CS594
Internet Draft
Intended status: IRC Class Project
Specification

Maxwell Oakes
Portland State University
Mar 8, 2022

# Internet Relay Chat Class Project
irc-draft-cs594-v1-0.txt

Abstract

This memo describes the communication protocol for an IRC-style client/server system for the Internetworking Protocols class at Portland State University. This protocol allows for a single server to host many clients. These clients can create channels to send text chat to each other. Clients can also send messages directly to each other via whispers.

Table of Contents

1. Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 2
  1.1 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . 2
    1.1.1 Server . . . . . . . . . . . . . . . . . . . . . . . . . 2
    1.1.2 Client . . . . . . . . . . . . . . . . . . . . . . . . . 2
    1.1.3 Message . . . . . . . . . . . . . . . . . . . . . . . . 2
    1.1.4 Channels . . . . . . . . . . . . . . . . . . . . . . . . 3
2. Message Structure . . . . . . . . . . . . . . . . . . . . . . . 3
  2.1. Message Object . . . . . . . . . . . . . . . . . . . . . . . 3
  2.2. Message Fields . . . . . . . . . . . . . . . . . . . . . . . 3
  2.3. OP Codes . . . . . . . . . . . . . . . . . . . . . . . . . . 4
  2.4. Usage and Example . . . . . . . . . . . . . . . . . . . . . 6
3. The Client . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
  3.1. Login . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
  3.2. Listening . . . . . . . . . . . . . . . . . . . . . . . . . 7
  3.3. Chatting . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  3.4. Commands . . . . . . . . . . . . . . . . . . . . . . . . . . 9
    3.4.1 Channels . . . . . . . . . . . . . . . . . . . . . . . . 9
    3.4.2 Channel Selecting . . . . . . . . . . . . . . . . . . . . 9
    3.4.3 Private Messages . . . . . . . . . . . . . . . . . . . . 10
    3.4.4 Server Information . . . . . . . . . . . . . . . . . . . 10
    3.4.5 Quitting . . . . . . . . . . . . . . . . . . . . . . . . 11
    3.4.6 Aliases . . . . . . . . . . . . . . . . . . . . . . . . 11
4. The Server . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  4.1. Initialization . . . . . . . . . . . . . . . . . . . . . . . 11
  4.2. Listening . . . . . . . . . . . . . . . . . . . . . . . . . 11
    4.2.1. Listening for New Clients . . . . . . . . . . . . . . . 11
    4.2.2. Listening for Messages from Existing Clients . . . . . . 12
  4.3. Handling Client Messages . . . . . . . . . . . . . . . . . . 12
    4.3.1. Text Chat . . . . . . . . . . . . . . . . . . . . . . . 12
    4.3.2. Whispers . . . . . . . . . . . . . . . . . . . . . . . . 12
  4.4. Non-Text Commands . . . . . . . . . . . . . . . . . . . . . 12

4.4.1. Channel Management . . . . . . . . . . . . . . . . . . . . . . 12
4.4.2. Client Quitting . . . . . . . . . . . . . . . . . . . . . . . 13
4.4.3. Table Information . . . . . . . . . . . . . . . . . . . . . . 12
4.5. Server-side commands . . . . . . . . . . . . . . . . . . . . . . 14
5. Channels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
5.1 Channel Structure . . . . . . . . . . . . . . . . . . . . . . . . 14
6. Server-side Client Structure . . . . . . . . . . . . . . . . . . . 15
7. Error Handling . . . . . . . . . . . . . . . . . . . . . . . . . . 15
8. Conclusion & Future Work . . . . . . . . . . . . . . . . . . . . . 16
9. Security Considerations . . . . . . . . . . . . . . . . . . . . . 16
10. IANA Considerations . . . . . . . . . . . . . . . . . . . . . . . 17
11. Normative References . . . . . . . . . . . . . . . . . . . . . . 17
12. Acknowledgments . . . . . . . . . . . . . . . . . . . . . . . . . 17

## 1. Introduction

This protocol described in this document is a smaller variation to the Internet Relay Chat (IRC) [RFC1459]. This protocol allows clients to communicate with each other via simple text message. This system employs a client-server architecture where the central server broadcasts client-originating messages.

## 1.1. Definitions

### 1.1.1. Server

The server is a single instance; a single server will be sending and receiving information to and from clients. It is run with an input of a hostname, port and name. The server does all message processing and name-checking.

### 1.1.2. Client

Multiple clients can connect to one server instance. The client is created with an input of an address and port of a server to connect to. The client's role is only to log into the server, send and receive messages. Very little is done outside of this. In this document, "client" will be referring to the program running this chat protocol, and "user" will be referring to the person that is using the client, and inputting information into the client's text input field and reading incoming messages.

### 1.1.3. Message

Messages are the piece of information that is passed between client and server. It contains several fields, one of which is the actual text content that the user will see. Other fields are for internal use by the server and client to decide how the entire message is processed. In this document, "message" will refer to this object, "text" will refer to what the user visually sees on the client, and "packet" will refer to the raw binary data that is sent over the internet.

### 1.1.4. Channel

A channel is a grouping of clients within a server. Within one channel, a client can send messages to other clients that are 'subscribed' to the channel. A user can input into the client to electively create, join, leave and delete rooms. A channel can have no clients, or all clients in a server, or any number of clients in between.

## 2. Message Structure

### 2.1 Message Object

The message is the key piece of information that is used by server and client, and the only type of information that is sent (other than default socket behavior like EOF). Both the server and the client encode the message into bytes to send the message, and decode them into a message structure.

### 2.2 Message Fields

A message is composed of several fields. All of which are used by the server or client for processing, or for the user for information.

a. Sender
This field is a string of the original sender of the Message, this will be the username of the client. If it is a message from the server, it will be the name of the server that was specified during server Initialization.

b. Category
This field is used by server and client as a way to find out what the packet does. (Note that the name is category and not 'type', because in many programming languages, 'type' is a reserved word, even though it is the most appropriate for this field). Types of categories are MSG_NAME, MSG_TEXT, MSG_CHANNEL, MSG_INFO, MSG_SIG, MSG_QUIT. Definitions of these will be in the next subsection of this document. When a client or server receives a message, the category is the first field that is checked to find out what is to be done with the message.

c. Subtype
The subtype is like a secondary category. It is meant to tell what type of action is to be done with the category. Categories usually have subtypes that are unique to them. More information subtypes will be in the next subsection. In private messages, this field would contain the target client's username.

d. Status

This field is essentially a boolean field. Its possible types are SIG_SUCCESS, SIG_FAIL, SIG_REQUEST, SIG_INVALID, SIG_USED. Half of these values are for types of responses to name input from the client.

e. Channels

This field is a list of strings of channel names. This field is only checked when the category is of type MSG_TEXT, indicating that the message contains a user's chat text.

f. Content

This is usually the largest part of the message object. For MSG_TEXT type messages, this is where the user's text chat will go. For nearly all other categories, this field will contain the content of the query that was sent from the client to the server, such as a submitted username, or the name of a room to create. No matter the result of the query, the server would send back the same content to the client as a type of confirmation of what was submitted to the server.

g. TimeSent

This field is a string in HH:MM:SS format. It is generated automatically on message creation on the server or client, and is meant to be user-readable as a way to tell when the message was sent. It is not referred to by the server or client after message creation, and is only meant for the user.

## 2.3. OP Codes

Each message category and subtype include several OP codes (or, codes) that are meant to be flags that are interpreted by the server or client when a message is received. Each category code and its respective subtype codes are listed below with definitions.

Category: MSG_NAME

This code is meant to indicate that the message contains a submitted username of a client. In this type of message, there is no subtype (left as null), the content will be the username. The status can indirectly indicate who the message is coming from and what the decision about its validity is.

Possible statuses are:

SIG_REQUEST, meaning that this message is a query to the server asking if the username is good to use for the client. This status is only sent from a client.

SIG_SUCCESS, meaning that the username is valid, and not used by another client.

SIG_INVALID, meaning that the username does not meet the naming criteria.

SIG_USED, meaning that the username is already in use by another client

Category: MSG_TEXT

This code is meant to indicate that the message is simply for text communication. In this category only, is the channel field of the message used. The channels that the message is directed to will be listed. With this category, there is no subtype.

Category: MSG_CHANNEL

This category is meant to indicate that the message contains an action regarding a channel. When sent by a client, the status will be SIG_REQUEST, indicating that it is a query to the server. The server will respond with a status of SIG_SUCCESS or SIG_FAIL, depending on if the action was valid or invalid, for whatever reason. In this category, the name of the channel will be in the content field.

Possible subtypes are:

CHANNEL_CREATE, meaning that this request or response is about the creation of a room. This query would return as a failure if the room is already created.

CHANNEL_JOIN, meaning that the request or response is about a client joining a channel. This query would fail if the client is already in the channel, or if the channel does not exist.

CHANNEL_LEAVE, meaning that the request or response is about a client leaving a channel. This query would return a fail if the client was not in the channel, or if the channel does not exist.

CHANNEL_DELETE, meaning that the request or response is about a client wanting to delete a channel. This query would return a fail if the channel does not exist, there are clients in the channel, or if the channel is flagged as undeletable.

Category: MSG_INFO

This category is meant to indicate that the message contains a query or response for a client about a status in the server. The subtype indicates what type of information is sought, and the content in the responding message will be a string in sentence format about what the server says about that query. For the INFO_USERS subtype, the content field can optionally contain a channel name, meaning that the client is asking about a specific channel.

Possible subtypes are:

INFO_USERS, meaning that the client is asking about the list of client's usernames currently in the server. If the content field is not null, the server's response will be the list of client usernames currently in that specified channel. If a channel is not specified, this will always return successful, and will return a list of client usernames currently connected to the server. If a channel is specified and it does not exist, it will return as a failure.

INFO_CHANNELS, meaning that the client is asking for a list of channels currently in the server. The content field is ignored for this subtype, and will always return as successful.

Category: MSG_WHISPER

This category is meant to indicate that the message contains text directed to a single client. The subtype is the target client's username, and the content will be the text submitted by the source client. If the target client exists, the message will be directed to the target client with a status of SIG_SUCCESS, and if the target client does not exist, it will be sent back to the client with a status of SIG_FAIL. Even though private messages pass from client to server to client, this category is the only type of message that retains the original sender string. When the target client receives the private message, it will be as if it was sent from the original sender, even though it just came from the server.

Category: MSG_QUIT

This message is sent only from client to server. It is meant to indicate that the client is gracefully closing its application process. No response is sent from the server. Subtype, status, and content fields are inconsequential.

## 2.4. Usage and Examples

If one were to intercept a message object, one could see the exact purpose of the message, its source and target. Below are a few examples of messages with explanations.

```
Message:
    Sender: "scouter"
    Category: MSG_CHANNEL
    Subtype: CHANNEL_JOIN
    Status: SIG_REQUEST
    Channels:
    Content: "chitchat"
    TimeSent: 09:45:12
```

The above message was originally sent from the client with the username of "scouter". Its goal is to ask the server to join a channel with the name of "chatchat". It was sent from the client at 9:45:12am. Channels are meaningless in this message, so it is left empty. While the time field is not read by any users, it is still in the message as it is auto-created on message instantiation. A response from the server might look like the following:

```
 Message:
     Sender: "Server"
     Category: MSG_CHANNEL
     Subtype: CHANNEL_JOIN
     Status: SIG_SUCCESS
     Channels:
     Content: "chitchat"
     TimeSent: 09:45:13
```

A new message is created by the server and sent to the client. The category and subtype are unchanged, but the status is changed to SIG_SUCCESS, meaning that the request was accepted and processed. When the client receives this message, it will know that it is now in the "chitchat" channel, and the client will track this accordingly.

## 3. Client

The client is meant to be a simple process, only listening to the server and performing resulting actions, and sending query messages to the server.

### 3.1. Login

Upon launching of the client process, the user is asked for a hostname and port. If no hostname is submitted, the client will use localhost (127.0.0.1). If no port is submitted, it will use the default of 7778. If the client fails to connect to a server within some number of tries, the process will close. If a connection is successful, the client will send an introductory packet to the server, not using the proprietary message object of this protocol.

The client will then prompt the user for a username. When submitted, the client will send the server a message with the username. If the username is in use or not valid, the client will state so to the user, and the user will be prompted to submit a username again. Realistically, the only restrictions on usernames are that they do not contain spaces (ASCII 0x20), as user command input is dependent on spaces to create message categories. If the username that was submitted was valid, the client will be placed into the default channel (typically called the "Lobby"), and the server will send a message to the client stating that this channel joining was done. The client will then add the Lobby channel to its local list that tracks currently joined channels.

The client will then launch a second thread to listen for more messages from the server. The main thread will be listening for user input.

### 3.2. Listening

In this second thread, the client will be listening for server messages indefinitely. Upon receiving a message, the client will decode the byte object into a readable message. If the message is a MSG_TEXT, the message will be interpreted as a text chat from a client, and be shown to the user in the readable format:

```
<time> [Channel1]...[ChannelN] <source username>: <chat text>

09:45:13 [chitchat] scouter: hello!
```

If the message is a MSG_CHANNEL, it will determine the subtype. Any message of this category with a status of SIG_FAIL will state to the user that the room action failed. It would be up to the user to act on that failure. If the status is a SIG_SUCCESS, the client will perform actions to track internally to mirror what the server knows about the client. The client tracks what channel(s) the client is in, and what the active channel(s) are. Upon successful joining or leaving requests, local channel lists are updated accordingly.

If the message is of category MSG_INFO, the content of the message will be displayed to the user, similar to as if the server had sent a text chat message to the user. If it is of subtype INFO_USERS with an unknown channel name submission, the user will be informed that the channel name was invalid.

If the message received was a MSG_WHISPER, a readable string will be presented to the user in a similar format to the user. It will not be in the exact same format as a regular chat text, so it is not possible to deceive a user with a channel named "Whisper":

```
[Whisper] <time> <source username>: <chat text>

[Whisper] 09:45:13 scouter: hello!
```

No other message object would come from the server. However, if an EOF comes from the server, the listening thread will close, and prompt the entire process to close gracefully.

## 3.3. Chatting

In the main thread of the application after the listening thread is spawned, the user will be able to input characters into the client. If the entered text starts with a special character (namely "/"), this indicates that this text input will be a command (discussed in the next section). If the first character is not that special character, it is a regular text chat string. Upon submitting (pressing "enter"), the client will create a MSG_TEXT type message with the chat text as the content, and send the message to the server. The server will then broadcast the message to certain users. See Section 4: Servers on how this works.

## 3.4. Commands

The client is able to input several commands to perform various actions, some local to the client, and others that create messages for the server.

### 3.4.1 Channels

The "/channel" command has many uses and subtypes. Its usage is the following:

```
/channel <create OR join OR leave OR delete> <channel name>
```

No matter the (valid) input for this command, a MSG_CHANNEL message will be sent to the server, with the corresponding message subtype: CHANNEL_CREATE, CHANNEL_JOIN, CHANNEL_LEAVE or CHANNEL_DELETE. The channel name will be placed in the message content, and the message status will be a SIG_REQUEST.

Upon response to this sent query message, the client will receive a message regarding its success. If it is a CHANNEL_JOIN or CHANNEL_LEAVE request, the client's local list will be updated accordingly.

### 3.4.2 Channel Selecting

A client is able to talk in more than one channel per text chat input. This is done with the following commands:

```
/shoutset <channel 1> ... <channel N>
```

The shoutset command allows the client to send or "shout" text chat messages to multiple channels at the same time. When this command is inputted, the client will select a subset of channels that the client has joined, and apply them to the shoutset list. This command's operation is done client-side only; no messages are sent to the server. A client cannot add a channel to its shoutset list if it is not joined to the channel via "/channel join". If a client leaves a channel, upon response from the server, the client will remove that exitted channel from the shout set list.

```
/shout <text>
```

Using the shout command, the client will generate a message very similar to a regular chat message, but the room field in the message will be composed of the shoutset list that was established with the shoutset command. The server will then broadcast the chat message to the clients that are in each of the rooms that the client specified. After a shout is performed, the shoutset list is maintained, so there is no need to rebuild the list every time, allowing /shout to be used quickly in succession is needed.

Finally, for the default text chatting with only one channel at a time, the user can input the following command to switch focus:

```
/talk <channel>
```

When a channel is selected via /talk, the user's regular text chat will go to the selected channel. This command will only be able to select a channel that the client is joined to via "/channel join". If a client's talk channel is exitted, the /talk command will need to be used again to select a new default channel.

### 3.4.3 Private Messages

It is possible to send messages directly to one client, ignoring channels.

```
/whisper <username> <text>
```

When the command is submitted, the server will respond back to the sender notifying the user that the message could not be delivered, if that is the case. If the message was successfully delivered, both the sender and destination client will receive the message and display the text content in chat. The target client does not need to be in any shared channels, they just need to be in the server at the same time.

### 3.4.4 Server Information

One general command allows querying of the server for information for the user.

```
/info <users> [channel name]
/info <channels>
```

For "/info users", the client will send a MSG_INFO message to the server to query the list of users in the server. Optionally, the user can input a channel name. No matter the valid input, the server will send back a message with content in a user-readable format, and the content will be displayed for the user. If the user did not specify a channel, the message will contain a list of all of the clients in the server. If the user did specify a channel, the list returned will contain only the list of clients in that channel. No matter the valid input's response, the content's string will also contain the number of users in the list that was returned.

For "/info channels", the client will send a message to the server asking for the channel list of the server. The response will always be a success and the content of the response message will be a user-readable string of the channel names.

#### 3.4.5 Quitting

Graceful exiting of the client application process is done via the "/quit" command. When this command is inputted, the client will send a MSG_QUIT message to the server and subsequently close the socket, and close all threads of the application. Using a keyboard shortcut such as Control-C will also perform the actions of the "/quit" command.

#### 3.4.6 Aliases

Due to the length of the words used in each command, each command includes aliases allowing for faster input. For each command, one can enter "/<first character of leading command>", such as "/t" for "/talk" or "/ss" for "/shoutset". The command parameters will remain unaffected. The command and their aliases perform the same actions.

## 4. The Server

The server application performs most of the processing of all of the client's queries. The server then sends back answers to queries to the appropriate clients. The server also maintains a list of all clients and rooms.

### 4.1. Initialization

Upon server instantiation, the server requires a hostname, port and display name. The hostname and port are used to create the server socket, and listening port for acquiring clients. If a port is not specified, the default port 7778 will be used. The server will then initialize a hashtable of channels and a hashtable of clients. The server will then start listening for connecting clients, and the first channel will be created called the "Lobby", and be added to the channel list.

### 4.2. Listening

Listening for clients is performed on a seperate thread from the main thread.

#### 4.2.1. Listening for New Clients

When a client connects to the server, a new thread will be created for them. The total number of threads running on the server will be 2+N, where N is the number of clients.

Upon client connection, the dedicated client thread will be listening for a message with the MSG_NAME category. The content field in this message will contain the username for the client. Upon receiving the message, the server will check that the username is valid, meaning that the name contains alphanumeric characters only and no spaces, AND that it is not already in use. The server will then send a MSG_NAME message back to the user with the response status of SIG_SUCCESS, SIG_USED, SIG_INVALID. If the response given back to the client is SIG_INVALID or SIG_USED, the server will continue to listen for a username.

When a username is valid, the SIG_SUCCESS message will be sent to the client, and the server will create a new object to house the client's information (primarily its socket information, active channels, and username). This new client object will be added to the client hashtable of the server as a value, and the username as a key. This new client will also be added to the default channel, Lobby. This process is called "Registration".

#### 4.2.2. Listening for Messages from Existing Clients

When a client is registered, the dedicated client thread will be listening for messages from that client.

### 4.3. Handling Client Messages

Upon receiving a message from a client, the server will first check the category of the message.

#### 4.3.1. Text Chat

If there is a message of category MSG_TEXT, it is a generic text chat. The server will then read the channel field of the message, and then send the message to each client in the specified channels. Only the status field is changed to SIG_SUCCESS in the message. No other field is modified in its copy that is sent to each client.

#### 4.3.2. Whispers

If the message is of category MSG_WHISPER, the message will be sent directly to the client specified by the username in the subtype field, and back at the sender as well. No fields are modified, and only the category and subtype fields are accessed when the username is valid. If the username is not valid, a new message is sent from the server to the origin client saying in user-readable text that the message could not be delivered.

### 4.4. Non-Text Commands

Handling commands is the bulk of the server's logic.

#### 4.4.1. Channel Management

When a message of category MSG_CHANNEL is received, the server will then read the message subtype, then the content field for the channel name. No matter the channel action requested, if the channel name is invalid (does not exist, or if it already exists for the channel creation command), then a failure message is sent back to the client. If the channel name is found to be valid, the appropriate action is performed.

In the case of a client joining a channel, the client object local to the server will be added to the channel's client list, and the channel will be added to the client's channel list.

In the case of a client leaving a channel, the client object local to the server will have the channel removed from its channel list, and the channel will remove the client from its client list.

Both of these channel joining and leaving functions are 'atomic', so any joining or leaving actions will result in correct modification of the channel's client list, and the client's channel list.

In the case of channel creation, the message's content field will be read to get the name of the channel to be created, and the new channel will be added to the server's local channel hashtable. The channel object will be the value, and the name of the channel will be the key. A message will then be sent to the client that the operation was successful.

For room deletion, the equivalent process will take place; the channel will be removed from the local hashtable, and the client will be notified of successful deletion. If any clients are still in the channel that is being deleted, the action will fail and the client will be notified that it cannot be deleted.

### 4.4.2. Client Quitting

In the event a client sends a MSG_QUIT message, the server will access the local client's channel list, and remove the client from each of the channel object's client lists. The client will then be removed from the client hashtable. When this message is received from the client, it is assumed that the client is taking its own steps to stop the client application process.

When the client sends an EOF packet to the server, the server will end the dedicated thread function of listening to that client, then join with the main thread. This entire process is called "unregistering".

### 4.4.3. Table Information

In the event a client sends a message of category MSG_INFO, the server will be looking at the current state of its client list or channel list.

If the subtype is INFO_CHANNELS, the local channel hashtable's keys will be read, and a new message will be sent to the client with a user-readable list of channel names in the content field.

If the subtype is INFO_USERS, the content field will be checked. If the content field is blank, the client is asking for a list of clients connected to the server. The client hashtable's keys will be read, and written to a message that will be sent to the inquiring client. If the client field has a valid channel name, the specified channel will have its user list read, and the server will create a message containing the list of client usernames in that channel, and send it to the inquiring client.

## 4.5. Server-side commands

After the client listening thread is created, the main thread is still running. At this point, a user/server administrator is able to input commands into the terminal of the server. The three commands are "channels", "users" and "quit". The channel command will print to console the list of all channels, and all of the clients within each channel. The users command will return a list of all connected client's usernames and the channels that they are in. The quit command will attempt to close all connected client's socket connections and close gracefully. Clients are informed that the server is closing.

## 5. Channels

### 5.1 Channel Structure

Within the server, a hashtable of channels is kept. This hashtable is created before any clients can join the server. These channels are represented as a channel object. This object contains the following fields:

a. Name
The user-readable name of the channel. It is the primary identifier of the channel, and the key for the hashtable for a channel.

b. Created by
This is a string of the client's username that created the channel. Used for informational purposes only, and filled in automatically upon channel creation.

c. Create time
The time that the channel was created. In string format listing full date and time.

d. CanBeDeleted
A boolean flag stating if the channel can be deleted. The only channel that cannot be deleted (by default) is the Lobby channel. Any other client-created channel can be deleted.

e. CurrentUsers
A list of client usernames that are currently in the channel

There are two functions within a channel:

a. joinChannel
An encapsulated way of adding a user to the CurrentUsers list of a channel

b. leaveChannel
An encapsulated way of iterating over the CurrentUsers list and removing a client from the list

## 6. Server-side Client Structure

A hashtable of clients is kept server-side as a way to track what channels a client is in, and what socket each client belongs to. When a client connects to the server, a new object is created for the client, and the username, and network/socket information is kept in this object, along with channel information. A server-sider client object contains the following fields:

a. address
The address that the client is connected from

b. port
The port that the server-side socket is connected to for this client

c. Username
The name that the client picked for itself. In the hash table storing the clients, this is the key, and the value is this entire client object

d. connectTime
A string of the full date and time that the client connected to the server

e. socket
The full socket object of the client

f. channels
A list of channel names that the client is connected to. Only contains the names as strings of the channels

## 7. Error Handling

The message status field is key to addressing errors. In general, it is the client that receives the failures and successes, and the server that distributes the failures and successes. For server-side errors such as users not existing in lists, or small, non-fatal issues, a failure signal is sent to the client that made the query to inform them that the action failed. In larger issues like a client disconnecting abruptly, the server will unregister them. This process is described in 4.4.2. Quitting. In that event, the server will continue to function normally.

If it is the server that abruptly halts, each connected client will be notified via the socket that the connection was closed, the client will then end threads gracefully, and prompt the user to close the application process manually.

If a client fails to connect to the server in the middle of a session, the client will close gracefully. It will always attempt to inform the server that the client is closing, then close the socket and end all threads.

## 8. Conclusion & Future Work

Many small enhancements can be done to this protocol. Firstly, the private messages pass through the server. An alternative to this is that the client could ask the server for the socket information of the target user, and the server could supply that. Then the client could connect directly with that user to deliver the message.

Additionally, whether the client has private messages go through the server or directly to the target client, the messages could be encrypted in some way. Currently, they are in plain text.

Another possible improvement is with the 'timeSent' field of messages. Currently they are meant to be displayed as-is to recipient clients in chat, and encoded as a string in format HH:MM:SS. An alternative is to have the timeSent field be milliseconds from some date/time, and it would be up to the client to interpret that. This would allow users to see the correct times if users are in different time-zones. It would also allow the showing of dates, in instances where an IRC might be running over the course of days with slow chat traffic.

## 9. Security Considerations

As mentioned in the previous section, messages sent using this system have no protection against accessing or modification. The server sees all messages that are sent. Private messaging may be easily intercepted by a 3rd party via network traffic inspection. Users wishing to use this system for secure communication should use/implement their own user-to-user encryption protocol.

However, it is worth noting that IP addresses of clients are hidden from one another. For some, having an IP address exposed to others can be unsettling. In this protocol, the server maintains all IP addresses, but each client knows only usernames, and there is no way for a client to get another client's IP address without the server sending that information itself.

This protocol does not account for modified versions of the client. In the version described in this document, it would be possible to have a malicious client connect and send any type of packet to the server. A possible improvement to be made would be some type of checksum that the client sends to the server upon connection. If the checksum is valid, the client has a verified/clean version of the client and can continue to connect to the server, otherwise the server could terminate the connection to the modified client immediately.

## 10. IANA Considerations

None

## 11. Normative References

None

## 12. Acknowledgments

Python's pickle library for data serialization. This library makes the message structure possible.
https://docs.python.org/3/library/pickle.html

IRC Project specification provided via 594 Canvas page. That document provided a structure for this document.
draft-irc-pdx-cs594-00.txt

Maxwell Oakes
Portland State University Computer Science
1825 SW Broadway, Portland, OR 97201

Email: maxoakes@pdx.edu