CMSC 23300 – Networks and Distributed Systems

Project 1: Internet Relay Chat

1 Introduction

In this project, you will implement a simple Internet Relay Chat (IRC) server called χirc . This project has three goals:

1. To provide a refresher of socket and concurrent programming covered in CMSC 15400.
2. To implement a system that is (partially) compliant with an established network protocol specification.
3. To allow you to become comfortable with high-level networking concepts before we move on to the lower-level concepts in this course.

This project is divided into three parts. The first part is meant as a relatively short warmup exercise, and you should be able to do it just by applying what you learned about network sockets in CMSC 15400. The other two parts are more complex, but should still be doable if you've taken CMSC 15400 (we will, nonetheless, be providing a review of sockets and concurrent programming in the first two discussion sessions of CMSC 23300).

This document is divided into three parts: Sections ?? through ?? provide an overview of IRC and provide several examples of valid IRC communications; Sections ?? and ?? describe how to get the χirc code and how to build it; finally, Sections ?? to ?? describe how the project will be graded, and the specific requirements of the three parts the project is divided into.

2 Internet Relay Chat

IRC is one of the earliest network protocols for text messaging and multi-participant chatting. It was created in 1988 and, despite the emergence of more sophisticated messaging protocols (including open standards like XMPP and SIP/SIMPLE, and proprietary protocols such as Microsoft's MSNP, AOL's OSCAR, and Skype), IRC remains a popular standard and still sees heavy use in certain communities, specially the open source software community.

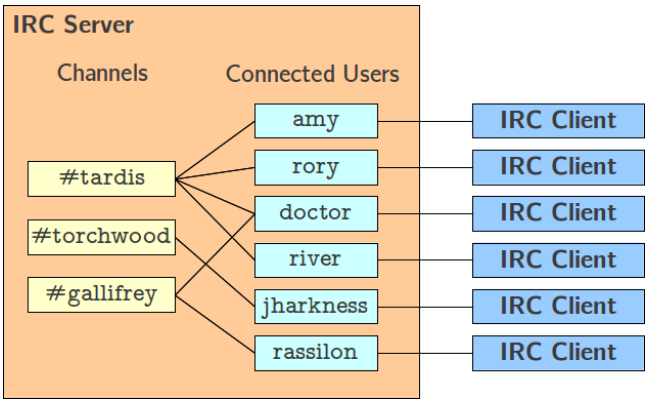

Figure 1: Basic IRC architecture

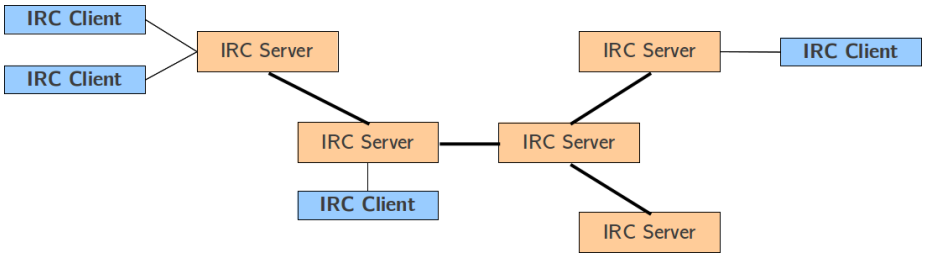

Figure 2: Multi-server IRC architecture

The basic architecture of IRC, shown in Figure **??**, is fairly straightforward. In the simplest case, there is a single *IRC server* to which multiple *IRC clients* can connect to. An IRC client connects to the server with a specific identity. Most notably, each client must choose a unique *nickname*, or "nick". Once a client is connected, it can communicate one-to-one with other users. Additionally, clients can run commands to query the server's state (e.g., to obtain a list of connected users, or to obtain additional details about a specific nick). IRC also supports the creation of chat rooms called *channels* for one-to-many communication. Users can join channels and send messages to the channel; these messages will, in turn, be sent to every user in the channel.

IRC also supports the formation of *server networks*, where multiple servers form a tree of connections to support more clients and provide greater capacity. Servers in the same network share information about local events (e.g., a new client connects, a user connected to a given server joins a channel, etc.) so that all servers will have a copy of the same global state. In this project, we will only consider the case where there is a single IRC server.

3 The IRC Protocol

The IRC protocol used by IRC servers and clients is a text-based TCP protocol. Originally specified in 1993 [RFC1459], it was subsequently specified in more detail in 2000 through the following RFCs:

- [RFC2810] **Internet Relay Chat: Architecture**. This document describes the overall architecture of IRC.
- [RFC2811] **Internet Relay Chat: Channel Management**. This document describes how channels are managed in IRC.
- [RFC2812] **Internet Relay Chat: Client Protocol**. This document describes the protocol used between IRC clients and servers (sometimes referred to as the "client-server" protocol)
- [RFC2813] **Internet Relay Chat: Server Protocol**. This document describes the "server-server" protocol used between IRC servers in the same network.

You are not expected to read all of these documents. More specifically:

- We recommend you do read all of [RFC2810], as it will give you a good sense of what the IRC architecture looks like. You may want to give it a cursory read at first, and revisit it as you become more familiar with the finer points of the IRC protocol.
- In Project 1b you will implement a subset of [RFC2812]. We suggest you read [RFC2812 §1] and [RFC2812 §2]. For the remainder of the RFC, you should only read the sections relevant to the parts of the IRC protocol you will be implementing.
- In Project 1c you will implement a subset of the functionality described in [RFC2811], which will require implementing additional parts of [RFC2812]. We suggest you hold off on reading [RFC2811] until we reach Project 1c; if you do want to read the introductory sections, take into account that we will only be supporting "standard channels" in the "#" namespace, and that we will not be supporting server networks.
- We will not be implementing any part of [RFC2813].

Finally, you should take into account that, although IRC has an official specification, most IRC servers and clients do not conform to these RFCs. Most (if not all) servers do not implement the full specification (and even contradict it in some cases), and there are many features that are unique to specific implementations. In this project, we will produce an implementation that is partially compliant with these RFCs, and sufficiently compliant to work with some of the main IRC clients currently available.

In the remainder of this section, we will see an overview of the message format used in IRC. Then, in the next section, we will see several example communications (involving multiple messages between a client and a server).

3.1 Message format

IRC clients and servers communicate by sending plain ASCII *messages* to each other over TCP. The format of these messages is described in [RFC2812 §2.3], and can be summarized thusly:

- The IRC protocol is a *text-based* protocol, meaning that messages are encoded in plain ASCII. Although not as efficient as a pure binary format, this has the advantage of being fairly human-readable, and easy to debug just by reading the verbatim messages exchanged between clients and servers.
- A single message is a string of characters with a maximum length of 512 characters. The end of the string is denoted by a CR-LF (Carriage Return - Line Feed) pair (i.e., "`\r\n`"). There is no null terminator. The 512 character limit includes this delimiter, meaning that a message only has space for 510 useful characters.
- The IRC specification includes no provisions for supporting messages longer than 512 characters, although many servers and clients support non-standard solutions (including ignoring the 512 limit altogether). In our implementation, any message with more than 510 characters (not counting the delimiter) will be truncated, with the last two characters replaced with "`\r\n`".
- A message contains at least two parts: the command and the command parameters. There may be at most 15 parameters. The command and the parameters are all separated by a single ASCII space character. The following are examples of valid IRC messages:

```
NICK amy
WHOIS doctor
MODE amy +o
JOIN #tardis
QUIT
```

- When the last parameter is prefixed with a colon character, the value of that parameter will be the remainder of the message (including space characters). The following are examples of valid IRC messages with a "long parameter":

```
PRIVMSG rory :Hey Rory...
PRIVMSG #cmsc23300 :Hello everybody
QUIT :Done for the day, leaving
```

- Some messages also include a *prefix* before the command and the command parameters. The presence of a prefix is indicated with a single leading colon character. The prefix is used to indicate the *origin* of the message. For example, when a user sends a message to a channel, the server will forward that message to all the users in the channel, and will include a prefix to specify the user that sent that message originally. We will explain the use of prefixes in more detail in the next section.

  The following are examples of valid IRC messages with prefixes:

```
:borja!borja@polaris.cs.uchicago.edu PRIVMSG #cmsc23300 :Hello everybody
:doctor!doctor@baz.example.org QUIT :Done for the day, leaving
```

3.2 Replies

The IRC protocol includes a special type of message called a *reply*. When a client sends a command to a server, the server will send a reply (except in a few special commands where a reply should not be expected). Replies are used to acknowledge that a command was processed correctly, to indicate errors, or to provide information when the command performs a server query (e.g., asking for the list of users or channels).

A reply is a message with the following characteristics:

- It always includes a prefix.
- The command will be a three-digit code. The full list of possible replies is specified in [RFC2812 §5].
- The first parameter is always the target of the reply, typically a nick.

The following are examples of valid IRC replies:

```
:irc.example.com 001 borja :Welcome to the Internet Relay Network borja!borja@polaris.cs.uchicago.edu
:irc.example.com 433 * borja :Nickname is already in use.
:irc.example.org 332 borja #cmsc23300 :A channel for CMSC 23300 students
```

4 Example communications

In this section, we will describe three example IRC communications. Before diving into the IRC RFCs, we suggest you read through these examples to get a better sense for what a typical conversation between an IRC client and server looks like. These examples will also serve to clarify the format of messages, prefixes, and replies.

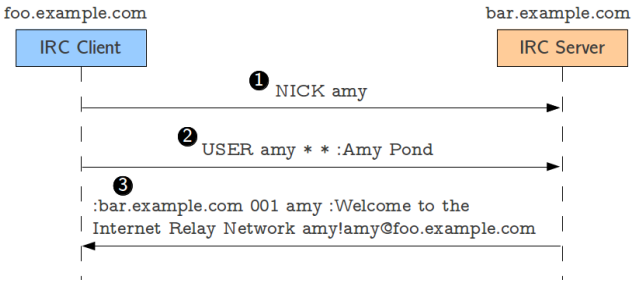

Figure 3: Connecting to an IRC server

4.1 Logging into an IRC server

When an IRC client connects to an IRC server, it must first *register* its connection. This is done by sending two messages: `NICK` and `USER` (messages 1 and 2 in Figure ??). `NICK` specifies the user's nick (`amy` in this case), and `USER` provides additional information about the user. More specifically, `USER` specifies the user's *username* (`amy`) and the user's *full name* (`Amy Pond`) (we will not be implementing the second and third parameters of `USER`). The username is typically obtained automatically by the IRC client based on the user's identity. For example, if you're logged into a UNIX machine as user `jrandom`, then most IRC clients will, by default, use that as your username. However, there is no requirement that your nick must match your username.

Assuming you've chosen a nick that is not already taken, the IRC server will send back a `RPL_WELCOME` reply (which is assigned code `001`). This reply has the following components:

- `:bar.example.com`: The prefix. Remember that prefixes are used to indicate the origin of a message. Since this reply originates in server `bar.example.com`, the prefix simply includes that hostname. This may seem redundant, given that the client presumably already knows it is connected to that server; however, in IRC networks, a reply could originate in a server other than the one the client is connected to.
- `001`: The numeric code for `RPL_WELCOME`.
- `amy`: The first parameter which, in reply messages, must always be the nick of the user this reply is intended for.
- `:Welcome to the Internet Relay Network borja!borja@polaris.cs.uchicago.edu`: The second parameter. The content of this parameter is specified in [RFC2812 §5]:

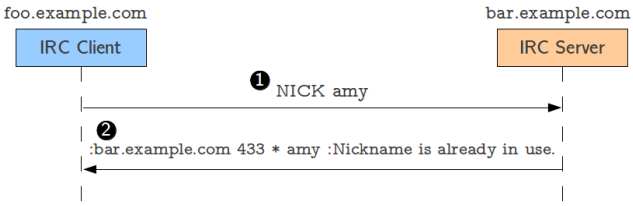

Figure 4: Connecting to an IRC server when the chosen nick is taken.

```
001    RPL_WELCOME
       "Welcome to the Internet Relay Network
        <nick>!<user>@<host>"
```

Notice how the specification of replies omits the first parameter, which is always the recipient of the reply. So, the specification lists the second and subsequent (if any) parameters.

One of the things sent back in the `RPL_WELCOME` reply is the *full client identifier* (`<nick>!<user>@<host>`), which is also used in other types of messages. It is composed of the nick as specified in the `NICK` command, the username as specified in the `USER`, and the client's hostname (if the server cannot resolve the client's hostname, the IP address is used).

Figure **??** shows a variant of the communication described above. If a user tries to register with a nick that is already taken, the server will send back a `ERR_NICKNAMEINUSE` reply (code `433`). Notice how the parameters in this reply are slightly different:

- `*`: The first parameter should be the nick of the user this reply is intended for. However, since the user does not yet have a nick, the asterisk character is used instead.
- `amy`: In the `ERR_NICKNAMEINUSE` reply, the second parameter is the "offending nick" (i.e., the nick that could not be chosen, because it is already taken).
- `:Nickname is already in use`: The third parameter simply includes a human readable description of the error. IRC clients will typically print these out verbatim.

So, notice how there is no uniform set of parameters sent back in all replies (other than the first parameter, which is always the recipient nick). When implementing a reply, you must consult [RFC2812 §5] to determine exactly what you should be sending back in the reply.

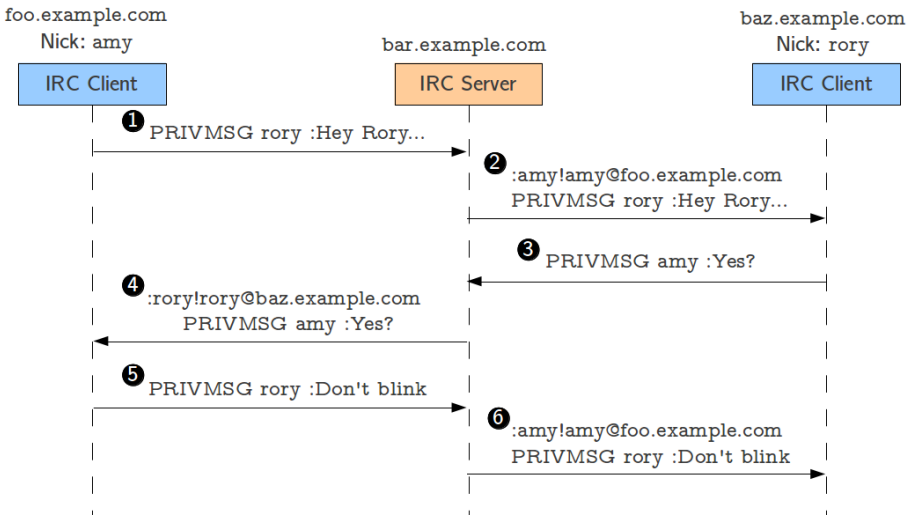

Figure 5: Sending a message to another user

4.2 Messaging between users

Once several users are connected, it is possible for them to send messages to each other, even in the absence of channels. In fact, most of Project 1b will focus on implementing messaging between users, whereas Project 1c will focus on adding support for channels.

To send a message to a specific nick, a user must send a `PRIVMSG` to the server. Figure **??** shows two users, with nicks `amy` and `rory`, exchanging three messages. In message 1, user `amy` sends a message to `rory`. The parameters for `PRIVMSG` are very simple: the first parameter is the nick of the user the message is intended for, and the second parameter is the message itself.

When the server receives that message, and assuming there is a `rory` user, it will forward the message to the IRC client that is registered with that nick. This is done in message 2, and notice how it is simply a verbatim copy of message 1, but prefixed with `amy`'s full client identifier (otherwise, the recipient IRC client would have no idea who the message was from). Messages 3 and 4 show a similar exchange, except going from `rory` to `amy`, and messages 5 and 6 show another message going from `amy` to `rory`.

Notice how all messages are *relayed* through the IRC server (hence the name of the protocol: Internet Relay Chat). Non-relayed messaging is not supported in the IRC specification, and we will not be implementing such a functionality in this project. However, there are two extensions to IRC (CTCP, the Client-to-Client Protocol, and DCC, Direct Client-to-Client) that are the *de facto* standard for non-relayed chat on IRC. Most IRC servers and clients support these extensions, even though they have never been formally specified as an RFC

(the closest thing to a specification is this document: `http://www.irchelp.org/irchelp/rfc/ctcpspec.html`).

4.3 Joining, talking in, and leaving a channel

Users connected to an IRC server can join existing channels by using the `JOIN` message. The format of the message itself is pretty simple (its only parameter is the name of the channel the user wants to join), but it results in several replies being sent not just to the user joining the channel, but also to all the users currently in the channel. Figure **??** shows what happens when user `amy` joins channel `#tardis`, where two users (`doctor` and `river`) are already present.

Message 1 is `amy`'s `JOIN` message to the server. When this message is received, the server *relays* it to the users who are already in the channel (`doctor` and `river`) to make them aware that there is a new user in the channel (messages 2a and 2b). Notice how the relayed `JOIN` is prefixed with `amy`'s full client identifier. The `JOIN` is also relayed back to `amy`, as confirmation that she successfully joined the channel.

The following messages (3, 4a, and 4b) provide `amy` with information about the channel. Message 3 is a `RPL_TOPIC` reply, providing the channel's *topic* (this is a description of the channel which can be set by certain users; we'll discuss this in detail later). Messages 4a and 4b are `RPL_NAMREPLY` and `RPL_ENDOFNAMES` replies, respectively, which tell `amy` what users are currently present in the channel. Notice how the `doctor` user has an at-sign before his nick; this indicates that `doctor` is a *channel operator* for channel `#tardis`. As we'll see in Project 1c, users can have *modes* that give them special privileges in the server or on individual channels. For example, a channel operator is typically the only type of user that can change the channel's topic.

Once a user has joined a channel, sending a message to the channel is essentially the same as sending a message to an individual user. The difference is that the server will relay the message to all the users in the channel, instead of just a single user. Figure **??** shows two messages being sent to channel `#tardis`. First, user `doctor` sends a `PRIVMSG` message, specifying the channel as the target (and not a nick, as we saw in Figure **??**). The server then relays this message to `river` and `amy`, prefixing the message with `doctor`'s full client identifier (messages 1, 2a, and 2b). Similarly, `amy` sends a message to the channel, which is relayed to `doctor` and `river`, prefixed with `amy`'s full client identifier (messages 3, 4a, and 4b)

Leaving a channel is accomplished with the `PART` message, which follows a similar pattern to joining and talking in the channel: the user wishing to leave sends a `PART` message, and this message is relayed to everyone in the channel so they are aware that the user has left. The server also internally removes that client from the channel, which means he will no longer receive any messages directed to that channel. Figure **??** shows an example of what this would look like. Notice how the `PART` message includes two parameters: the channel the users wants to leave, and a "parting message" (which is relayed as part of the `PART` message to all users in the channel).

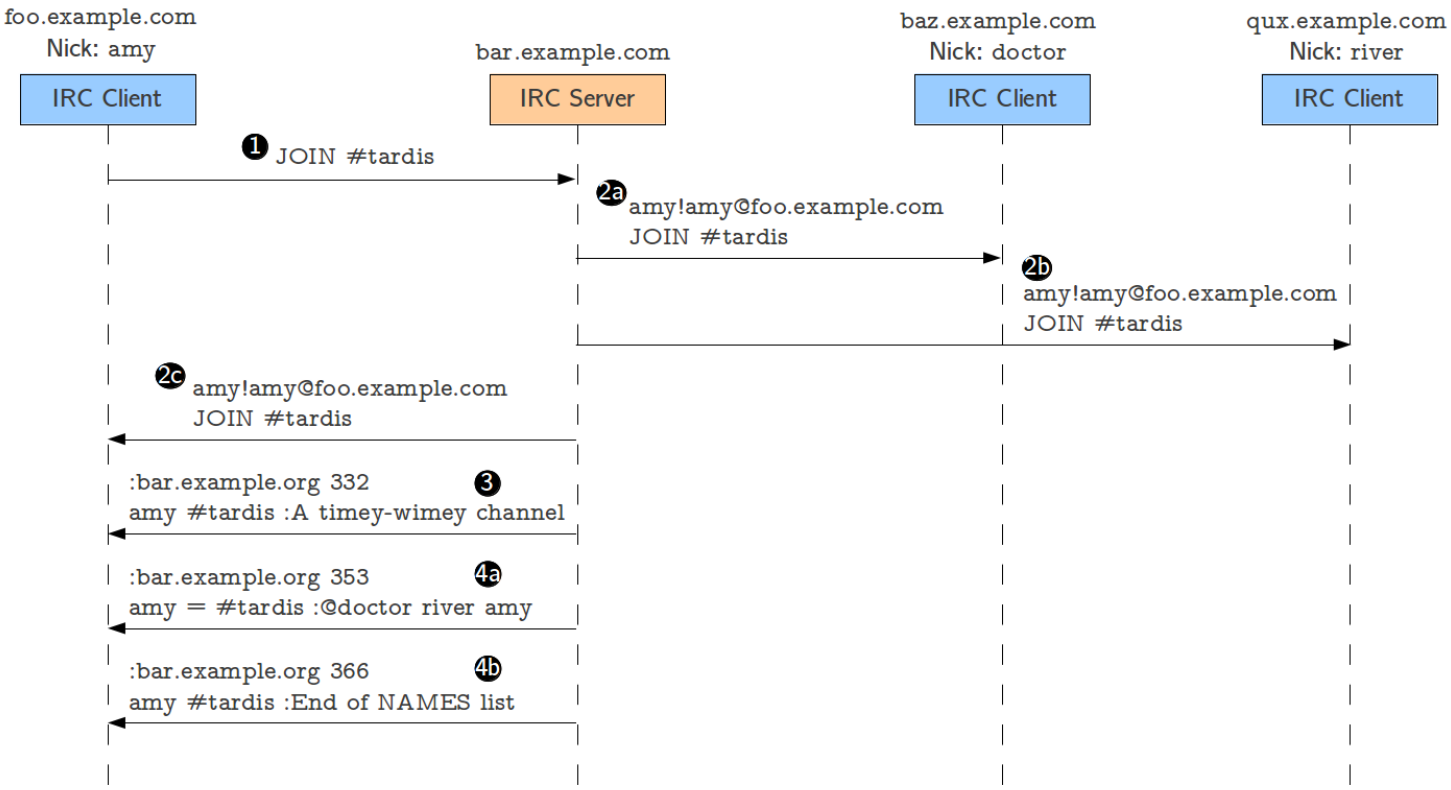

Figure 6: Joining a channel

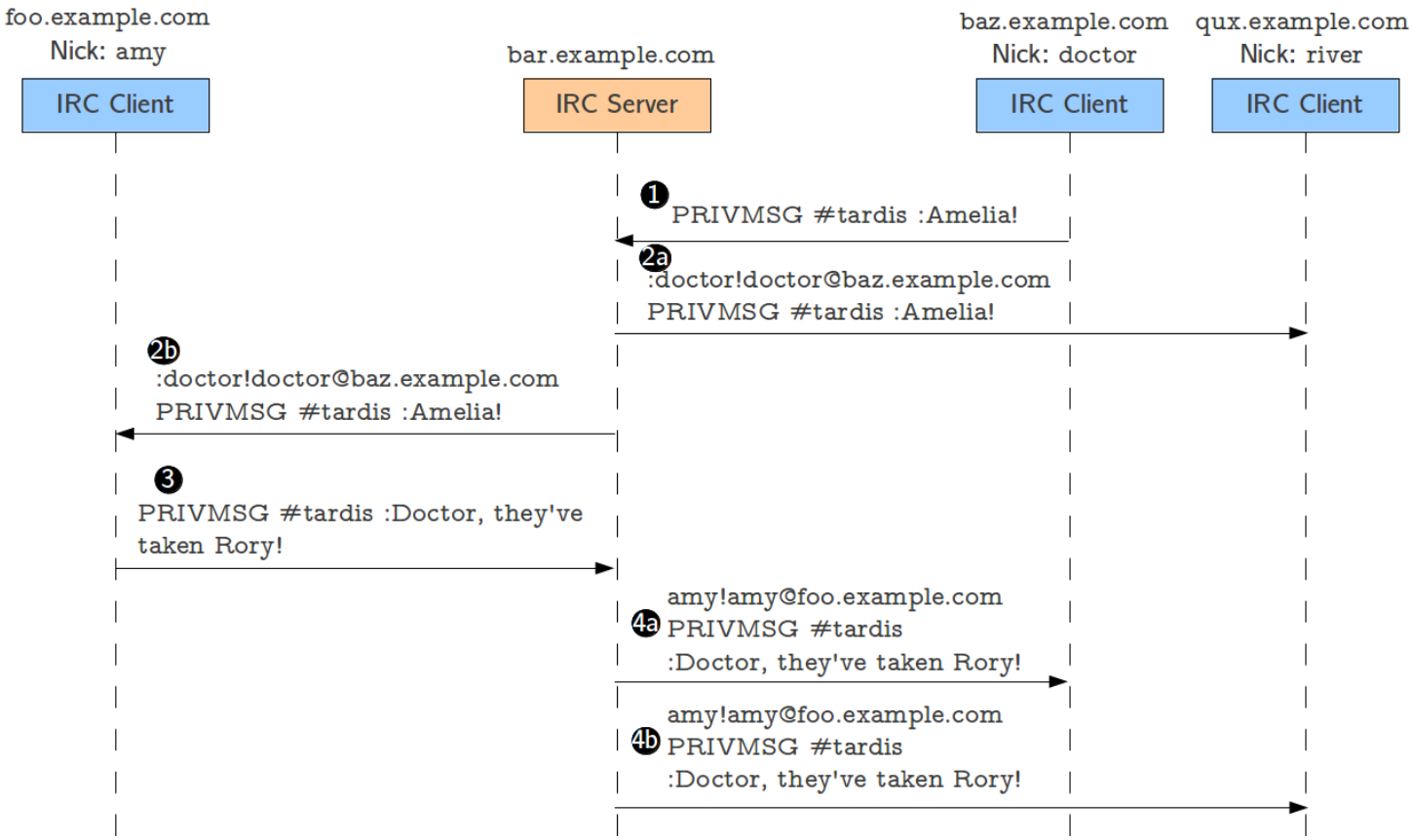

Figure 7: Talking in a channel

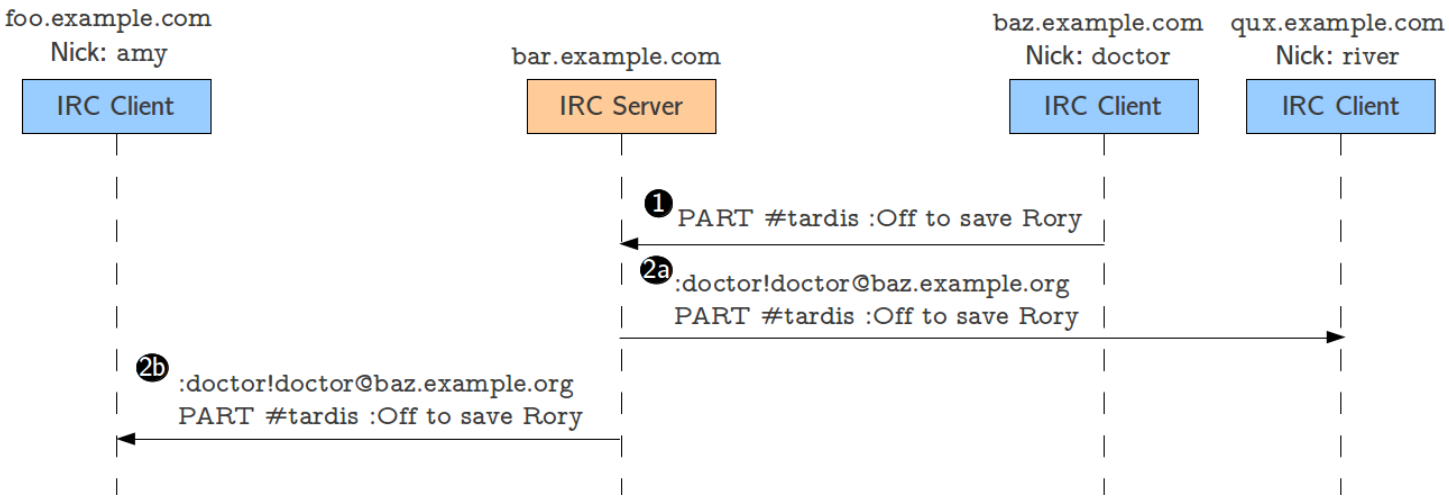

Figure 8: Leaving a channel

5 Getting the χirc source code

We are providing some basic code and Makefiles to use as a starting point in your project. It is located in the following GitHub repository:

`https://github.com/uchicago-cs/chirc`

In this course, you will be provided with a private GitHub repository where your team can work collaboratively on this project. We will also be using your GitHub repository to collect and grade your code. Your instructor will provide specific instructions on how to set up your repository and how to make project submissions.

6 Building and Testing χirc

Once you have the χirc code, you can build it simply by running Make:

```
make
```

This will generate an executable called `chirc` that accepts two parameters: `-p` and `-o`. The former is used to specify the port on which the server will listen, and the latter to specify the "operator password". You need to run the executable with at least the `-o` option, although this option will not be relevant until Project 1c. For example:

```
./chirc -o foobar
```

The provided code, however, doesn't do anything other that process the command-line parameters. You should nonetheless verify that it builds and runs correctly.

To modify the code, you should *only* add files to the `src/` directory. Take into account that, if you add additional `.c` files, you will need to modify the `src/Makefile` file so they will be included in the build (more specifically, you will need to include a new object file in the `OBJS` variable).

The provided code also includes a number of automated tests that will be used to verify the correctness of your implementation. You can run all the tests by running the following:

```
make grade
```

This output may not be too useful when you first start working on the project (and most of the tests will be failing), so you can also run only a single *category* of tests like this:

```
make grade CATEGORY=category
```

Where *category* is one of the following:

1. Project 1a
    - BASIC_CONNECTION
2. Project 1b
    - CONNECTION_REGISTRATION
    - PRIVMSG_NOTICE
    - PING_PONG
    - MOTD
    - LUSERS
    - WHOIS
    - ERR_UNKNOWN
    - ROBUST
3. Project 1c
    - CHANNEL_JOIN
    - CHANNEL_PRIVMSG_NOTICE
    - CHANNEL_PART
    - CHANNEL_TOPIC
    - MODES
    - AWAY
    - NAMES
    - LIST
    - WHO
    - UPDATE_1B

When a specific test fails in `make grade` (with or without `CATEGORY`), the test output will include the *name* of the test that is failing. For example, if you see the following:

```
======================================================================
ERROR: test_connect_basic1 (tests.test_connection.BasicConnection)
----------------------------------------------------------------------
. . .   error description   . . .
```

You can run just that test by running the following:

```
make singletest TEST=test_connect_basic1
```

Also, `make singletest` does not suppress the output of the `chirc` executable, which will allow you to see any useful logging messages before the test fails.

You can also generate an HTML report that provides a summary of successful and failed tests in a way that is easier to browse. Just run the following:

```
make htmltests
```

This will write a brief summary to the console, and a more detailed report to `report.html`.

7 Grading

For each part of the project, 50% of the project grade will be determined by the number of automated tests your solution passes successfully, as determined by running `make grade`. We will use the exact same tests provided in the upstream repository (i.e., there are no additional secret tests that only the graders have access to). So, if your solution passes 90% of the tests before the deadline, you can be certain that you will get at least 45 out of 100 points (i.e., 90% of 50).

The remaining 50% of the project grade will be determined by the "design" of your code. We will read through your code and check whether you used the appropriate data structures, synchronization primitives, etc. in the various components of your solution. In our experience, the "design" grade is typically very similar to the "tests" grade so, if you pass 90% of the tests, it is likely you will get a "design" grade in that vicinity. However, the type of feedback you get for this grade is more detailed (instead of a list of automated tests that you pass/fail).

In the specification of what you must do in each project, each part is accompanied by a number of points (e.g., «10 points»). This refers to the design points for that specific part. The amount of points for the automated tests will be roughly, but not exactly, the same (see the report generated by `make htmltests` for the exact points allocated to each automated test).

8 Tips

- *Don't be intimidated by the length of the project specification.* The main reason the remaining sections are long is to carve out exactly what part of the IRC specification you have to focus on. You will actually be implementing a fairly small subset of the specification.

- Similarly, take into account that the division of points is not proportional to the amount of effort required to obtain those points. The points are distributed in such a way that a reasonable amount of work should produce a *good* implementation that will get you roughly 75% of the points (and will place you in the B range). If you want to produce an *excellent* implementation (placing you in the A range), you will have to go the extra mile to obtain the final 25% of the points.

- Your implementation will require using data structures to store collections of users, channels, etc. We do not expect you to implement a data structure from scratch. The instructors' reference implementation uses the SimCList library (`http://mij.oltrelinux.com/devel/simclist/`), and we suggest you use it too.

- As you read the remainder of this document and the IRC specification itself, you'll notice that the same patterns come up over and over. In a sense, your server is just a piece of software that transforms IRC messages into other IRC messages (altering the state of the server in the process). So, before you start tackling individual tests, we suggest you read through the whole document and design your server in such a way that it is easy for you to (1) parse incoming messages, (2) add support for new messages, (3) manipulate the state of the server (e.g., "create a new channel", "add a new user to this channel", etc.), and (4) construct outgoing messages. Doing so can pay off handsomely later on, even if you spend the first few days feeling like you're not making any progress towards actually earning points.

- If you're unclear about how your server is meant to behave in some cases (specially the more obscure corner cases), take into account that there are literally hundreds of production IRC servers on the Internet that you can log into to test how they've interpreted the IRC specification. We suggest using Freenode servers, which you can log into simply by running:

```
telnet irc.freenode.net 6667
```

- Similarly, you can also test your implementation with an existing IRC client. We recommend using `irssi` (`http://irssi.org/`), which is installed on the CS Linux machines.

- Finally, it can sometimes be useful to take a peek at the exact messages that are being exchanged between a client and your server. You can use network sniffers like `tcpdump` and Wireshark. The console version of Wireshark, `tshark` can be useful to debug the automated tests. In particular, you can capture the traffic of a test (run with `make singletest`) by running `tshark` like this:

```
tshark -i lo \
       -d tcp.port==7776,irc -R irc -V -O irc -T fields -e irc.request -e irc.response \
       tcp port 7776
```

  Note that the automated tests use port 7776 to avoid conflicts with the default IRC port (6667), in case you have a server running separately from the tests.

  If you run the above during test `test_connect_basic1`, you should see the following:

```
NICK user1
USER user1 * * :User One
  :haddock 001 user1 :Welcome to the Internet Relay Network user1!user1@localhost.localdomain
  :haddock 002 user1 :Your host is haddock, running version chirc-0.1
  :haddock 003 user1 :This server was created 2012-01-02 13:30:04
  :haddock 004 user1 haddock chirc-0.1 ao mtov
  :haddock 251 user1 :There are 1 users and 0 services on 1 servers
  :haddock 252 user1 0 :operator(s) online
  :haddock 253 user1 0 :unknown connection(s)
  :haddock 254 user1 0 :channels formed
  :haddock 255 user1 :I have 1 clients and 1 servers
  :haddock 422 user1 :MOTD File is missing
```

  Take into account that the automated tests close the connection as soon as the test has passed, which means sometimes some messages will not be sent. For example, in this specific test, `tshark` may not capture any messages after the `001` reply.

9 Project 1a «50 points»

This first project is meant as a warm-up exercise to get reacquainted with socket programming. You must implement an IRC server that implements the `NICK` and `USER` messages only well enough to perform a *single* user registration as shown in Figure **??**. Take into account that a barely minimal server that meets these requirements, and passes the automated tests for this project, can be written in roughly 50 lines of C code (in fact, we will *give you* those 50 lines of code). However, although this kludgy solution will get you a perfect score on the tests, it will earn you a zero on the design grade.

So, you should start implementing your solution with the requirements of the rest of the project in mind. More specifically, your solution to Project 1a should meet the following requirements:

- «12.5 points» You must send the `RPL_WELCOME` *only* after the `NICK` and `USER` messages have been received.
- «25 points» You must take into account that you may get more or less than one full message when you read from a socket. You may not solve this problem by reading one character at a time from the socket.
- «12.5 points» Your solution must parse the nick and username from the `NICK` and `USER` messages, and compose the correct `RPL_WELCOME` reply.

Although not required for this project, you should take into account that the remaining two parts of the project will involve adding support for additional messages and replies. As we mentioned in Section **??**, any time you spend writing a message parser and constructor (that works with more than just `NICK` and `USER`) will be time well spent. However, if your solution to Project 1a takes some shortcuts by assuming that you will only be dealing with the `NICK` and `USER` messages and the `RPL_WELCOME` reply, you will not be penalized for it.

Your server must be implemented in C, and must use sockets. There should be no need for you to use pthreads at this point.

10 Project 1b «100 points»

In this part of the project, your main goal will be to allow users to send messages to each other (as seen in Section ??). You will also implement a couple extra messages that will make your server compliant enough to test with existing IRC clients.

Since you will be supporting multiple users, you will now have to spawn a new thread for each user that connects to your server. This, in turn, may result in race conditions in your code. You must identify the shared resources in your server, and make sure they are protected by adequate synchronization primitives.

The messages you have to implement are presented in suggested order of implementation. Nonetheless, once you've implemented Connection Registration, the remaining messages are mostly independent of each other.

10.1 Connection Registration «40 points»

Implement connection registration, as described in [RFC2812 §3.1], with the following exceptions:

- You must implement the `NICK`, `USER`, and `QUIT` messages. You must *not* implement the `PASS`, `SERVICE`, or `SQUIT` messages. You do not need to implement the `OPER` and `MODE` messages yet (you will implement them in Project 1c).
- In the `NICK` message, you are only expected implement the `ERR_NICKNAMEINUSE` reply.
- You can ignore the `<mode>` and `<unused>` parameters of the `USER` message.
- In the `USER` message, you are only expected to implement the `ERR_ALREADYREGISTRED` reply.
- After a connection has been registered, the `RPL_WELCOME` reply must be followed by the `RPL_YOURHOST`, `RPL_CREATED`, `RPL_MYINFO` replies (in that order). For the `RPL_MYINFO` reply, the user modes are `ao` and the channel modes are `mtov`.
- The `ERROR` message sent in reply to a `QUIT` must include this error message:

  `Closing Link:` *hostname* `(`*msg*`)`

  hostname is the user's hostname. *msg* is the `<Quit Message>` parameter provided in the `QUIT` message. If none is provided, the default is `Client Quit`

Take into account the following:

- The `NICK` and `USER` messages can be received in any order, and a connection is not *fully* registered until both messages have been received (and neither contain any errors)
- The `NICK` command can also be used *after* the connection registration to change a user's nick.
- You can safely skip the `QUIT` command and revisit it later, as no other commands depend on it.
- Most IRC servers send the replies corresponding to the `MOTD` and `LUSER` messages after the welcome messages are sent. Most of our tests expect this but, until you implement `MOTD` and `LUSER`, you can get away with simply sending the following replies verbatim:

```
:hostname 251 user1 :There are 1 users and 0 services on 1 servers
:hostname 252 user1 0 :operator(s) online
:hostname 253 user1 0 :unknown connection(s)
:hostname 254 user1 0 :channels formed
:hostname 255 user1 :I have 1 clients and 1 servers
:hostname 422 user1 :MOTD File is missing
```

  This will be enough to pass the connection registration tests (they check that the correct replies are sent, but don't actually check whether they contain accurate information).

10.2 `PRIVMSG` and `NOTICE` «30 points»

Implement messaging between users, as described in [RFC2812 §3.3], with the following exceptions:

- The only supported `<msgtarget>` is nicknames.
- You must only implement the `ERR_NOSUCHNICK` reply.

Take into account the following:

- If user `user1` sends a sequence of `PRIVMSG` messages to `user2`, then `user2` *must* receive them in the same order that `user1` sent them.
- If users `user1` and `user2` each send a single message to `user3`, the messages are not expected to arrive in the same order that `user1` and `user2` sent them.

10.3 `PING` and `PONG` «2.5 points»

Implement the `PING` and `PONG` commands, as described in [RFC2812 §3.7.2] and [RFC2812 §3.7.3], with the following exceptions:

- You can ignore the parameters in `PING`, and simply send the `PONG` response to the client that sent the `PING` message.
- You must silently drop any `PONG` messages you receive (do *not* send a `ERR_UNKNOWNCOMMAND` reply)

Take into account the following:

- Implementing `PING` and `PONG` is essential to testing your server with real IRC clients. IRC clients will sent `PING` messages periodically and, if they do not receive a `PONG` message back, they will close the connection.

10.4 `MOTD` «5 points»

Implement the `MOTD` command, as described in [RFC2812 §3.4.1], with the following exceptions:

- You can ignore the `<target>` parameter.

Take into account the following:

- Your server should read the "Message Of The Day" from a file called `motd.txt` in the directory from where you ran the server.
- If the file does not exist, you must return a `ERR_NOMOTD` reply.

10.5 `LUSERS` «10 points»

Implement the `LUSERS` command, as described in [RFC2812 §3.4.2], with the following exceptions:

- You can ignore the `<mask>` and `<target>` parameters.
- You must return the replies in the following order: `RPL_LUSERCLIENT`, `RPL_LUSEROP`, `RPL_LUSERUNKNOWN`, `RPL_LUSERCHANNELS`, `RPL_LUSERME`
- You do not need to support the `ERR_NOSUCHSERVER` reply

Take into account the following:

- You must send the replies even when they are reporting a zero value (i.e., ignore this from [RFC2812 §5.1]: "When replying, a server MUST send back RPL_LUSERCLIENT and RPL_LUSERME. The other replies are only sent back if a non-zero count is found for them.")

- An "unknown connection" is any connected client that isn't fully registered (i.e., any client that hasn't successfully sent `NICK` and `USER`).
- The number of users in the `RPL_LUSERCLIENT` reply is the number of registered users (i.e., all open connections, minus unknown connections).
- The number of clients in the `RPL_LUSERME` reply is the total number of connections, including unknown connections.

10.6 `WHOIS` «10 points»

Implement the `WHOIS` command, as described in [RFC2812 §3.6.2], with the following exceptions:

- The command must accept a single parameter: a nick (i.e., there is only a single `<mask>`, and it must be a nick; ignore the `<target>` parameter)
- You must only send back the following replies, in this order: `RPL_WHOISUSER`, `RPL_WHOISSERVER`, `RPL_ENDOFWHOIS`.
- You must supply a value for parameter `<server info>` in `RPL_WHOISSERVER`, but we won't be checking its contents.
- You must support the `ERR_NOSUCHNICK` reply.

Take into account the following:

- You will be implementing `RPL_WHOISOPERATOR`, `RPL_WHOISCHANNELS`, and `RPL_AWAY` in Project 1c.

10.7 `ERR_UNKNOWNCOMMAND` «2.5 points»

If your server receives any message not described here (or in Project 1c), you must return a `ERR_UNKNOWNCOMMAND` reply.

11 Project 1c «100 points»

In this part of the project, your main goal will be to add support for channels and modes. You will now have to deal with the fact that messages may be relayed to multiple users, sometimes across multiple channels.

The parts of this project are presented in suggested order of implementation. Nonetheless, once you've implemented channels (`JOIN`, `PART`, and sending messages to channels), implementing modes and the remaining messages are all fairly independent of each other.

11.1 `JOIN` «15 points»

Implement the `JOIN` command, as described in [RFC2812 §3.2.1], with the following exceptions:

- The command must accept a single parameter: a channel name.
- You must only support the `RPL_TOPIC` and `RPL_NAMREPLY` replies.

Take into account the following:

- You must only send a `RPL_TOPIC` reply if the channel has a topic (you will implement the `TOPIC` message later). Otherwise, that reply is skipped.
- Although not stated explicitly in [RFC2812 §3.2.1], the `RPL_NAMREPLY` reply must be followed by a `RPL_ENDOFNAMES` (as shown in Figure ??). Basically, you are sending the same replies generated when a `NAMES` message (with this channel as a parameter) is received.
- The first automated tests for `JOIN` will check that the `RPL_NAMREPLY` and `RPL_ENDOFNAMES` replies are sent, but won't validate their contents. So, you can get away with just sending the following replies (substituting ***nick*** with the recipient nick):

```
:hostname 353 nick = #foobar :foobar1 foobar2 foobar3
:hostname 366 nick #foobar :End of NAMES list
```

  Once you implement the `NAMES` message, you can simply substitute this with a call to the same code that handles the `NAMES` message. Note that, until you implement `NAMES` correctly, most IRC clients will show your channels as having the three users in your hardcoded `NAMES` reply (`foobar1`, `foobar2`, and `foobar3`)

11.2 PRIVMSG and NOTICE to channels «15 points»

Extend your implementation of `PRIVMSG` and `NOTICE` from Project 1b to support sending messages to a channel.

Until you implement modes, you will not need to support any additional replies in `PRIVMSG`. However, take into account the following:

- Despite its name the `ERR_NOSUCHNICK` is also the appropriate reply when a non-existent channel is specified.
- Users cannot send `PRIVMSG` and `NOTICE` messages to channels they have not joined. When this happens, a `ERR_CANNOTSENDTOCHAN` reply must be sent back (only in the case of `PRIVMSG` messages).
- Once you have implement modes, there may be additional cases where a message will be denied if the user has insufficient privileges to speak on a channel.

11.3 PART «10 points»

Implement the `PART` command, as described in [RFC2812 §3.2.2], with the following exceptions:

- The command must accept either one parameter (a channel name) or two parameters (a channel name and a parting message)
- You must only support the `ERR_NOTONCHANNEL` and `ERR_NOSUCHCHANNEL` replies.

Take into account the following:

- Once all users in a channel have left that channel, the channel must be destroyed.

11.4 TOPIC «10 points»

Implement the `TOPIC` command, as described in [RFC2812 §3.2.4], with the following exceptions:

- You only need to support the `ERR_NOTONCHANNEL`, `RPL_NOTOPIC`, and `RPL_TOPIC` replies.
- You will not need to support the `ERR_CHANOPRIVSNEEDED` reply until you implement modes.

11.5 User and channel modes «25 points»

In IRC, users can have certain *modes* assigned to them. Modes are identified by a single letter, and they are binary: a user either has a mode, or he doesn't. The possible user modes are described in [RFC2812 §3.1.5], and we will be implementing only the following modes:

`a` - The *away* mode. Users with this mode are considered to be "away from IRC". Besides being displayed as such on an IRC client, it will also affect how certain messages directed to a user will be processed.

`o` - The *operator* mode. Users with this mode have administrative privileges on the IRC server, and have access to certain commands available only to operators.

The above two modes are global modes: they have effect across the entire server. Users can also have channel-specific modes (or *member status* modes, see [RFC2811 §4.1]). We will be implementing the following member status modes:

`o` - The *channel operator* mode. Users with this mode on a channel have special privileges on that channel.

`v` - The *voice* mode. Users with this mode are able to send messages to moderated channels (described below).

Finally, channels themselves can also have modes (see [RFC2811 §4]). We will be implementing the following modes:

`m` - The *moderated* mode. When a channel has this mode, only certain users are allowed to send messages to the channel.

`t` - The *topic* mode. When a channel has this mode, only a channel operator can set the channel's topic.

These modes are managed with the `OPER`, `MODE`, and `AWAY` commands. For now, we will focus on the first two.

You must implement the `OPER` message as described in [RFC2812 §3.1.4], with the following exceptions:

- You must only support the `RPL_YOUREOPER` and `ERR_PASSWDMISMATCH`.

Take into account that you should expect a `<user>` parameter but will ignore its content; the password expected by the `OPER` command is the one specified in the `-o` command-line parameter to the `chirc` executable.

You must implement the `MODE` message as described in [RFC2812 §3.1.5] (for user modes) and [RFC2812 §3.2.3] (for member status and channel modes), with the following exceptions:

- For user modes:

  - You only need to support two (and only two) parameters: the nick and the mode string. The mode string will always be two characters long: a plus or minus character, followed by a letter.
  - You only need to support the `ERR_UMODEUNKNOWNFLAG` and `ERR_USERSDONTMATCH` replies.
  - If there are no errors, the reply to the `MODE` message will be a relay of the message, prefixed by the user's nick and with the mode string in a long parameter. So, if a user sends this message:

    ```
    MODE jrandom -o
    ```

    The reply should be:

    ```
    :jrandom MODE jrandom :-o
    ```

- For channel modes:
  - When only a single parameter (a channel name) is used, the only error condition you must support is the `ERR_NOSUCHCHANNEL` reply (although this is not included in the specification for `MODE`). If the command is successful, return a `RPL_CHANNELMODEIS` reply (in this reply, the `<mode>` parameter must be a plus sign followed by the channel modes; you must omit the `<mode params>` parameter).
  - When two parameters (a channel name and a mode string) are used, you must support the following error replies: `ERR_NOSUCHCHANNEL`, `ERR_CHANOPRIVSNEEDED`, and `ERR_UNKNOWNMODE`. If the command is successful, the message is relayed back to the user and to all the users in the channel.
- For member status modes:
  - You only need to support three parameters: the channel, the mode string, and the nick.
  - You must support the following error replies: `ERR_NOSUCHCHANNEL`, `ERR_CHANOPRIVSNEEDED`, `ERR_UNKNOWNMODE`, and `ERR_USERNOTINCHANNEL`.
  - If the command is successful, the message is relayed back to the user and to all the users in the channel.

You must observe the following rules when dealing with modes:

- The `OPER` message is the *only* way for a user to gain operator status (the `o` user mode). As indicated in the specification, a request for `+o` by a non-operator should be ignored.

- The `a` user mode cannot be toggled using the `MODE` command. Only the `AWAY` message can manipulate that mode. Requests to change it should be ignored.
- When a channel is created (when the first user enters that channel), that user is automatically granted the channel operator mode.
- In a channel, only a channel operator can change the channel modes.
- In a channel, only a channel operator can change the member status modes of users in that channel.
- When a channel has the `m` mode, only channel operators and users with the `v` member status can send `PRIVMSG` and `NOTICE` messages to that channel. Other users will receive an `ERR_CANNOTSENDTOCHAN` reply.
- When a channel has the `t` mode, only channel operators can change the channel's topic. Other users will receive a `ERR_CHANOPRIVSNEEDED` reply.
- In terms of permissions, server operators (i.e., with user mode `o`) are assumed to have the same privileges as a channel operator. However, a server operator *does not* explicitly receive the `o` member status upon joining a channel (the user will simply have, implicitly, the same privileges as a channel operator).

11.6 `AWAY` «5 points»

Implement the `AWAY` command, as described in [RFC2812 §4.1].

11.7 `NAMES` «5 points»

Implement the `NAMES` command, as described in [RFC2812 §3.2.5], with the following exceptions:

- We are not supporting invisible, private, or secret channels, so you can consider that all channels are visible to a user sending the `NAMES` command.
- You only need to support `NAMES` messages with no parameters or with a single parameter.
  - When no parameters are specified, you must return a `RPL_NAMREPLY` reply for each channel. Since we are not supporting invisible users, the final `RPL_NAMREPLY` must include the names of all the users who are not on any channel. If all connected users are in a channel, this final `RPL_NAMREPLY` is omitted.
  - When a single parameters is specified, that parameter is interpreted to be a channel.

- You do not need to support the `ERR_TOOMANYMATCHES` and `ERR_NOSUCHSERVER` replies.

Take into account the following:

- Channels and nicks do not need to be listed in any specific order.
- When you implement modes, nicks with channel operator privileges on a channel must have their nick prefixed by @ in the `RPL_NAMREPLY` reply. Similarly, nicks with "voice" privileges must have their nick prefixed by +.

11.8 `LIST` «5 points»

Implement the `LIST` command, as described in [RFC2812 §3.2.6], with the following exceptions:

- You only need to support `LIST` messages with no parameters (list all channels) or with a single parameter (list only the specified channel).
- You do not need to support the `ERR_TOOMANYMATCHES` and `ERR_NOSUCHSERVER` replies.

Take into account the following:

- Channels do not need to be listed in any specific order.
- In the `RPL_LIST` reply, the `<# visible>` refers to the total number of users on that channel (since we are not supporting invisible users, the number of visible users equals the total number of users in the channel).

11.9 `WHO` «5 points»

Implement the `WHO` command, as described in [RFC2812 §3.6.1], with the following exceptions:

- If a mask is specified, you only need to support the case where the mask is the name of a channel. If such channel exists, you must return a `RPL_WHOREPLY` for each user in that channel.
- We are not supporting invisible clients so, if no mask is specified (or if `0` or `*` is specified as a mask), you must return a `RPL_WHOREPLY` for each user in the server that doesn't have a common channel with the requesting client.
- You do not need to support the `o` parameter.
- You do not need to support the `ERR_NOSUCHSERVER` reply.

Take into account the following:

- When a channel is not specified, the `<channel>` field in the `RPL_WHOREPLY` reply must be set to `*`.
- In the `RPL_WHOREPLY` reply, the `<hopcount>` should be hardcoded to `0` (zero).
- The `RPL_WHOREPLY` must return a series of flags, which is specified as `( "H" / "G" > ["*"] [ ( "@" / "+" ) ]` without explanation (furthermore, the `>` is a typo, and should be a right parenthesis). The flags must be constructed thusly, in this order:
  - If the user is not away, include `H` ("here")
  - If the user is away, include `G` ("gone")
  - If the user is an operator, include `*`
  - If the user is a channel operator, include `@`
  - If the user has the voice mode in the channel, include `+`

  When a channel is not specified, the `@` and `+` flags are not included (regardless of what channel modes that user may have in the users he belongs to).

11.10 Updating commands from Project 1b «5 points»

Update the implementation of the following commands:

- `NICK`: When a user sends this message, and the change of nick is successful, it must be relayed to all the channels that user is in.
- `QUIT`: When a user sends this message, it must be relayed to all the channels that user is in. Take into account that a `QUIT` results in that user leaving all the channels he is in.
- `WHOIS`: Add support for the `RPL_WHOISOPERATOR`, `RPL_WHOISCHANNELS`, and `RPL_AWAY` replies. These are only sent if the user is an IRC operator, on at least one channel, or away, respectively. The order of all the replies will be: `RPL_WHOISUSER`, `RPL_WHOISCHANNELS`, `RPL_WHOISSERVER`, `RPL_AWAY`, `RPL_WHOISOPERATOR`, `RPL_ENDOFWHOIS`.