Alex Springer

Portland State University

December 6, 2019

Internet Relay Chat RFC Class Project

Alex-Springer-irc-pdx-cs494.txt

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79. This document may not be modified, and derivative works of it may not be created, except to publish it as an RFC and to translate it into languages other than English. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts. 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." The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html This Internet-Draft will expire on Fail June, 2020.

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.

2. Basic Information

All communication described in this protocol takes place over TCP/IP in the form of sockets, with the server listening for connections on port 4001. 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.

3. Message Format Semantics

```
          <form onSubmit={this.handleSubmit}>
            <div class="input-group">
              <input
                class="form-control"
                type="text"
                id="message-input"
                name="message-input"
                placeholder="Enter a message to chat."
```

```
                />
                <div class="input-group-append">
                  <button type="submit">Send</button>
                </div
```

An input bar is provided to the user on the web page where they can type in a string that they wish to send. This string can be any characters, and there is no constraint on the length of the message. Due to no constraints being placed on the strings users can enter, there are no errors to check for.

4. Username Semantics

```
    const name = prompt("Enter your name please:");
```

The semantics for user names are the same as for messages. The user is prompted with an input box where they can enter any string of characters available to them. Due to the length, and the type of characters having no constraints, there are no errors to check for.

5. Core Features

5.1. First message sent to the server

```
io.on("connection", socket => {
```

5.1.1 Usage

Whenever a unique client connection occurs it is broadcasted to the server. This is automatically done and is logged to the console on the server.

5.2 Listing Rooms

Server:

```
socket.emit("active-rooms", roomNameList);
```

Client:

```
this.state.socket.on("active-rooms", roomNameList => {
  let i = 0;
  for (i = 0; i < roomNameList.length; i++) {
    this.appendRoom(roomNameList[i]);
  }
});
```

5.2.1 Usage

In the event of a new connection, the server broadcasts a list of all rooms created by users prior to the new clients connect. Obviously, on the very first connection to the server this list of rooms will be empty and thus no rooms will be shown on the client.

5.3 Creating Rooms

Client:

```
handleSubmit(event) {
  event.preventDefault();
```

```
  const roomInput = document.getElementById("room-input");
  var room = roomInput.value;
  if (!document.getElementById(room)) {
    this.appendRoom(room);
    this.state.socket.emit("send-room", room);
  }
  roomInput.value = "";
}
```

5.3.1 Usage

To create a room, the user is provided an input bar on the web page where they type a string of any length. Once that chat room is created, the name of that room is then sent to the server to be stored in an object that keeps track of all rooms and users that are in that room. Note, a user cannot create a room that has the same name as another room. If the user attempts to create a room that already exists, the room name will not be sent to the server and the link to the room will not be appended to the page.

5.3.2 Response

Server:

```
socket.on("send-room", room => {
  rooms[room] = { users: {} };
  socket.broadcast.emit("new-room", room);
});
```

Client:

```
    this.state.socket.on("new-room", room => {
      this.appendRoom(room);
    });
```

Once the server receives the name of a newly created, it adds the room name to the list of active rooms and initializes a list of active users, which is empty on the creation of a new room. Then the name of the new room is broadcast to all clients except for the client that created the room. Once the room name is received by clients, the link to the room is appended to the webpage.

5.4 Listing Users

Server:

```
socket.emit("active-users", rooms);
```

Client:

```
  this.state.socket.on("active-users", rooms => {
      var currentRooms = rooms[this.state.room].users;
      var singleRoom = Object.values(currentRooms);
      var activeUsers = Object.values(singleRoom);
      let i = 0;
      for (i = 0; i < activeUsers.length; i++) {
        this.appendUser(activeUsers[i]);
```

```
    }
});
```

5.4.1 Usage

Upon connecting to a chat room, the server broadcasts the list of all rooms, and the users inside them. However, when displaying active users, it must only be the active users in the current room. To achieve this, each chat room has a unique URL in the following form: https://localhost:3000/:roomName. Thus the current room name can be retrieved from the URL and be used as the key to retrieve the users in the current room. Once the list of active users is retrieved, the list is iterated through and each user in the list is displayed on the web page.

5.5 New User

Client:

```
const name = prompt("Enter your name please:");
this.setState({ name: name });
this.appendMessage("You joined!");
this.appendUser(name);
this.state.socket.emit("new-user", { name: name, room: room });
```

5.5.1 Usage

Upon entering a room, the client is prompted to enter a name. Entering a name will give the client confirmation they joined the room by appending “You joined!” to the web page. When the

name of the user is broadcast to the server, it is done so with the room name as well. This is to tell the server which room the new user joined, so it can store the user in the appropriate room.

5.5.2 Response

Server:

```
socket.on("new-user", data => {
  socket.join(data.room);
  rooms[data.room].users[socket.id] = data.name;
  socket.to(data.room).broadcast.emit("user-connected", data.name);
});
```

Client:

```
  this.state.socket.on("user-connected", name => {
    this.appendMessage(`${name} connected.`);
    this.appendUser(name);
  });
```

When the server receives the new user broadcast from the client, the server tells that socket to join a channel room, where room is the name of the room the new user joined. Once the socket has joined the room channel, the new user's name is then broadcast specifically to other sockets in the same channel. When the new user is received by other users in the room, a notification that they joined is appended to the web page.

5.6 Sending Messages

```
handleSubmit(event) {
  event.preventDefault();
  const messageInput = document.getElementById("message-input");
  var message = messageInput.value;
  this.appendMessage(`You: ${message}`);
  messageInput.value = "";
  this.state.socket.emit("send-message", {
    room: this.state.room,
    message: message
  });
}
```

5.6.1 Usage

To send messages the user types in a string of any length into the input bar provided at the bottom of the web page. When the user submits the message, they are notified that their message was sent by appending “You: {whatever message they typed} “ to the web page. Just like when a new user joins, when a message is broadcasted to the server, it is done so with not only the message but also the room name.

5.6.2 Response

Server:

```
socket.on("send-message", data => {
  socket
```

```
    .to(data.room)
    .broadcast.emit("chat-message", {
      name: rooms[data.room].users[socket.id],
      message: data.message
    });
});
```

Client:

```
  this.state.socket.on("chat-message", data => {
    this.appendMessage(`${data.name}: ${data.message}`);
  });
```

Once the server receives the broadcast that a new message has been sent, it then uses the room name sent by the client to send a message to the other users in that room. Once again, the server broadcasts to every user in the specified room except for the client that sent the message because their message can be appended to the web page without help from the server.

5.7 Disconnecting

Server:

```
socket.on("disconnect", () => {
  console.log("user disconnected");
  getRooms(socket).forEach(room => {
    socket
      .to(room)
```

```
        .broadcast.emit("user-disconnected", rooms[room].users[socket.id]);
      delete rooms[room].users[socket.id];
    });
  });
});

function getRooms(socket) {
  return Object.entries(rooms).reduce((names, [name, room]) => {
    if (room.users[socket.id] != null) names.push(name);
    return names;
  }, []);
}
```

5.7.1 Usage

A heartbeat mechanism is implemented at the Engine.IO level, allowing both the server and the client to know when the other one is not responding anymore. That functionality is achieved with timers set on both the server and the client, with timeout values (the pingInterval and pingTimeout parameters) shared during the connection handshake. Those timers require any subsequent client calls to be directed to the same server, hence the sticky-session requirement when using multiples nodes. Because disconnect detection is done automatically we are not able to send the room name and user name from the client. However since each user is stored in a key value pair of (socket.id, username), and we have access to the socketid, we must search through

the user list of each room looking for the user the corresponds to the socket.id. Once the room and user are found, the server broadcasts the name of the client to the room it is leaving.

5.7.2 Response

Client:

```
this.state.socket.on("user-disconnected", name => {

  this.appendMessage(`${name} disconnected.`);

  this.removeUser(name);

});
```

As shown above, once the client has received the broadcast that a user has disconnected, it appends a message to the web page and then removes the user from the list of active users also displayed on the web page.

6. Error Handling

Both server and client MUST detect when the socket connection linking them is terminated by keeping track of the heartbeat messages. 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 and, Unless instructed otherwise, a disconnected client will try to reconnect forever, until the server is available again.

7. Conclusion

This specification provides a generic message passing framework for multiple clients to communicate with each other via a central forwarding server. Without any modifications to this specification, it is possible for clients to devise their own protocols that rely on the text-passing system described here. For example, transfer of arbitrary binary data can be achieved through transcoding to base64. Such infrastructure could be used to transfer arbitrarily large files, or to establish secure connections using cryptographic transport protocols such as Transport Layer Security (TLS).

8. Security Considerations

Messages sent using this system have no protection against inspection, tampering or outright forgery. The server sees all messages that are sent through the use of this service. Additionally, since there are no constraints on the usernames and messages clients can enter it is possible that this application is susceptible to malicious attacks in the form of buffer overflow. Users wishing for a system that is not susceptible to such attacks should place constraints on the length and content of the strings users enter for messages and usernames. Users wishing to use this system for secure communication should use/implement their own user-to-user encryption protocol.