CS594 Portland State University
RFC Draft November 28th, 2022
Intended Status: IRC Class Project Specification
Expires: January 1, 2023

Internet Relay Chat Class Project

Status of this Memo

Internet-Drafts (IDs) are living documents in service of the Internet Engineering Task Force (IETF), its forums, and its groups. Other working documents that serve as Internet-Drafts may also be distributed.

IDs are valid for a maximum of six months ad may be updated, replaced, or deprecate by updated versions asynchronously. Citing IDs as anything else than “work in progress” is discouraged.

The present list of IDs can be accessed at http://www.ietf.org/ietf/lid-abstracts.html.

The present list of ID shadow directories are available at http://www.ietf.org/shardow.html.

Abstract

This memo describes the communication protocol for an IRC-style client/server messaging platform for the Internet-working protocols class at Portland State University.

Table of Contents

1. Introduction

This specification describes a IRC protocol that enables users to communicate asynchronously. A central server relays messages to other users on the platform.

A user can join rooms, a room is comprised of a set of users, and all users can view the set of messages in each room which they are apart of.

Users can also send messages directly to each user on the platform.

2. Basic Information

Platform communication enabled over TCP/IP protocols with the server listening for connections on port 3000. All relays between clients/server occur asynchronously. This connection is subject to termination for any reason, which an message may or may not be sent in such an event. Server may additionally limit a set number of users and rooms if the implementation calls for such a specification. Corresponding error codes are implementation dependent.

3. Message Infrastructure

3.1 Generic Message Format

```
3   const formatMsg = (username, messageText) => {
4     const messageObject = {
5       username: username,
6       messageText: messageText,
7       time: moment().format('h:mm a'),
8     };
9     return messageObject;
10  };
```

3.1.1 Field Definitions:

- messageObject – metadata of message contents
  - username – sender
  - messageText – Data being sent
  - time – Timestamp formatted in hour, minute.

Client and server validate the content of the messageObject else an error is returned. The infrastructure handles the checksum / data encoding of the datagram being passed from client to server.

3.2 Error Messages

Errors are handled within a package handler in the Socket.IO infrastructure package-lock.json file. A component emitter handles appropriate error.

```
2031    "send": {
2032      "version": "0.18.0",
2033      "resolved": "https://registry.npmjs.org/send/-/send-0.18.0.tgz",
2034      "integrity": "sha512-qqWzuOjSFOuqPjFe4NOsMLafToQQwBSOEpS+FwEt3A2V3vKubTquT3vmLTQpFgMXp8AlFWFuP1qKaJZOtPpVXg==",
2035      "requires": {
2036        "debug": "2.6.9",
2037        "depd": "2.0.0",
2038        "destroy": "1.2.0",
2039        "encodeurl": "~1.0.2",
2040        "escape-html": "~1.0.3",
2041        "etag": "~1.8.1",
2042        "fresh": "0.5.2",
2043        "http-errors": "2.0.0",
2044        "mime": "1.6.0",
2045        "ms": "2.1.3",
2046        "on-finished": "2.4.1",
2047        "range-parser": "~1.2.1",
2048        "statuses": "2.0.1"
```

3.2.1 Usage

Error sent by client or server before closing of socket connection.

4. Label Semantics

Users and rooms involve creating labels. Each are validated within SocketIO infrastructure inputted by user.

5. Joining Room Messages

```
io.on('connection', (socket) => {
  socket.on('joinRoom', ({ username, room }) => {
    const client = joinUser(socket.id, username, room);
    socket.join(client.room);

    socket.emit('message', formatMsg(APP_NAME, 'Welcome to Real-Chat-App!'));
    socket.broadcast
      .to(client.room)
      .emit(
        'message',
        formatMsg(
          APP_NAME,
          `A new user (${client.username}) has joined the chat, the chat just got more real!`
        )
      );

    io.to(client.room).emit('usersInRoom', {
      room: client.room,
      roomUsers: getUsersInRoom(client.room),
    });
  });
```

5.1 Usage

Upon joining a room, a message is formatted welcoming user to the room. This is sent once and never again. Another message is sent from the server to associated clients that a new user, with their username, to participants in the room. This occurs once for each user that joins the room.

5.2 Leaving Rooms

```
socket.on('disconnect', () => {
  const client = leaveUser(socket.id);
  if (client) {
    io.to(client.room).emit(
      'message',
      formatMsg(APP_NAME, `A user (${client.username}) has left the chat!`)
    );

    io.to(client.room).emit('usersInRoom', {
      room: client.room,
      roomUsers: getUsersInRoom(client.room),
    });
  }
});
```

5.2.1 Usage

Application sends notification to remaining users in room that a user has left. Server removes the client from the specified room and sends a single notificatio to all members.

5.3 Create Room

```
app.post('/add-room-to-index', (req, res) => {
  room = req.body.indexCreateRoom;
  indexPageRooms.unshift(room);
  console.log(indexPageRooms);
  res.redirect('/');
});
```

5.3.1 Usage

Rooms are created by users via a indexCreateRoom Method. User is than redirected to the room. SocketIO handles the label creation semantics.

5.3.2 Listing Rooms

A log of existing indexed rooms is outputted for all the users.

5.3.3 Response

Response is redirected for the user in the event that the room has been created.

5.4 Field Definitions

5.4.1 Usage

Target name – name of receiver of message. Can be a user or a room.
Message – data being relayed from sender to receiver.

There is no specification on message length implemented by programmers. Specification for message content handled by SocketIO. Any rule that is violated is terminated by server.

```
//routes for express

app.get('/', (req, res) => {
  res.render('index', { indexPageRooms });
});

app.post('/add-room-to-index', (req, res) => {
  room = req.body.indexCreateRoom;
  indexPageRooms.unshift(room);
  console.log(indexPageRooms);
  res.redirect('/');
});

app.get('/converse', (req, res) => {
  res.render('converse');
});

app.post('/converse', (req, res) => {
  const user = req.body.indexUsername;
  const room = req.body.indexSelectRoom;
  const indexInfo = { room: room, user: user };
  res.render('converse', { indexInfo });
});
```

6. Server Messages

```
server.listen(PORT, () => {
  console.log(`Server running on port: ${PORT}`);
});
```

6.1 Usage

Server messages relayed when server indicates it is running and listening on a port. When message is conveyed, connection is made and able to send / receive messages.

6.1.2 Field Definitions

```
const getUsersInRoom = (room) => {
  return usersArray.filter((user) => user.room === room);
};
```

- Identifier – Contains names of the rooms a user belongs to. Label Semantics handled by SocketIO
- Users – Array of users on platform

7.2 Forwarding Messages to Clients

```
  io.to(client.room).emit('usersInRoom', {
    room: client.room,
    roomUsers: getUsersInRoom(client.room),
  });
});

socket.on('userMessage', (messageToSend) => {
  const client = getCurrUser(socket.id);
  io.to(client.room).emit(
    'message',
    formatMsg(`${client.username}`, messageToSend)
  );
});
```

7.2.1 Usage

Messages sent by a user asynchronously get forwarded to the correct socket id for all users in a room. This message is only sent once and viewed by the list of users associated with a particular room id identified via it’s corresponding socket id handled by the node modules. Each message is formatted and associated with username of the sender.

8. Error Handling

Imported node modules have standard error handling packages programmer can use to capture and recored error logs to output to console.

```
function createError () {
  // so much arity going on ~_~
  var err
  var msg
  var status = 500
  var props = {}
  for (var i = 0; i < arguments.length; i++) {
    var arg = arguments[i]
    var type = typeof arg
    if (type === 'object' && arg instanceof Error) {
      err = arg
      status = err.status || err.statusCode || status
    } else if (type === 'number' && i === 0) {
      status = arg
    } else if (type === 'string') {
      msg = arg
    } else if (type === 'object') {
      props = arg
    } else {
      throw new TypeError('argument #' + (i + 1) + ' unsupported type ' + type)
    }
  }
```

8.1 Usage

This is to catch any unsupported type operations for sending data to rooms that have been terminated, users who don’t exist, or other invalid operations allowable by the client/server.

9. Conclusion

This outline provides a generic overview framework for allowing multiple users to communicate asynchronously via a central server. These protocols are not specific to this framework and other clietns can devise their own protocols that rely on the system described here. Encoding is possible for any given room with access to the proper key if such a specification or requirement is necessary. File sharing too is possible given the right FTP implementation for basic text files.

10. Security Concerns

Using an open source framework always has some level of 0 day concerns and additioanlly encumpant on the programmer to take precaution when scaling their application. There is no guarantee that information being passed is always secure.

11. IANA Considerations

None

12. Acknowledgements

This document was formatted based on the provided RFC Sample in the CS594 Course.