Colin Clemmons

CS240H

ANSI-Terminal IRC in Haskell

My project was to implement something like an Internet Relay Chat server in Haskell. Given my lack of experience in networking and concurrency, I chose to simplify my task by writing a server that worked with the already existing and common netcat and ANSI terminals rather than following the IRC daemon standard. I ended up with a functional chat server that easily handles many users, though I have not been stress tested, but the lightweight nature of Haskell threads and text only messaging makes for a lightweight fully functional chat server that can connect many users with a low hardware requirement.

In writing the project, I made three main modules, the first is Lib.hs which lays all of the groundwork for the user-server interaction. It defines the User, Server, and Msg (message) data types as well as provides a type alias for rooms. A User consists of a username, a handle for interaction with the user, a mutable variable of their current active room, a mutable variable of their current joined rooms, a mutable variable for their window information, and a reference to the server metadata. A room consists of two parts, a name and an STM.TChan. The channel is duplicated for each user. A Rooms is a map from bytestring names to the room's channel. A Server is a mutable map of Rooms and a mutable set of users. A message consists of the sending user, the room name it is sent to, and timestamp of when it was received.

Lib.hs also provides the main barebones functionality for the server. It provides mainapp which starts the server, opening a given port, initializing the server's default room, and then accepting new users as they connect. It provides the userHandler which gets the necessary information from the user on their name, their terminal, and then repeats receiving messages from the client and sending them to the client until the client terminates their session either by using the ":quit" command or otherwise, at which point the thread tears disposes of itself.

NetHandleUI.hs provides the user interface API for the application. It defines the window type which is a width, height, and the handle used for interaction. This is redundant when paired with the user type, but useful. It extends the functionality of the ansi-terminal library to include the ANSI code that prints the cursor location, This allows mostly automated terminal sizing information. Finally, it provides an API for ease of use with Lib.hs, newInput puts the cursor in the appropriate place, sendBuilderOffset sends a message with a number of lines it needs to scroll to place it in the appropriate place, and linesInput calculates the number of lines a bytestring is given the sizing of the window. A difficulty in this was getting it to work with the default Mac terminal (and the default Linux terminal). I initially tested only on iTerm2, which handles newlines and ANSI scrolling codes slightly differently. I am not completely sure what the difference is, but what caused the messages to show up stacked in iTerm2 caused them to erase each other in the Mac terminal. This was remedied by using an ANSI scroll up in combination with a newline at the end of the message to appropriately handle both cases to the best of my ability.

Parsers.hs uses attoparsec with bytestrings to create a small API to parse information for the rest of the program, it parses users' commands to a command type, parses the cursor location information from the ANSI window, and a username.

Server side functionality is as follows. The majority of the application is written in the src/* files, app/Main.hs is small and consists of running the application on a given port constant. To change the port the server binds to requires only a simple change of that number. The application runs completely in memory, this means that only a minimal disk is required, logs come only in the form of what errors Haskell mentions. As far as I am aware, the server should be able to run indefinitely, there should be no memory leaks, threads are always properly closed and the user's little user metadata is disposed of. Due to the lightweight nature of Haskell threads, and the lack of bound threads, the performance is bound only by the number of threads can be run concurrently on the machine. The timestamp of the message ensures that the order of the messages is consistent across all users.

Client functionality is as follows: the user types a message that appears in the console along the bottom of the screen, pressing enter when they are done composing, the message is then sent to all subscribers of the user's active room. The message then pushes the other received messages up and is appended to the bottom of the stack of messages. This allows the user to see their history of received messages and their messages in that progression. Each message is preceded by a header with information about that message: "<*user@roomname hh:mm*> where the italics are replaced with the true values. The user also has a set of commands that they can call with ":*command*", the commands are as follows. "quit" quits the chat room, closes the connection to the user and announces their exit while saying goodbye to the user. "clear" clears the user's terminal window. "resize" should be called when the user resizes their terminal for a proper user interface. "users" prints a list of the users online at the moment. "help" displays a help message that is the same as the one shown on entrance. Then there are the room commands that facilitate movement between rooms and management of sending and receiving. "where" sends the user a system message telling them their current active room. "switch *roomname*" switches the user's active room to *roomname*. "join *roomname*" subscribes the user to the room *roomname*. "leave *roomname*" unsubscribes the user from the room *roomname*.

Overall, there are few bugs that I am aware of, but I know that a connection that times out does not close properly on the server side, but killing netcat on the user side with ^c or closing the handle by sending EOF (^d) does lead to a properly closed user thread. In killing the user, all of their information should be garbage collected, but there is the possibility that the duped TChan (which as I understand is just a pointer) may stay uncollected and be problematic, as it would allow the buildup of unread messages in memory. The final possible issue is that unused rooms are not ever deleted, but there should never be any duplication, so whether it is an issue is questionable; the safer choice would be to have a pre-set list of rooms, but I did not want to limit the program in that way, and implementing a full admin system where administrators can create or delete rooms would have been too much more work. A potential for future development would be to create a user side application or server side library (used with handles) that provides a curses like API for UI over the network in Haskell.