Colin McCoy
CS 494
Internet Draft
Intended Status: IRC Class Project Specification
Expires: June 2019

Internet Relay Chat Class Project
draft-irc-pdx-cs494-00.pdf

Status of this Memo

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.

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.

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.

Basic Information

All communication described in this protocol takes place over TCP/IP, with the server listening for connections on port 2000. 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. A client MAY choose to disconnect from the server at any time, when it does so it SHOULD send a control message to the server indicating a quit.

Message Infrastructure

All messages are passed as a bitstream, in chunks no larger than 1024 bytes. Both the client and server SHOULD break larger messages into multiple chunks no larger than 1024 bytes before sending them. Messages broken up in this way MAY be interpreted by both client and server as separate messages. Client and server SHOULD decode incoming messages into the UTF-8 format, and encode outgoing messages, also in UTF-8.

Control Messages

The special character '/' is to be reserved for control messages, any user input beginning with this character SHOULD be treated as a control message by the client and SHOULD receive special processing before being sent to the server. Depending on the content of the message, the client MAY send it directly to the server, MAY perform additional actions before or after sending it to the server, or MAY perform actions without sending the message to the server. The server SHOULD interpret messages beginning with '/' as control messages, and take the appropriate action.

The following control messages SHOULD be recognized and handled appropriately by client and server. Additional messages MAY be defined and handled. Messages beginning with '/' but not matching any known control message SHOULD be intercepted by the client and MAY cause the client to send an appropriate response to the user. Such messages SHOULD NOT be forwarded to the server.

> /quit
> /join "channel"
> /leave "channel"
> /switch "channel"
> /list
> /listmembers "channel"

Control Message Handling: Client Side

- /quit - the client MUST send this message to the server and SHOULD close the connection to the server afterward
- /join "channel" - the client MUST send this message to the server and MAY perform additional actions, such as adding the channel to a locally stored channel list.
- /leave "channel" - the client MUST send this message to the server and MAY perform additional actions, such as removing the channel from a locally stored channel list.

- /switch “channel” - depending on implementation, the client MAY change a local variable to keep track of the user’s current channel, and it MAY send this message to the server.
- /list - the client SHOULD send this message to the server, it MAY perform handling locally if a server channel list is stored locally.
- /listmembers “channel” - the client SHOULD send this message to the server, it MAY perform local handling if channel membership information is stored locally.

Control Message Handling: Server Side

- /quit - the server SHOULD remove the user from any channel lists that they are in, and MUST close the connection.
- /join “channel” - The server SHOULD add the user to the member list for the requested channel, and MUST begin sending any subsequent messages directed to that channel to the user. If the requested channel does not exist, the server SHOULD create the channel and add the user to it.
- /leave “channel” - The server SHOULD remove the user from the member list for the requested channel, and MUST stop sending messages from that channel to the user.
- /switch “channel” - The server SHOULD change the user’s active channel, if such information is stored at the server level.
- /list - The server MUST reply with a list of all public channels on the server, it SHOULD send this as a string in the standard message format.
- /listmembers “channel” - The server MUST reply with a list of all members in the requested channel, if there are any. It SHOULD send this as a string in the standard message format and it SHOULD return an appropriate message if the channel does not exist.

Asynchronicity

A requirement of this protocol is that both server and client MUST be able to send and receive messages in an asynchronous manner - that is, on the client side, the receipt and proper handling of messages from the server MUST not be held up by the sending of messages or hung on user input, or vice versa. Similarly, the server MUST be able to handle the sending and receiving of multiple messages from multiple connected clients, while also accepting new incoming connections, in a way that is transparent and asynchronous from the individual client’s perspective.

One possible way to handle this is as follows (this is the way that the author has handled it in his implementation):

Client side:

The ability to receive messages and the ability to send messages are implemented as separate classes each inheriting from the Python threading.Thread class, and are each given their own execution thread in the main process. In this way, both the sending and receiving of messages can be performed in parallel, or in pseudo-parallel operation governed by the OS. The code for initializing these threads in the main function is as follows:

```
listen = listenThread(s)
send = sendThread(s)
listen.start()
send.start()
```

```
listen.join()
send.join()
```

The first two lines initialize instances of the threaded classes, the last four start them running in two separate threads and then wait for them to stop running.

Server side:
Threading is a possible solution to maintaining asynchronicity on the server side as well, but was not the tactic taken by this implementation. Another tactic, the one taken by this implementation, is to maintain a data structure of active connections, and to iterate through this data structure, servicing connections in a round-robin fashion, either checking each one to see if it has a waiting message, or in this implementation by making use of the Python select.select method to choose only connections that do have pending messages. Note that a special socket, the one listening for new TCP client connections, must be added to this data structure so that new connections can be made, and special handling must be performed when this is the connection selected to accept the new connection and add it to the data structure. Truncated implementation code follows:

```
while(1):
        inputready,outputready,exceptready = select.select(input,[],[])

         for s in inputready:

                if s == server:
                // accept new connection and add to input list

                else:
                // receive and process message
```

Passing Messages
As mentioned above, messages are to be passed as byte streams encoded in the UTF-8 format, and both client and server SHOULD encode all outgoing messages in this format and SHOULD decode all incoming messages in this format.

This can be done in python with code similar to the following snippets:

Encoding to byte stream:

```
msg = msg.encode('UTF-8')
```

Decoding from byte stream:

```
msg = data.decode('UTF-8')
```

Furthermore, the client SHOULD break long messages into chunks no larger than 1024 bytes, and send these chunks as separate messages. This practice allows the server to rapidly iterate through its round-robin connection servicing described in the implementation above, without slowing down to wait for the last packet of a particularly large message to arrive or having to

perform complicated end of message checking. The server SHOULD treat these messages as separate and dispatch them to the appropriate channels as such. This maintains a good user experience, as larger messages are simply broken into separate messages from the same user, while maintaining their textual order.

A way to implement the client side of this in python follows:

```
while(len(msg) > 1024):
        s.send(msg[:1024])
        msg = msg[1024:]

s.sendall(msg)
```

The server side of this is trivial, as it simply calls recv on the connection every time it is serviced with a maximum byte size of 1024, catching the following portion of a chunked message on the next time that the connection is serviced and treating it as a separate message.

Handling errors:
This document identifies three major categories of errors to be handled by the client and server.

- Error type one: bad user input.
    - Problem: The user sends a message larger than the server is prepared to receive on a single connection service.
    - Handling: The client side SHOULD break messages larger than 1024 bytes into multiple messages. The server SHOULD be prepared to receive messages of up to 1024 bytes in a single servicing pass.
    - Problem: The user tries to join a channel that does not exist.
    - Handling: The server MAY return an appropriate error message or it MAY create the requested channel (this implementation chooses the latter).
    - Problem: The user tries to switch to a channel that does not exist
    - Handling: The server OR client MAY return an appropriate error message or the server MAY create the channel and switch the user to it (this implementation chooses the first of these options).
    - Problem: The user tries to list members of a channel which does not exist
    - Handling: The server SHOULD return an appropriate error message
- Error type two: client cannot connect with server. This can happen because the server has crashed, the server has closed connection with the client unexpectedly, or a TCP connection error has occurred. All three of these scenarios SHOULD be handled by informing the user that the server could not be reached, and gracefully closing the client program. This can be done in Python by wrapping all socket operations in a try/catch block.
- Error type three: The server cannot connect with the client. This can happen because the client has crashed or closed the connection unexpectedly, or a TCP connection error has occurred. In the round-robin connection servicing implementation with Python select, losing connection with a client will not be catastrophic, as the dropped connection will simply never register as having a new message, and will thus never be serviced. Still, the server SHOULD periodically check for and drop such failed connections, and SHOULD implement error handling such as a try/catch block around sections receiving

messages from a client connection, in case a connection error occurs during message transmission.

Joining the server

On startup, the client MAY request a nickname for the chat from the user. It SHOULD pass this nickname, or a predefined default, to the server upon connection. The server SHOULD store the user's nickname along with their connection, for the purpose of listing users in a channel, and identifying messages coming from users to other users. It MAY send a message to all users in the default channel that the user has joined.

Either the client or the server or both MAY assign a default channel to the user. Upon joining, the server MAY add the user to this default channel, and set it as their current channel Either the client or the server MAY maintain information about the user's current channel, but one of them MUST do so. This implementation chooses to store the user's current channel at both the client and the server, with the server being the object of record which determines to which channel the user's messages are delivered.

Sending messages

Any messages given as input to the client program by the user NOT beginning with the reserved command character '/' should be passed to the server, after breaking any messages larger than 1024 bytes into separate 1024 or less sized messages. Messages sent by the client that are not control messages SHOULD be delivered to all users in the sending user's current channel. The sending user's current channel MAY be stored by the server, or it MAY be passed by the client along with the message, but one or the other MUST be true (this author's implementation uses the former approach).

Joining a channel

When the user inputs a message beginning with the /join command, the client MUST send this message to the server, and it MAY perform additional actions, such as adding the channel to a local list. The server SHOULD add the user to the member list for the given channel, it MAY create the channel if it does not exist, or it MAY send other input back to the user if it does not add them to the given channel for whatever reason. After the user has joined a channel, the server MUST send all subsequent messages directed to this channel to the user.

Leaving a channel

When the user inputs a message beginning with the /leave command, the client MUST send this message to the server, and it MAY perform additional actions, such as removing the channel from a local list. The server SHOULD remove the user from the given channel if they are a member, it MAY send an informational message back from the user if the channel does not exist or they are not a member. After a user has successfully left a channel, the server MUST NOT send any more messages directed to that channel to the user.

Creating a channel

This MAY be implemented as separate functionality, or it MAY be done through the user issuing the /join command with a channel name that does not exist. This author's implementation chooses the latter.

Receiving messages
Any message bound for an extant channel with members MUST be sent to all members of the channel by the server. The client MAY print inbound messages for all channels that the user is currently in, or it MAY print only messages for the user's current channel, storing messages for other channels in a buffer. This author's implementation chooses the former, with additional context given on each message to show what channel the message was sent from.

Switching channels
Any non-control messages sent to the user MUST be relayed to all members in the user's current channel. When the user issues the /switch command, the client MAY send this command to the server, and it MAY update the user's locally stored current channel. Upon receiving this message at the server, the server MAY update the user's server-stored current channel. This author's implementation chooses to do both, with the current channel of record to which messages from the user are sent being the one stored at the server. The server MAY also send back an informational message if the requested channel does not exist.

Sending messages to a different channel
This may be done through an explicit command, or it may be done by the user changing their current channel with the /switch command. This author's implementation chooses the latter. Note that in this implementation, users are able to switch to a current channel without joining it, meaning that they can send messages to this channel without receiving any from it. Other possible implementations may refuse to allow the user to switch to a channel without first joining it.

Listing channels and listing members
These commands SHOULD be passed by the client to the server, and the server SHOULD service them by returning a list of channels, a list of members in a given channel, or a message that a given channel does not exist, accordingly. These commands MAY also be serviced by the client, if the client maintains a local channel state and channel membership state (this author's implementation does not).

Quitting the server
When the user inputs a /quit control message, the client MUST send this message to the server, and the client MUST close the connection. The server MUST also close the connection, and SHOULD remove the user from any channel lists. The server MAY send a message to other users that the user has quit.