CS494 Jordan Malubay
Internet Draft Portland State University
Intended status: IRC Class Project Specification December 5, 2021
Expires: June 2022

Internet Relay Chat Class Project
draft-irc-pdx-cs494.txt

Status of this Memo

This Internet-Draft will expire on June 5, 2022.

Copyright Notice

Abstract

This memo describes the communication protocol for an IRC-style client/server system for the Internetworking Protocols class at Portland State University.

Table of Contents

1. Introduction

This specification describes a simple Internet Relay Chat (IRC) protocol by which clients can communicate with each other. This system employs a central server which ''relays'' messages that are sent to it to other connected users.

Users can join rooms, which are groups of users that are subscribed to the same message stream. Any message sent to that room is forwarded to all users currently joined to that room.

Users can create new rooms on the server that another user can subscribe to.

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 [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying significance described in RFC 2119.

In this document, the characters ">>" preceding an indented line(s) indicates a statement using the key words listed above. This convention aids reviewers in quickly identifying or finding the portions of this RFC covered by these keywords.

3. Basic Information

All communication described in this protocol takes place over TCP/IP, with the server listening for connections on port 7734. Clients connect to this port and maintain this persistent connection to the server. The client can send messages and requests

to the server over this open channel, and the server can reply via the same. This messaging protocol is inherently asynchronous - the client is free to send messages to the server at any time, and the server may asynchronously send messages back to the client.

As is described in [4.2], both the server and client may terminate the connection at any time for any reason. They MAY choose to send an error message to the other party informing them of the reason for connection termination.

The server MAY choose to allow only a finite number of users and rooms, depending on the implementation and resources of the host system. Error codes are available to notify connecting clients that there is currently a high volume of users or groups accessing the server.

4. Message Infrastructure

4.1. Generic Message Format

```
header
Struct.pack(HEADER_FMT,
            opCode,
            length)

payload
Struct.pack(msg_fmt,
            Header,
            payload)
```

4.1.1. Field definitions:

- header.HEAD_FMT – specifies the type for the opCode and the length fields, this affects the size of the header struct. The packet is built dynamically based on the types. This allows the header to change to accommodate more opCodes in the future.
- header.opCode - specifies what kind of message is contained within the payload. Variable format based on HEADER_FMT.
- header.length - specifies how many bytes of payload follow the header in this message (excludes header size).
- Both client and server MUST validate that the length is valid for the provided opcode. If not, the entity that detects the error MUST notify the sender of the error and provide the error code IRC_ERR_ILLEGAL_LENGTH to the opposite party (see error messages section).

- payload - variable length payload. Messages that do not contain a payload will be packed with a '0' string. As the header cannot contain a length of 0. All payloads are UTF-8 strings that are converted to binary strings when packed.

4.1.2. Operation Codes (opCodes)

```
OPCODE_ERR              = 0
OPCODE_HELLO            = 1
OPCODE_GET_ROOMS        = 2
OPCODE_LIST_ROOMS       = 3
OPCODE_LIST_USERS       = 4
OPCODE_CREATE_ROOM      = 5
OPCODE_JOIN_ROOM        = 6
OPCODE_LEAVE_ROOM       = 7
OPCODE_SEND_MSG         = 8
OPCODE_BROADCAST_MSG    = 9
```

4.2. Error Messages

```
header
Struct.pack(HEADER_FMT,
            opCode = OPCODE_ERR,
            length = 2)

payload
Struct.pack(ERROR_FMT,
            errCode)
```

4.2.1. Usage

MAY be sent by either the client or the server to inform them of an error from the sender. If the error occurs from decoding the packet the sender SHOULD resend the packet every time they receive the error.

4.2.2. Field definitions:

- errCode - specifies the type of error that occurred

4.2.3. Error Codes

```
ERR_UNKNOWN         = 0
ERR_ILLEGAL_OPCODE  = 1
ERR_ILLEGAL_LENGTH  = 2
ERR_NAME_EXISTS     = 4
ERR_ILLEGAL_NAME    = 5
```

```
ERR_TOO_MANY_USERS  = 7
ERR_TOO_MANY_ROOMS  = 8
ERR_NOT_IN_ROOM     = 9
```

5. Client Messages

5.1. First message sent to the server

```
header
Struct.pack(HEADER_FMT,
            opCode = OPCODE_HELLO,
            length = len(username.encode())

payload
Struct.pack(msg_fmt,
            username.encode())
```

5.1.1. Usage

Before subsequent messages can be sent, a connecting client MUST provide a username.

The server MUST associate the client's username with the socket connection of the user. This message SHOULD be sent only once; if the server receives the message more than once, the server will associate the new username with the user’s socket.

5.1.2. Field Definitions

- username specifies the name the connecting client wishes to use.
- username must not be the same name provided by any other currently connected client. If the name already exists, the server MUST return the error IRC_ERR_NAME_EXISTS. The client can then attempt to reconnect with a different name.

5.2. Listing Rooms

```
header
Struct.pack(HEADER_FMT,
            opCode = OPCODE_GET_ROOMS,
            length = 2)

payload
Struct.pack(msg_fmt,
            ‘0’)
```

5.2.1. Usage

Sent by the client to request a list of all the rooms currently on the server.

5.2.2. Response

Server MUST return a comma separated string of rooms on the server with opcode OPCODE_LIST_ROOMS.

5.3. Creating Rooms

```
header
Struct.pack(HEADER_FMT,
            opCode = OPCODE_CREATE_ROOM,
            length = len(payload))

payload
Struct.pack(msg_fmt,
            room.encode())
```

5.3.1. Usage

Sent by the client to create a chat room. If a room by that name exists, the server MUST send an error with the error code ERR_NAME_EXISTS. After creation the server MUST at the user to the room and send the user an OPCODE_JOIN_ROOM to inform the user they are in the room.

Upon joining a room, the server MUST send an OPCODE_LIST_USERS message to the user so the user can see the current user is the room.

5.3.2. Field Definitions

- room - Name of the room to join or create. MUST not contain any whitespace.

5.4. Leaving a Room

```
header
Struct.pack(HEADER_FMT,
            opCode = OPCODE_LEAVE_ROOM,
            length = 2))

payload
Struct.pack(msg_fmt,
            '0')
```

5.4.1. Usage

Sent by the client to leave a chat room.

Upon receiving this message the server MUST remove the client from the specified room and MUST send an OPCODE_LIST_ROOMS message to the user to display the rooms list to the user.

The server MUST send an ERR_NOT_IN_ROOM message to users that leave request when the client is not currently a member a room.

5.5. Sending Messages

```
header
Struct.pack(HEADER_FMT,
            opCode = OPCODE_SEND_MSG,
            length = len(payload))

payload
Struct.pack(msg_fmt,
            message)
```

5.5.1. Usage

Sent by a client to send a text message to a room.

The server MUST send an OPCODE_BROADCAST_MSG message to all users in the same room as the sending client.

5.5.2. Field Definitions

- message - Message to send to the room.

  MUST be less than MAX_PACKET_SIZE minus HEADER_SIZE.

  MUST UTF-8 formatted string before bye conversion.

  If the message breaks any of these rules, the server MUST provide the error code ERR_ILLEGAL_MESSAGE.

6. Server Messages

6.1. Listing Rooms

```
header
Struct.pack(HEADER_FMT,
            opCode = OPCODE_LIST_ROOMS,
            length = len(payload))

payload
Struct.pack(msg_fmt,
            roomlist)
```

6.1.1. Usage

Listing response message sent by the server to the client. Used for listing rooms that are on the server.

6.1.2. Field definitions

- roomlist – UTF-8 comma separated string constructed from the server’s list of rooms.

6.2. Listing Users

```
header
Struct.pack(HEADER_FMT,
            opCode = OPCODE_LIST_ROOMS,
            length = len(payload))

payload
Struct.pack(msg_fmt,
            userlist)
```

6.2.1. Usage

Listing response message sent by the server to the client. Used for listing user that are in a room on the server.

6.2.2. Field definitions

- userlist – UTF-8 comma separated string constructed from the server’s list users in a room.

6.3. Forwarding Messages to Clients

```
header
Struct.pack(HEADER_FMT,
            opCode = OPCODE_BROADCAST_MSG,
            length = len(payload))

payload
Struct.pack(msg_fmt,
            message)
```

6.3.1. Usage

Used to send all clients of a particular room a message from a user in that room.

All users in the room are send the message including the original sender.

6.3.2. Field Definitions

- msg - Message sent to the room.

  MUST be less than MAX_PACKET_SIZE minus HEADER_SIZE.

  MUST UTF-8 formatted string before bye conversion.

7. Error Handling

Both server and client MUST detect when the socket connection linking them is terminated by actively sending traffic. If the server detects that the client connection has been lost, the server MUST remove the client from all rooms to which they are joined. If the client detects that the connection to the server has been lost, it MUST consider itself disconnected.

8. Conclusion & Future Work

This specification provides a barebones framework for multiple clients to communicate with each other via a relaying server. The protocol is incomplete as current client and server messages do not all have proper responses leading to errors in state synchronization. These errors can place clients in states that will prevent them from properly communicating with the server until they establish a new connection with the server.

9. Security Considerations

Messages sent using this protocol have no security and packets that are intercepted and inspected could easily be decoded from their byte strings back into UTF-8 with little work. The header size can be changed but after looking through a few packets can also be figured out with very little work.

The payload could be ciphered using encryption but the server does not have any method of key distribution messages so the protocol would need to be expanded to allow that.

10. IANA Considerations

None

10.1. Normative References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

11. Acknowledgments

This document was prepared using draft-irc-pdx-cs594-00.txt by Byron Marohn.

Authors' Address

Jordan Malubay
Portland State University Computer Science
1825 SW Broadway, Portland, OR 97201

Email: jmalubay@pdx.edu