CS438 Project Report

Aarya Bhatia

April 29, 2023

Abstract

In this project I implemented a client and server program following the IRC (Inter Relay Chat) protocol in the C language. The IRC is a distributed chat server protocol which enables message delivery between clients that are connected to different servers in real time. An IRC server does this by relaying the messages to its peers.

The goal of this project is to get familar with distributed systems and learn about an important networking protocol used by many internet users. This project taught me the networking protocols used at a large scale and gave me insight about the kind of problems we can come across. This project also gave me experience in programming, debugging and testing distribtued systems.

For this project, I followed the RFC 1459 given in `https://www.rfc-editor.org/rfc/rfc1459` for the IRC Server specifications.

The source code for this project can be found on Github here: `https://github.com/aarya-bhatia/irc`. The repo contains a `README.md` file that describes how to build and use the programs.

Here is a demo video for this project on YouTube: `https://www.youtube.com/watch?v=MTd-5KoxiPo`

1 Implementation

The IRC is an application layer protocol used for real time text communication between multiple users on the Internet. The protocol was developed in the late 1980s and has since become a popular means of online communication.

The server is implemented as a single-threaded event-driven application. It uses the linux polling API and non-blocking TCP sockets for all network communication. This allows the server to service many requests at once and be more efficient. The server uses buffers and message queues for communication. There are various data structures like the `Connection struct` to handle the communication for each connection. The message queue allows the server to prepare messages to be sent to a client before the client is ready to receive them. It also lets the server parse multiple messages sent by the client at the same time and save them for future. The buffers store the current request or response data. Since the connections are multiplexed, these buffers may be in an incomplete state. However, when the messages are fully received or sent, the message queues will always be in a correct state.

An IRC network is composed of many servers running in parallel as well as many clients connected to a single server. The network topology is always a spanning tree by design. This ensures that there are no loops in the network. Therefore, servers can relay messages to every connected peer to propogate messages from one server to the others. Secondly, there is always a single path between each pair of clients due to the nature of the network. This eliminates the risk of duplicate messages and infinite cycles of relayed messages.

The servers in the IRC network need to keep the state of the network in sync with each other. A server keeps track of its own information as well as the information of clients and peers connected to it. The server also keeps track of other servers and clients that it can reach through its peers. With this information, the server implements the cycle detection logic so that any illegal connection can be dismissed immediately. The server also implements nick collision detection and has the ability to check if a nick exists on the network.

The client and server use a config file to configure the servers. Only servers specified in the file are allowed to participate in the network. This is recommended by the RFC. The config file contains the name of the server, the IP address, port number and password as a CSV file. The password is used for server-server registration. The port is where the server listens for new connections from clients or peers. It is possible to edit this file to add or remove a server. Here is an example of a valid entry in the `config.csv` file:

`testserver,192.168.64.4,5000,password@1234`

The client is implemented as a multi-threaded application so that it can communicate with the user on stdin as well as the server over a socket. The client also uses the `Connection struct` and messages queues like the server. The client message queues are synchronised by mutex locks so messages can be sent from various threads. Most commands read from stdin are sent verbatim to the server as plain-text, with only the addition of a CRLF delimiter.

A user can start the client by typing `./build/client <ServerName>` on the terminal, where `ServerName` is a server that exists in the config file. The client will try to establish a TCP connection with the IP and port as specified by the entry in the config file. The user can interact with the client by typing IRC commands such as "NICK example" or `PRIVMSG \#aarya :Hello`. The new line character is used to end the messages. All messages are at most 512 bytes. The client also displays the replies sent by the server to stdout asynchronously.

The client can accept special commands that are prefixed with a '/'. These commands allow us to perform special actions like registration. We can use the client to behave as a user or another server (only for testing). For users, the client can recognise a text file that contains the username, realname and nick for that user. This allows the user to login with `/client filename`. The client will load all the user details from the specified file and make the registration requests to the server on its own. These requests include the NICK and USER message.

For example, a client can create a file 'login.txt' with the following contents: `aarya aaryab2 :Aarya Bhatia`. Now the client can start the program and type `/client login.txt`. The client will use this file to create the initial registration request and push them to the message queue, ready to be sent to the server. The server registers the client if the request succeeds and it notifies the other servers about the users existence. This enables the new user to receive messages from other users on the network. The registration can fail for various reasons, for example the NICK chosen by the user may be in use. Furthermore, the requests sent by the client may be malformed. In any case, the server will remove the connection with the client and no other messages will be exchanged.

After registration, the user can quit the client program by typing the `QUIT [:<reason>]` command. This will gracefully stop the client. There will be a final exchange of messages between the client and server to faciliate the dismissal of the client from the network. The exact process for quitting is explained in the 'Client' section. When a user quits the network, the server must update the data structures on its side, and notify the entire network that the client has left. The other servers recursively update and notify their peers until everyone in the network knows that the client has left.

For chatting between users or on a channel, we use the `PRIVMSG` or `NOTICE` command. The difference is that the NOTICE command does not generate automatic replies even on an error. The NOTICE command is primarily meant for bots but has the exact same format as PRIVMSG. These commands are discussed later.

To use the client program in 'server mode', we can run a similar command on the client such as `/register ServerName`. Here `ServerName` refers to a server in the config file as before. This command establishes a server-server connection where the client is acting as the 'other' server. This mode is useful for testing.

The initial messages sent in a server-server connection are different from that of a client-server connection. For a server-server connection, the IRC protocol requires that a "SERVER" and "PASS" message is exchanged between both parties - the receiver and the sender. The server initiating the connection will be called the 'ACTIVE SERVER' whereas the opposite server will be called the 'PASSIVE SERVER'. In a client-server connection we use the 'NICK' and 'USER' message.

2 Server

The server handles three kinds of polling events for each socket connection:

- If the event occurs on the server's listening socket, it indicates to the server that the server can accept new connections.
- On the event of an error, a client has disconnected. A disconnection event is of importance because the server should inform the rest of the network about this client. The server also needs to update its hashtables to remove this user wherever required. The server notifies the network by sending a `KILL <client>` message and fills in the nick of the disconnecting client.
- On a write event, we send any pending messages from that user's message queue if any. We

can prepare messages to be sent to any client by adding them to their message queues. The messages will eventually get delivered to the client when they are ready to receive them.

- On a read event from a client, we receive any data they have to send and parse the messages if it is completed. A message is 'completed' when the CRLF delimiter is found in the message. Moreover, every IRC message has a maximum length of 512 bytes. If a message exceeds this limit without a CRLF we return an error status and the server can remove the client sending the malformed request. On a successful but incomplete message, we store the bytes in the user's request buffer. Messages are transferred to the queue only when they are completed. A message that is malformed or contains an invalid command may generate an error response or it may be ignored.

The main structs used by the Server are the `Peer`, `User`, `Connection` and `Server` struct. The following sections discuss some of these data types in more detail.

3 Connection struct

- The connection struct is a generic type which helps us read or write data to any socket connection. It is used by both client and server.

- The connection struct has a type parameter which can either be `UNKNOWN_CONNECTION`, `CLIENT_CONNECTION`, `PEER_CONNECTION`, or `USER_CONNECTION`.

- A new connection on the server is set to be `UNKNOWN`. It is later promoted to a `PEER` or a `USER` connection when the client sends the initial messages i.e. a user would send a `NICK/USER` pair and a server would send a `PASS/SERVER` to register themselves.

- Each connection can also store an arbitary data pointer. This parameter is used to store a pointer to a Peer or User struct depending on the type of connection. It is initialised when the connection type is determined. The request handlers only deal with this data interact with the Connection through the 'message queue'.

- Message Queues and Message Buffers: A connection struct contains incoming and outgoing message queues in addition to request and response buffers. This allows us to send or receive multiple messages from a client at once. It is also used for storing chat messages from another client. Since we may not send or receive all the data at once (due to nonblocking IO), we also keep track of where we are in the buffers using integer indices and offsets. All of this logic is encapsulated by the Connection struct.

- Note that the User and Peer struct have their own internal message queues. Initially messages are put in the main message queue, but after client has registered, the messages are put in the internal message queues. This is done so we don't have to deal with Connection structs in the request handler functions and we do not have to cast the data to the right type. We also create an abstraction between the client and the connection.

4 Server struct

The server data is stored in the Server struct. The server contains data about all of its clients, users and the network - such as the nicks that exist in the network. The servers must know this information to avoid nick conflicts. Also, the server uses this data to route messages.

The Server uses the following data structures:

- A hashtable `connections` is used to map sockets to connection structs for each connection, i.e clients and peers.

- A hashtable `nick_to_user_map` to map a nick string to a user struct for a user. The users get a random nick in the beginning. When the user updates their nick, the entry in the hashmap for that user is also updated. This map is used to check if a nick is available and fetch a user by their nick.

- A hashtable `name_to_channel_map` to map each channel name to a channel struct. Each server has their own copy of the channel. Since different users are connected to different servers, a channel message must be propogated to the entire network in order to reach all channel members. But a single server does not know all the clients in the channel.

- A hashtable `name_to_peer_map` is used to map the name of a peer server to a Peer struct.

When a server-to-server connection is established the remote server becomes a peer for the current server. The server which initiates the connection is known as the `ACTIVE_SERVER` and the server which accepts the connections is known as `PASSIVE_SERVER`. The entry is added after registration as the peer name is not known before the SERVER message is received The IRC does not enforce 'default names' unlike the case with users.

- The entry for a peer is removed when the peer quits or disconnects. The server that discovered the disconnected peer must generate a `SQUIT` message to notify the other servers in the network as well. This has a similar purpose as the `KILL` message for user disconnections.
- A hashtable `nick_to_serv_name_map` is used to determine which users are connected to each server. This map is updated when a peer advertises a new user connection or relays the message from another server. This map helps keep track of all the users on the network at each server. The entries are removed when a user disconnects and a server sends a KILL request for that user.

5 Parser

I implemented a simple message parser which is used to parse messages and check for errors. This is used by both server and client. The details about the syntax of the message is given in the RFC and will be discussed briefly. Simply speaking, an IRC message includes the following tokens in the message:

- **Source**: This identifies the sender of the message. Clients never fill in this field. However, a server fills in this field on behalf of the client for commands like `PRIVMSG`. The server always fills in this field with their own server name when sending a message to a user or peer. When this source field is used, it begins with a ':' character.
- **Command**: This field is filled by a 'command' if the message is a request from the client. It is filled by the 'reply' if the message is a response from the server. For example, when a user sends a `PRIVMSG`, then `PRIVMSG` is the command. When the server sends a message, often times the 'command' is filled by a numerical code. A list of all numerical replies can be found in the RFC and also in the `replies.h` file included in the source code.
- **Parameters**: There are 15 parameters that we can use in the IRC protocol. These are filled with any parameters we might want to send with a message. Usually, the first parameter is the 'target'. For example, in a `PRIVMSG`, this target can be the nick of the receipeint of the message. Parameters do not contain any spaces because space is the separator used between parameters.
- **Body**: This is considered as the final parameter and can contain arbitary text. This is where the user would add their 'message' in a `PRIVMSG` command. The body is prefixed by another ':' character, and can contain spaces. This field is also used for the 'realname' of the user in a 'USER' message, because the 'realname' can contain spaces. The servers often use this field to describe the reply, as most IRC clients hide the 'parameters' from the user. They usually use this field to display to the user.

6 Commands

The server currently supports the following commands from the clients: MOTD, NICK, USER, PING, QUIT, HELP, PRIVMSG, NOTICE, INFO, WHO, NAMES, LIST, PART, JOIN, TOPIC, CONNECT.

6.1 QUIT

- The QUIT command is used by a client to indicate their wish to leave the server.
- All data associated with this user is freed and socket is closed.
- An ERROR reply is sent before closing the socket to allow the reader thread in the client to quit gracefully.
- A client is not removed immediately. Instead we set a `quit` flag in the user data because we still need to send the final messages.
- When all messages are sent and the user is marked as quit, the server closes the connection.

6.2 MOTD

- This command sends the "Message of the Day" to the user. - It looks up the current message from a file (motd.txt) that contains a list of quotes.
- This file was downloaded using the `zenquotes.io` API using a small python script in `download_quotes.py`.
- This list can be updated by running this script at any point.
- The server fetches the line that is at position equal to `day_of_year % total_lines`, where `total_lines < 365`.
- The filename can be changed at any time as the server reopens the file each time. It is not neccessary to cache this file as the MOTD is only sent once for each client on registration or if someone requests for it.

6.3 NICK/USER

These two commands are used for user registration.

- A user can register with NICK and USER commands - NICK can be used to update nick at any time
- The username and realname set by the USER command cannot change.
- Users can use NICKs to send messages to another user. (See PRIVMSG and NOTICE for more information).
- A user's nick is freed i.e. made available to other users when they leave the session.
- A server always knows every client on the network.
- NOTE: There is a particular scenario known as **net split** where we experience a NICK collision and we cannot guard against it: Suppose there are two *disjoint* IRC networks containing a user with the same NICK. Suppose that these two networks are joined by a server-server connection. This results in us having two clients with same NICK on the new network. However, it is impossible to prevent this at registration time. Thus, the IRC specs suggest to use the KILL command and remove both the clients from the network. This is the approach used by me in the project.

6.4 PRIVMSG

The PRIVMSG command works as follows: It takes the name of a target as the first parameter and the message text as the final parameter. The server checks if the target is a user or channel before making a decision on how to relay it forward. If the target is found on the original server, the server can push the message to the target user(s) message queues without relaying it to anyone. Otherwise, the server must relay the message to send it to the destination client's server.

- This command is only enabled after registration for both sender and receiver.
- User can send messages to another user using their nick only. This command can accept any nick that exists in the network as a valid target.
- This command can also accept any channel name as a valid target. We only support channels that are prefixed with #. These characters have special meaning- for example, they tell us if the channels are public or private.
- Since users can exist on different servers, the server has to relay these messages to certain peers. Channel messages are different from user-user messages because all members of a channel do not exist on the same server. The server does not know where the members of the channel live, whereas a user can only live on one server. Therefore, there are more messages relayed for a user-channel message as compared to a user-user message.
- If the message target is a user, the message is relayed to only one peer in each step. When the final server connected to the target user receives the message, it can deliver the message to the user and stop relaying. Every user-user path is unique because the IRC network is a spanning tree and does not contain cycles.
- If the message target is a channel, the message is broadcasted to all of the peers of each server. This way, the message is propogated to every other server on the network. we could decrease the number of messages sent if we knew where all the members of the channel lived. At each step of a channel relay message, the server does two things. First, it delivers the message to any available channel members on the current server. Second, it relays the message to its peers. The messages to the users are delivered using the queue system.

7 Client

The client implements a thread-based model with two threads:

7.1 Main thread

- It is the duty of the main thread to read user input from stdin.
- The main thread blocks on the getline() instruction till the user types ENTER to send a line of text.
- If the input is a valid IRC command, the string will be terminated by CRLF as required.
- If the input is a special command starting with a /, the client will translate the string to the corresponding IRC command.
- The IRC command is added to the outbox queue of the user to send to the server through the worker thread.
- The main thread quits when the user enters the QUIT or SQUIT command.

7.2 Worker thread

- This thread polls the server socket for read/write events.
- It displays the messages read from the server to stdout.
- It writes available messages from the outbox queue to the server.
- This thread quits when the server sends a ERROR message. An ERROR message is also sent in response to the QUIT command. This allows the worker and main thread to exit gracefully.

The process to quit the program is as follows: The user types the QUIT command. The main thread adds the final message to the queue and exits afterward. The worked thread sends the QUIT message from the queue but waits for the response. The server replies to the QUIT command with a ERROR message. The worker thread receives a ERROR message and exits.

The ERROR message is also sent in case of errors such as incorrect password. In this case, the worker thread will exit while the main thread is stll alive. The main thread can now reconnect to the server if required. As part of future work, the main thread could relaunch the worker after an error has occured and prompt the user for the new registration message.

8 Special commands

The following commands have been designed for demonstration and are not part of the specification.

The `TEST_LIST_SERVER` command is asynchronous in nature, i.e. it involves communicating with the entire network to create a response. Therefore, the reply is sent over a series of messages and we maintain the current state of the response to decide when the response is completed.

The command `TEST_LIST_SERVER` is used to get a list of all servers on the network. Note that, in my implementation, each server knows which servers exist in the network. However, I the purpose of this command is to watch my servers communicate with each other to generate a response. This command should only be used when running my servers, as other servers would not support this command.

To implement this command we use two kinds of replies.

- 901 `RPL_TEST_LIST_SERVER`
- 902 `RPL_TEST_LIST_SERVER_END`

The algorithm is described as follows:

- A user requests the origin server for a list of servers on the network using the `TEST_LIST_SERVER` command.
- The origin server adds the user to a map `test_list_server_map` and initialises a struct with the following members:
  - A pointer to the connection struct of the user. This variable is used to send the response to the user in the future.
  - A set containing all the "remaining" peers, filled with the current peers of the origin server: this set inidicates which peers still need to send a reply. We can use this set to decide when the response is over.
- We initially send the client the names of the original server's peers. Each server is returned in a new message of type `RPL_TEST_LIST_SERVER`.

- Next, we make a `TEST_LIST_SERVER` request to each of the peers in the set. The client nick is used as an identifier in these requests to distinguish them.
- We can continue processing other commands from the user while waiting for replies for the `TEST_LIST_SERVER` request.
- When one of the peers sends back a reply for the servers it has found, we relay the replies the original client and remove the peer from the set. The peers perform this action recursively. In particular, when one of the peers sends a `RPL_TEST_LIST_SERVER` to the origin server, we add this response to the original client's message queue, accessed by the connection struct. The response contains the client nick which we can use to index the `test_list_server_map` for the data.
- When the set is empty, we have finished listing all servers so we can send a `RPL_TEST_LIST_SERVER_END` to the original client and remove their entry from the map. If one of the servers leave, they are removed from this set too, otherwise the client will wait forever.
- If during this period a new server joins the network, we ignore them from the response. It is possible to add this peer to the set for "real time" updates.
- If during this period the user sends another list command, we do not acknowledge it. This prevents us from seeing duplicate server names in the response because the command is asynchronous.

There are numerous ways to extend the functionality of the IRC protocol and add support for features like file transfer or unicode emojis.

We can either send data through a custom client-side protocol or a server-side protocol. For example, our messages can encode a file and the target client can decode the bytes to get the original file.

In my example, `TEST_LIST_SERVER` was a server-side protocol as the servers need to implement the functionality of this command.

8.1 Network State

In this section, I list the important events in the network. It is critical to handle these events properly or the network state will get corrupted.

- Event of client connection:
  - The client is removed if their nick is already in use by another client.
  - The network is notified about this new client using a NICK command, which is different from the NICK command used by the client.
- Event of server disconnect:
  - Broadcast a SQUIT message for each server behind that connection
  - Broadcast a QUIT for each client behind that connection
- Event of new server connection:
  - Error checking: A server that is already known, either directly or indirectly is not added again. If a server name is not found in the config, it is not accepted into the network. A server cannot connect to "itself".
  - **Authenticate Server**: Both servers must share the passwords of the opposite server in the PASS message and confirm that they match the configured passwords (The servers can have a different config file). Both servers also share their name with each other with the SERVER message.
  - Both servers share the names of the servers that they know about
  - Both servers share the nicks of the clients that they know of
  - Both servers share their channels with each other
  - At this stage, the servers also check for cycle detectioin and NICK collision as explained before.
- Event of client disconnect:
  - Remove client from all data structures
  - Broadcast a KILL for that client to all peers

9 Routing messages

Observe that the IRC network is a spanning tree structure, so it has no cycles. We use a BFS like algorithm to route messages from one server to another.

Suppose we have three clients: Alice and Bob and Kate. There are three servers A,B and C. There network graph has the following topology: $A \leftrightarrow B$ and $A \leftrightarrow C$. Further, suppose Alice is connected to server A, Bob is connected to server B and Kate is connected to server C.

Suppose alice wants to send a message "hello" to kate.

The following actions will take place:

- Alice will send a message `PRIVMSG Kate :hello` to server A
- server A will check if Kate is a client known to it.
- Since Kate is not a client on server A, server A will relay this message to each of its peers in the `name_to_peer_map`, namely server B and server C.
- server A will add B and C to a visited set to avoid relaying multiple times to a peer - each peer can appear multiple times in the `name_to_peer_map` because there is a many-to-one correspondence between the servers and the peers.
- server B and server C take the same action when they recieve the message from A.
- Since Kate does not live on server B, server B can ignore the message. B will not relay message back to A as the messages are never relayed to the sender as it is redundant.
- server C will finally find that Kate lives on server C, so server C will not relay this message any further.
- server C will add this message to Kate's message queue. When Kate is ready to receive messages, this server will send this message over a socket to Kate's client program.
- the client program will display this message to Kate once it receives the full message i.e. the CRLF bytes. This completes the journey of the message from client Alice on server A to client Kate on server C.

As described above, the messages are propogated down the network until they reach the destination or an edge node. The messages make their way to the destination through a series of relays on intermediate nodes. This is the main idea of routing messages in the IRC protocol. In fact, there are similarities between IRC and distance vector routing, except there is no concept of shortest paths, as there is only one path. This example was a general description of routing messages, however, we can do better- Since each server knows where the client lives, each server only needs to relay the message to one peer instead of all. As the servers send advertisments, the other servers in the network can update their tables and learn the "next hop" node for each target. If the server does not keep track of this data, then it may relay everywhere and still have the same outcome.

10 Conclusion

While IRC protocol has its advantages, it should only be used in small networks. Several limitations of the IRC protocol are mentioned in section 9.1 of the RFC. There is an overhead of knowing the network state on every server. Also it is hard to communicate all network updates with very little latency at the scale of the IRC network. For such reasons, we should seek a more scalable protocol today.

In conclusion, this project taught me a lot about networking protocols and distributed systems.