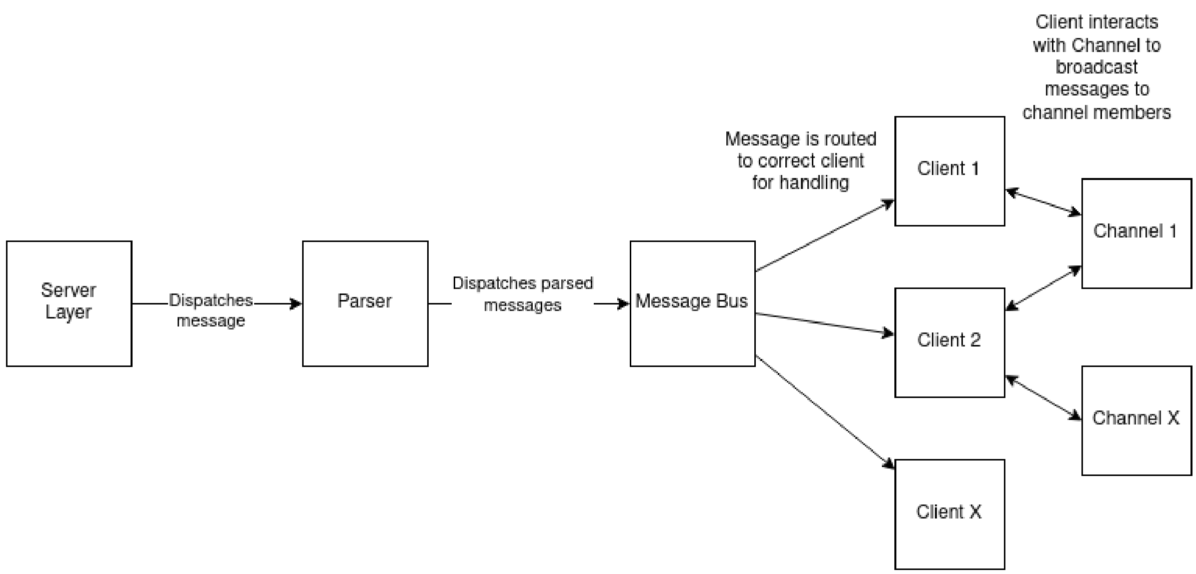
Design and Implementation

- Typical Message Life Cycles
- Event Loop
- Messages
- Handler Chain
    - Parser
    - Message Bus
    - Client
    - Channel
- IRC Implementation
    - Testing with a Client
- Acknowledgements

In what follows we describe the high level architecture of Pyircd. This can be depicted as follows:



Broadly speaking, each component in the diagram has the following responsibilities:

Server Layer: Runs the event loop and maintains references to socket connections for each client.

Parser: Parses messages according to the IRC specification.

Message Bus: Routes messages to the appropriate `Client`. If a `Client` for a particular address has not been instantiated yet, the message bus will instantiate it.

Client: Handles all IRC messages per the protocol, is responsible for handling its own life-cycle and for indicating that messages are to be written back to the client. For emphasis, the `Client` is an abstraction on the server-side. We refer to it as the `Client` since it handles all interactions on behalf of the actual client that is connected to the server.

Channel: The channel abstraction reflects the concept of an IRC channel in the specification. It allows for clients to communicate with other clients in the channel without explicitly knowing which clients are part of the channel.

Typical Message Life Cycles

For an example of how the above flow works, consider the beginning of the IRC connection registration process as an example.

A connection to an IRC server will be initiated by the client sending a message similar to `NICK foo\r\n`. This message would be processed as follows:

1. The `recv` call in the server's event loop will load the message into an `in_buffer` associated with the socket connection
2. The server will check the buffer and notice the IRC termination delimiter `\r\n` indicating that a complete message has been received. At this point, the server will create a new `Message` containing the client's address among other info such as the message and pass it along to the parser.
3. The parser will parse the message and generate a new `Message` that has the `command` and `parameters` fields populated with `"NICK"` and `["foo"]` respectively.
4. The message bus will check to see if any `Client` is associated with the inbound `Message`'s address. In our example, this is the first message the client has sent, so the message bus will instantiate a new `Client` and pass the message along to be handled.
5. The `Client` will invoke `_handle_nick` and will set the user's nickname to `foo` .

The above flow falls short of demonstrating how a write to a client would be done. We can see this by considering how the next command in the connection registration process, the `USER` command, would be handled.

Suppose that the previous client now sends us `USER foo 0 * bar \r\n` where `foo` is the username and `bar` is the real name of the user. This message is supposed to conclude the IRC connection registration process and result in the server sending a welcome message to the client.

In such a case, the message will be handled exactly the same as the above example with the exception that the message bus would not need to instantiate a new `Client` and that once the message reaches the `Client` a write will be done.

The `Client` will write its response to the `out_buffer` for the respective client socket. The `Client` object is given a reference to its sockets' `out_buffer` on instantiation. The reference is contained in the `Message` that is passed to the `Client`.

Once the client writes to the `out_buffer` its job is done. It is now up to the server layer to make a `send` call to transmit the message to the client. This is done during the normal execution of the event loop. As part of the loop, the server will routinely check if a socket is ready for writing and if there is anything to be written in the socket's `out_buffer`. If so, a `send` call will be made to transmit the data. Therefore, the `Client` "signals" to the server that there is data ready to be written, but the server writes of its own accord.

In what follows we go into further details of each of the components of our system.

Event Loop

As mentioned, the event loop is responsible for handling new connections to the daemon and servicing existing connections. New connections are received by the server socket, which is the first socket that we initialize on the server side. On receiving a new connection, we associate certain state with the new socket such as the previously mentioned `out_buffer` and `in_buffer` . These are unique to each socket connection.

Existing connections are serviced by dispatching appropriate messages to the parser or writing back into sockets if the `out_buffer` contains data.

The high level logic for server initiation and the event loop is as follows:

```
1. Create server socket that will be used to accept new connections
2. Begin event loop (using the best Selector on system (epoll, select, etc)
3. Check if any socket is ready for read/write
4. If server socket has a new connection:
  i. Register new client and go to 3
5. If non-server socket (a client) is active:
  i. If readable:
    * Read data into in_buffer
    * Check if in_buffer has IRC delimiter and if so dispatch message to parser
  ii. If writeable
    * Check if out_buffer has any data and if so write to socket
6. Go to 3
```

As noted above the event loop does not always use the `select` system call. We found an interesting note in the `cpython` source code indicating that “epoll|kqueue|devpoll > poll > select”. Python’s `DefaultSelector` picks the most efficient implementation for us. In our experience, `epoll` is typically used.

The event loop does not always check if a socket is writable. This is an optimization we developed to ensure the event loop is not needlessly looping over sockets since a healthy socket is generally always ready for writes. By default, we do not listen for write events on client sockets. When a `Client` writes a message to its `out_buffer` , it toggles the event loop to listen for write readiness on the respective client’s socket. The event loop will, after sending the message, set the socket back to only listen for reads.

The only case that is not mentioned here is how client disconnects are handled on the server side. In the event of a disconnect, the server will generate a “server side” event/message indicating the respective client has disconnected. This message is forwarded by the parser to the message bus and ultimately to the client without being parsed. The `Client`, on receiving this message, will unregister all its state so as to be garbage collected.

Messages

The event loop generates a `Message` that is dispatched to handlers whenever a message terminated with the IRC termination delimiter, `\r\n` is received. A `Message` has the following fields:

1. `client_address` - the address and port a client is connecting from, used to uniquely identify the client
2. `action` - the action to be carried out by the next handler
3. `message` - An unparsed IRC message
4. `key` - A SelectorKey, used to provide reference to the `out_buffer` to the Client for writing
5. `command` - If message is parsed, command will hold the IRC command
6. `parameters` - If message is parsed, this field will hold the IRC parameters

Handler Chain

Various handlers are responsible for carrying out the logic dictated by the IRC protocol as follows.

Parser

The parser parses the `message` field of a `Message` and generates a new `Message` with `command` and `parameters` fields populated. It dispatches this to the message bus.

Message Bus

As outlined previously the message bus has the simple task of routing messages to the correct client or instantiating them if they do not exist.

Client

The `Client` abstraction is the heart of the backend and is responsible for executing all of the logic that is specified by the IRC protocol. The `Client` will handle all client related state, such as the current nickname, registration status and joined channels.

The `Client` is also responsible for ultimately writing back information to the client that it represents. This is possible because the `Client` maintains a reference to the `out_buffer` associated with its clients socket connection. Each `Client` instance has a `send_message` method that handles writing to the `out_buffer` of the respective client.As previously mentioned, once the `Client` writes to the `out_buffer`, it toggles an option to listen for write readiness on the socket. The event loop will see this and make a `send` system call on the socket the next time it sees the socket is ready for writing and will toggle listening for write readiness off once it is done sending the message.

Finally, the `Client` handles its own lifecycle and is responsible for removing all state when a client sends a `QUIT` message or disconnects. Removing all such state is necessary so that the `Client` can be garbage collected so as to avoid memory leaks. In the event of a `QUIT` message, the client has reference to another variable `unregister_socket` that is associated with its respective socket connection. The `Client` toggles this to signal to the server layer that the socket connection must be terminated. The `Client` subsequently removes all state associated with it.

Channel

Channels in IRC are group chats of multiple users. Our implementation uses the Publisher-Subscriber design pattern to accomplish this functionality. When a `Client` joins or first creates a channel, it registers its own `send_message` method as a callback with the channel. The channel in return gives it a `broadcast` method.

The `broadcast` method is a closure that calls the `send_message` callback for every client registered with the channel except for the client that made the call. This effectively allows sending messages to all other clients in the channel.

IRC Implementation

IRC is a very broad protocol with multiple RFCs, implementation and features. We implemented a subset of the protocol per the horse docs which is a well accepted reconciliation of multiple RFCs.

In particular we implemented the following commands:

- NICK
- USER
- QUIT
- JOIN
- PART
- LUSERS
- PRIVMSG

- MOTD
- LIST

These allow users to accomplish the following functionalities:

- Connect to the IRC server
- Set and change nickname
- Join a channel
- Send and receive messages from/to other clients on the channel
- Leave a channel
- List existing channels on the server
- Send direct messages to users
- List number of users on the server

Testing with a Client

The above functionalities can be tested using IRC clients such as Weechat or Pidgin. The best client to test with is Weechat or any client that allows executing IRC commands through a text based input.

After installing Weechat, follow these instructions to test the above functionality:

1. Start our daemon, by default it will run on `localhost:6667`
2. Setup the connection using `server add pyircd` `localhost:6667`
3. Add a nickname using `` `/set irc.server.pyircd.username "My user name"` ``
4. Connect to the server using `/connect pyircd`
    a. You should see output indicating a successful connection and our logo as the message of the day
5. See the number of users on the server by typing `/lusers`
    a. Output should be…

```
There are 1 users and 0 invisible on 0 servers
I have 1 clients and 0 servers
// Our server is connected to 0 other servers, thus the 0
```

6. Change your nick by typing `/nick my_new_nick`
7. Create a channel by using `/join #d58` (channel names must start with a #)
    a. You can now join the channel using another client. See our demo for how to use Pidgin to do so
    b. You can now send message to the channel
8. Leave a channel by typing `/part` most clients will show output indicating you left the channel
9. Private message another connected user by `/query other_user my_message` (this command is Weechat specific, the IRC command is `/privmsg other_user my_msg`)
    a. Some clients do not show any error if the other user does not exist, although our server sends back the error
10. List existing channels by using `/list`
11. Quit using `/quit`

Acknowledgements

The IRC community has a put a lot of effort into developing resources to facilitate the development of IRC related projects. Our project relied heavily on community efforts. We were able to make contact with members of the community who guided us to resources.

In particular, we found the following resources invaluable:

Modern IRC Client Protocol Specification - The main documentation we relied on for implementation

irctest - An integration test suite we used to test our server during development

IRC Parser Tests - Tests we used to verify our parser was up to spec.

Additionally, we referenced the Python docs example on Selectors and this socket programming guide when implementing our server layer.