CS594 Haritha Munagala, Susham Yerabolu

Internet Draft Portland State University

Intended Status:IRC Project Specification June 8, 2018

Expires: June 2018

# Internet Relay Chat Project

Abstract

This document describes the communication protocol for an IRC client and server for the Internetworking Protocol Spring 2018 class at Portland State University.

Table of Contents

## 1. Introduction

This specification describes the Internet Relay Chat (IRC) protocol where clients can communicate with each other. The system will utilize a server where clients will connect to and chat with other connected clients.

## 2. Background Information

This protocol is built upon the TCP/IP layer (RFC 793 and 791 respectively) with the server listening to incoming messages on port 10005. The server will act as the central host for clients to gather and chat among themselves. The connection between the client and server will be persistent and asynchronous, that is, the client and server can send messages to one another at any time. All messages sent between the server and client are encrypted using AES 128.

## 3. Message Infrastructure

All messages sent between the server and client will be in JSON format. Every JSON message sent from the client to the server will contain a “command” key telling the server what the client wants to do. Based on the command, there will be other keys in the JSON object that the particular command needs in order to function properly. These keys are discussed in detailed in the next section.

# 4. Client Actions/Messages

## 4.1 Connection to server

Before a client can exchange messages with other users, they first establish a connection with the server and inform the server their username.

The server then associates the client username with their connection. The client should only send the connection request once; any subsequent messages will be ignored by the server.

### 4.1.1 JSON Command Format

```
{
    "command": "NICKNAME“
    "name": "username"
}
```

## 4.2 Listing Rooms

Client performs this action by sending a request to the server to retrieve all rooms which have at least one user in them.

In response, the server then returns the list of all current rooms.

### 4.2.1 JSON Command Format

```
{
    "command": "LR”
}
```

### 4.3 Creating a Room

Client sends a request with name of the chat room to create. The server checks to make sure that a room with the provided name does not exist. If it doesn’t, then the server creates the room and adds the requesting client to it. If a room with the name exists, the server informs the requesting client and to try again with a different name.

#### 4.3.1 JSON Command Format

```
{
    "command": "CR",
    "roomName": "ROOMNAME"
}
```

### 4.4 Joining a Room

Client sends a request with the name of the room to join. If the room exists, the client is added to its user list and then the server sends a message to all other users in the room about the new member who joined. If the room does not exist, then the server informs the client about it. The server will not let a client to join the same room twice.

#### 4.4.1 JSON Command Format

```
{
    "command": "JR",
    "roomName": "ROOMNAME"
}
```

### 4.5 Leaving a Room

Client sends a request with the name of the room they want to leave.The server then removes the client from the user list of the specified room and sends a notification message to all users in that room that the user list has changed and provides the updated user list. Also, the server ignores any leave requests for rooms where the client is not a member of. If the last client in the room leaves, then the server deletes the room.

#### 4.5.1 JSON Command Format

```
{
    "command": "LER",
    "roomName": "ROOMNAME"
}
```

### 4.6 Listing users in a room

Client sends request to server requesting the list of users in the specific room. The server sends a copy of the user list for that room to the client. If the room does not exist or the client is not part of that room then, the server sends a message notifying the client that the action cannot be performed.

#### 4.6.1 JSON Command Format

```
{
    "command": "LRC",
    "roomName": "ROOMNAME"
}
```

### 4.7 Sending Messages

#### 4.7.1 Room Message

The client sends a message to the server indicating which room it is intended for. The server checks to make sure the room exists and sends the message to all other users in the room. The server then sends the requesting client that their message has been sent.

##### 4.7.1.1 JSON Command Format

```
{
   "command": "MR",
   "roomName": "ROOMNAME",
   “message”: “MESSAGE”
}
```

#### 4.7.2 Private Message

The client sends a message to the server indicating the intended recipient of the message. The server finds the client with the matching username and forwards the message to the target client.

##### 4.7.2.1 JSON Command Format

```
{
   "command": "PM",
   "target": "TARGET",
   “message”: “MESSAGE”
}
```

### 4.8 Listing all clients connected

The client sends a message to the server asking for a list of all active users on the server. The server sends the list of clients usernames that are currently connected.

#### 4.8.1 JSON Command Format

```
{
    "command": "LC"
}
```

### 4.9 Sending Files

#### 4.9.1 Sending files to room

The client sends the file to the server in chunks indicating which room to share the file with. Then, the server sends the file to all others users that are part of that room.

##### 4.9.1.1 JSON Command Format

```
{
   "command": "SFR",
   "target": "TARGET",
   “file_name”: “FILENAME”,
   “file_size”: “FILESIZE”
}
```

#### 4.9.2 Sending files in private message

The client sends the file to the server, indicating with whom the file should be shared with. The server checks to make sure that the intended recipient is connected to the server and if so, starts sharing the file the user.

#### 4.9.2.1 JSON Command Format

```
{
   "command": "SFP",
   "target": "TARGET",
   “file_name”: “FILENAME”,
   “file_size”: “FILESIZE”
}
```

## 5. Server Actions/Messages

### 5.1 Notifications

Server sends a notification to the client for certain scenarios. For example, when the client sends a message to a room, the server sends a notification to the client indicating the their message was sent to the room or the server encountered some error. The following JSON format illustrates the JSON that the server sends to the client regarding the particular example:

```
{
  "message": "<SERVER> Message sent to room"
}
```

### 5.2 Forwarding Messages

#### 5.2.1 Forwarding Messages to Room

Server forwards a message to all other clients in a room when a client sends a message to the room.

##### 5.2.1.1 JSON format

```
{
   "message": "<SERVER> $SENDER in $ROOM says: $MSG"
}
```

#### 5.2.2 Forwarding Messages to Client

In the case of private messages, the server sends a forwards the private message to the recipient.

##### 5.2.2.1 JSON format

```
{
    "message": "<SERVER> $SENDER sent a message to you: $MSG"
}
```

## 6. Error Handling

### 6.1 Client Crash/Disconnect

The server manages all clients that are connected to it. If the client disconnects or crashes, the server will remove the client from all rooms that the client was in and then disconnecting the client from the server.

### 6.2 Server Crash/Disconnect

The client detects if the server crashes, displays the notification “Server down” and exits.

### 6.3 Invalid Command/Arguments

The client checks the validity of the arguments sent in a command and asks to retry in case of failure. Also, it warns about the invalid about, if any entered.

### 6.4 Other Unforeseen Errors

There may be errors that could arise that were not thought of ahead. Any unexpected errors or bugs should be reported to the RFC authors so that a fix can be implemented as soon as possible.

## 7. Security Considerations

- The protocol does not support private rooms. Any client that is connected to the server can see all active rooms and join any they choose to.

- The protocol provides end-to-end encryption using AES-128 encryption algorithm when sending and receiving messages. The server and client have a pre-shared key which is used to encrypt the data before sending it to the recipient. The Initialization Vector (IV), which is required by AES, is generated randomly and sent after adding to the encrypted data which is then extracted at the receiver's end. The IV has a fixed block size of 16 bytes and is maintained by adding and removing padding bytes at sender and receiver’s end respectively.

  - Encryption

    It takes three parameters to encrypt the data:

    Key - Private key shared between Client and Server
    Mode - AES Mode (We used CFB)

    IV - 16 byte random Initialization Vector

    The padding is added to the data if it’s not 16 bytes and then encrypts the padded data. The random generated IV is prepended to the data before sending.

  - Decryption

    It takes three parameters to decrypt the data:

    Key - Private key shared between Client and Server
    Mode - AES Mode (We used CFB)

    IV - 16 byte random Initialization Vector

The IV is extracted from the data (the first 16 bytes of the encrypted message). The data is then decrypted using the private key with the padding removed at the end, if any.