message type */
Flat array of channel names delimited by a ":"

/* payload for LIST_USERS message type */
Flat array of usernames delimited by a ":"
```

3.1.1. Message Definitions

- o Message Type - 8-bit value holding the message type.
- o Message Length - 32-bit value holding the length of the entire message.
- o Response Code - 32-bit value holding the result of a client request.
- o Payload - Size dependent on the payload shown above.
- o Source User - 16-byte username of the client that the message originated from.
- o Channel Name - 16-byte channel name that this message applies to.
- o Chat Text - 256-byte 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. All of the messages require a response from the server.

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

- o `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.
- o `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).
- o `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`.
- o `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`.
- o `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`.
- o `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.
- o `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.

- o 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.
- o 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.
- o 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.
- o 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.
- o 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

RESP_INVALID	0
RESP_SUCCESS	BIT(0)
RESP_INVALID_LOGIN	BIT(1)
RESP_INVALID_CHANNEL_NAME	BIT(2)
RESP_NOT_IN_CHANNEL	BIT(3)
RESP_ALREADY_IN_CHANNEL	BIT(4)
RESP_SERVER_HAS_NO_CHANNELS	BIT(5)
RESP_CANNOT_GET_USERS	BIT(6)

Response Code - 4 bytes

3.3.1. Response Code Definitions

- o 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).
- o RESP_SUCCESS - The server sets the resp_code to this when sending a successful response back to the client.
- o RESP_INVALID_LOGIN - The server sets the resp_code to this when a client sends a LOGIN with invalid username and/or password.
- o 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.
- o 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.
- o 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.
- o RESP_SERVER_HAS_NO_CHANNELS - The server sets this when there are currently no channels created if the client tries to LIST_CHANNELS.
- o 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

Full Copyright Statement Copyright (C) The Internet Society (1999). All Rights Reserved. This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English. The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns. This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

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`