Features

Summary

In this project, I have implemented a server and client program that follow the Inter Relay chat protocol. 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 system. It uses the linux polling API as well as non blocking IO 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 messages that are relayed. However, the server needs to keep track of all the servers that it is currently connected to and all the servers that it can reach through one of its peers. The server implements the cycle detection logic so that any illegal connection can be dismissed.

The servers 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. 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 manage the servers. The config file behaves like a database for the network. 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, with only the addition of a CRLF delimiter. This works well because IRC is a text-based protocol.

A user can start the client by typing `./build/client <server-name>` on the terminal, where server name is a server that exists in the config file. The client will try to connect to the host and IP specified by the server's entry in the file. Then, 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. 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 register faster. 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 delivered 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 removal of the client from the network. The client should send a QUIT message to the server and the server will reply with an ERROR message. The client thread will exit on receiving the ERROR message. The main thread would have exited after the first QUIT message was sent. Finally, the program can be closed. Note that the server must update the data structures on its side, and notify the entire network that the client has left. The other servers recursively perform the same action.

For chatting between users or on a channel, we use the `PRIVMSG` command. There is more about this command later. The way this command works is that it takes the name of a target as the first parameter and the 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.

To use the client program in 'server mode', we can run a similar command such as `/register servername`. Here servername refers to a server name in the config file as mentioned before.

Logic

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

- If the event occurs on the server's own listening socket, it signals to the server that it can accept new connections.
- On an error event, the client is disconnected. A disconnect is important because the server should inform the rest of the network about this user. 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 pending messages from that user's message queue if any. So we can prepare messages to be sent to any client by adding them to their message queues. The messages will eventually get delivered when the client is 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 malformed client. 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.

The main structs used by the Server are the `Peer`, `User`, `Connection` and `Server`. This is what each of them do:

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 arbitary data for the client. 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.

- Message queues and 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 to and fro 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, we also keep track of where we are in the buffers using integer indices and offsets.
- 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 handlers functions and have to cast the data to the right type.

Server

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, such as a client or peer.
- 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. As they update their nicks, the entry in the hashmap for that user is also updated. This map is useful to check which nicks are available and also access the User data when we are to deliver a message to some 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 entry is removed when a server quits or disconnects and a SQUIT message is generated by the server that realised this event.
- 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.

There is also a simple message parser in `message.h` whcih is used to parse messages and check for errors. This is used by both server and client.

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 and a special command TEST_LIST_SERVER which is discussed later.

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.

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.

NICK/USER: 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 risk of NICK collisions in an unavoidable scenario known as a **net split**: If there are two disjoint IRC networks containing the same NICK and these two networks are joined by a server-server connection then we end up with two clients with same NICK on the new network. However, it is impossible to prevent this at registration. 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.

PRIVMSG

- This command is only enabled after registration for both sender and receiver.
- User can send messages to another user using their nick only.
- Server can accept any nick that exists in the network as a valid target.
- Server can also accept any channel name as a valid target. We only support channels that are prefixed with '#'.
- Since users can exist on different servers, the server has to relay such messages to its peers. Channel messages are different because members of a channel may exist on multiple servers.
- If the message target is a user, the message is relayed to only one peer in each step, which is either the destination server or a server that is peers with the destination server. Every client to client message has a unique path because the IRC network is a spanning tree and does not contain cycles.
- If the message target is a channel, the message is always broadcasted to all peers of a server so that the message is propogated to every other server. If the server knew which members live on which servers, we could decrease the number of redundant messages sent. At each step of a channel relay message, the server should check for any members of that channel on the current server and deliver the message to each user of the channel using the message queue system.

Client

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

Main thread

- It is the duty of the main thread to read user input from stdin.
- This thread blocks on the getline() instruction.
- If the input is a valid IRC command, the string will be terminated by CRLF appropriately.

- If the input is a special command starting with a /, the client will generate the corresponding command.
- The irc command is added to the outbox queue to send to the server.
- The main thread quits when the user enters the QUIT or SQUIT command.

Worker thread

- This thread polls the server socket for read/write events.
- It displays the messages read from the server to stdout.
- It writes the messages from the queue to the server.
- This thread quits when the server sends a ERROR message.

The process for exiting all threads works as follows: The user types in the QUIT command. The main thread recognises that it is time to quit and after adding this last message to the queue, will quit. The worked thread will send this message but wait for the response. The server will reply to a QUIT with a ERROR message. When the worker thread receives a ERROR message it will quit. The ERROR message is also sent in case of errors such as incorrect password.

Special Commands

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

The following command is asynchronous in nature, i.e. they involve communicating with the entire network to create the full responsne.

Edge cases:

Additional Command

I added an additional command that is not a real command but is used for demonstration and testing. The command "TEST_LIST_SERVER" is used to get a list of all servers on the network in an asynchronous way. To implement this command we use three kinds of replies and send the reply as a multipart message.

Replies:

- 901 RPL_TEST_LIST_SERVER_START
- 902 RPL_TEST_LIST_SERVER
- 903 RPL_TEST_LIST_SERVER_END

Algorithm:

- User requests origin server to list all servers on the network.
- server adds user to the map `test_list_server_map` and initialises a struct with the following members:

    - A reference to the connection struct to send messages: This is set to the user's connection
    - A set containing all the peers of this server: This set inidicates which peers still need to send a reply
- we send the original client all the names of the original server's peers. Each server is in a new RPL_TEST_LIST_SERVER message.
- now we request the peers of the original server to also send a list reply for the same request.
- we can continue processing other commands from the user while waiting for replies for the TEST_LIST_SERVER request.
- When one of the peers recursively sends back a reply for any servers behind their connection, we can send relay it to the original client.
- 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.
- Also, when one of the peers sends a RPL_TEST_LIST_SERVER_END to the origin server, this indicates that the peer has finished listing all the servers behind their server. Therefore, this peer is removed from the set.
- 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 during this period a new server joins, we simply ignore them from the response.
- If during this period the user sends another list request, we do not acknowledge it.

How to keep consistent state across the network

- 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:
    - Both servers share the names of their peers with each other
    - Both servers share the NICKs of their clients with each other
    - Both servers share their channels with each other
- Event of client disconnect:
    - Broadcast a QUIT for that client to all peers

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 cat. There are three servers A,B and C. There are the following edges in the network graph: A <-> B and A <-> C. Suppose alice is connected to A, bob is connected to B and cat is connected to C.

Suppose alice wants to send a message “hello” to cat.

The following actions will take place:

- alice will send a message `PRIVMSG cat :hello` to server A
- server A will check if cat is a client known to it.
- Since server A knows C, it will check if cat is a client on the server A.
- Since cat 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 because we may have multiple entries in the hashtable that map to the same peer. This is because there can be multiple servers behind one connection and each server keeps track of all other servers in its hashtable.
- server B and server C recursively do the same thing.
- server B knows the client cat and cat does not live on server B. At the point server B can ignore the message because server B’s only peer is A and a server should never relay a message back to the sender, otherwise there would be an infinite loop.
- server C will finally see that cat lives on that server. server C will not relay this message any further to avoid wasting bandwidth.
- server C will add this message to cat’s message queue. When cat is ready to receive messages, this server will write this message over the socket to cat’s client program.
- the client program will display this message to cat.

As you can see, all messages are propogated down the network until they reach the destination or an edge. The messages make their way to the destination through a series of relays on intermediate servers. This strategy would work on any number of servers as long as they follow the main rule that the network is a spanning tree.

Note: It is possible to optimise the path. There are optional features in the protocol to allow the servers to gain more information about the network. For example, we can attach a “hop count” parameter to some messages so that servers can know the distance between two nodes on the network. With more information, the server can pre compute the shortest path to the client and only relay messages to a few peers.