Processing Relay Chat – PRC

David Chen – Team PLA (Pretty Lazy Acronym)

Period 6

1 Description

Processing Relay Chat (PRC) encompasses a client program, server program, and protocol for instant messaging over the network. To accomplish this in Processing PRC the Network library.
PRC will include the following features:

- Ephemeral plaintext (printable ASCII)[1] messaging
- Usernames with deterministic colors
- A slash command framework for advanced user interaction
- File sharing
- Chat log export

1.1 Protocol

1.1.1 Packet Format

PRC uses a JSON-inspired TCP packet schema, in which ASCII separator characters are used, simplifying the burden of packet parsing. Bytes are denoted here using 3-digit octal notation.

Non-printable ASCII, save for the specified special characters, are not permitted in a PRC packet. Row names and data fields may include arbitrary printable ASCII strings (characters ranging from `\040` SPACE to `\176` ~).

Each row takes the form of `NAME\037DATA`, with `\037` being the "unit separator" byte, analogous to `:` in JSON. A row can be terminated with the "record separator" character `\036`, analogous to `,` in JSON, to indicate that there is another record to parse, or with `\003`, the "end of text" byte, to terminate parsing. Each row should be processed by the program as a String to String hashmap, with the NAME being the key, and the DATA being the value.

A packet will NOT be parsed until a terminating "end of text" byte is detected.

1.1.2 Commands

By convention, commands are represented with a four or five letter, capitalized short string, though arbitrary capitalization, spacing, or length is possible. The command is specified in a PRC packet row named `Command`.

NAME is the username registration command. The client should provide a `UUID` field, which will be echoed in the response packet, in order to discern whether it is recieving its own name. The client provides the field `User` to describe the requested username, and `Old User` if the client already had a registered username. The server will validate that the username does NOT include `#`, and assign a username of `GuestN`, with N being the number of already registered users, if there is an invalid `User` field. The server will then echo the packet, but with the final, published `User` value.

SEND is the message send command. The fields `User`, `Content`, and `Channel` are required from the user, and the server will specify `HOST` based on the client's IP address.

JOIN is the channel join command. The `Channel` field is required, and the server will validate that the channel name does NOT include `#`, and is of appropriate length (18). If the channel has not been created yet, the server will add a new channel to its listing. The server will respond with a `SYNC` packet.

[1] Due to font limitations, Unicode glyphs for other languages/emojis cannot be fully supported.

SYNC is the state synchronization packet. This packet, when sent by the client (or, `CHAN` for backwards compatibility), will request the server to broadcast a `SYNC` packet. The packet contains two fields, `Channels` and `Users`, with `#` to deliminate entries.

QUIT is the quit command. When a client sends this command to the server, it is requesting a graceful exit, which the server will perform, then send a terminating `QUIT` packet. When a client receives this packet, it will gracefully exit. The server will also send this packet before a server shutdown.

ERROR is the error packet. The client should never send this packet. The server will send this packet to the client when the client has sent a complete packet containing user errors, and the client should display the packet's contents to the user.

2 Functionality

As of right now, message sharing, user color generation, and slash commands are complete. Chat log export and file sharing are to be completed.

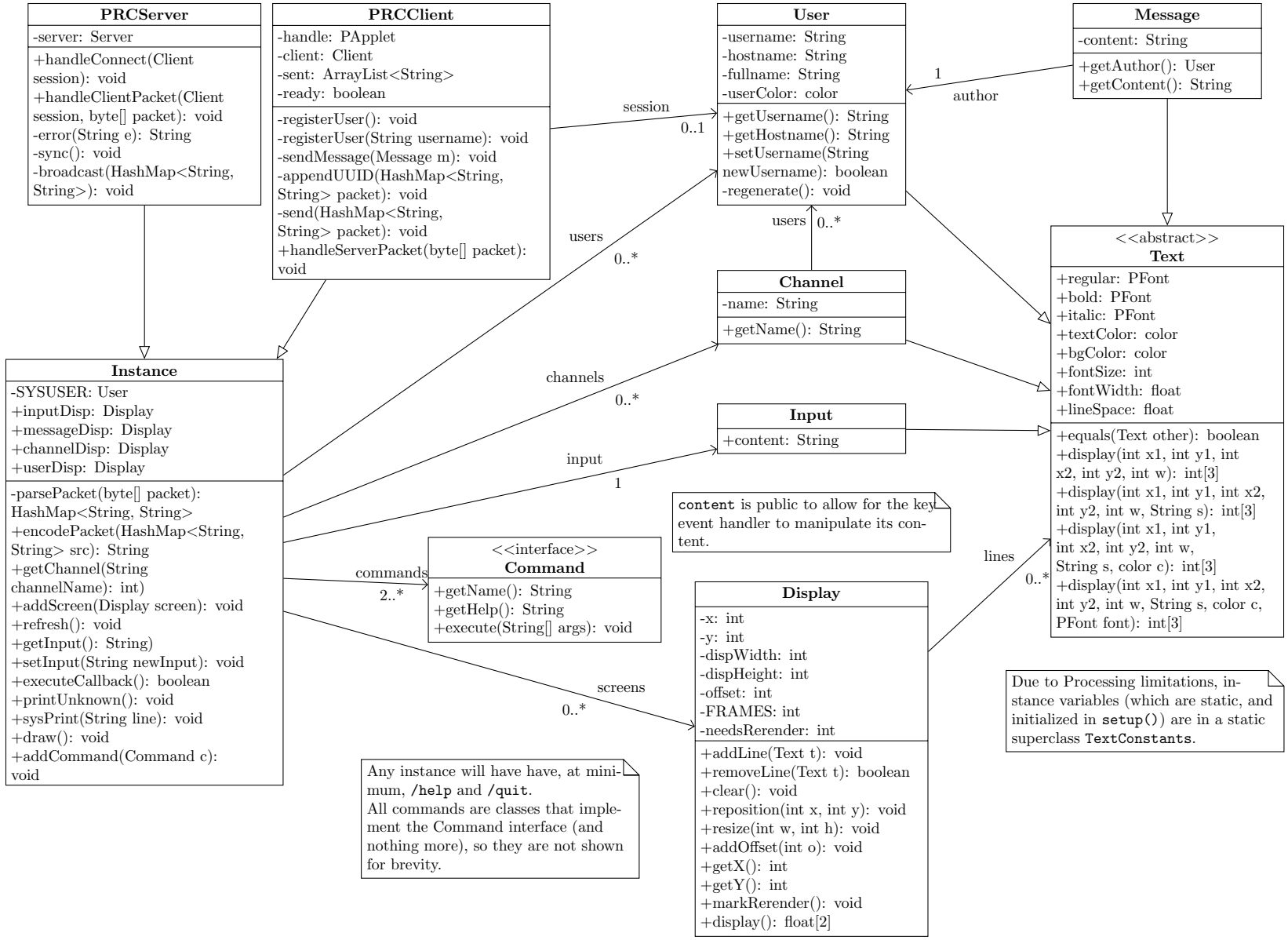
2.1 Issues

1. To share font data among `Text` subclasses, it was appropriate to use `static` variables, but there may be "non-`static` inner type" issues when trying to declare `static` variables in the `Text` abstract class. To fix this, the abstract class itself must be declared `static`. (Source)
2. In testing, it appears that a Client and a Server running in the same PApplet may lead to unexpected behavior. Thus, it will be necessary to run the server separately from the client.
3. The Display class had a tendency for displayed lines to overflow the boundaries of the screen. The solution is to use the bounding box features in Processing's `text()` function to cut off lines as needed, and additional logic to prevent attempted display of additional lines outside of the boundary. To avoid issues with losing message data, it will be possible for the message box to scroll.
4. The programs would behave erratically if there was a failure to connect to the server or bind to a valid port. To fix this, we can add additional error checks that will attempt to automatically resolve the issue, or prompt the user for corrections.

3 Log

- 05/20: Began writing program specifications in LaTeX.
- 05/21 – 05/22: Constructed UML class diagrams for anticipated object classes.
- 05/23 – 05/24: Corrected text location alignment issues.
- 05/25: Implemented Text subclasses and display() functions; turned Text into an abstract base class to deduplicate logic; Implemented Display and constructed a testing mockup.
- 05/26 – 05/28: Adjusted UML class designs to provide additional necessary functionality. Began working on network transmission of messages.
- 05/29 – 06/02: Worked around rendering bug that prevented messages from instantly appearing on the screen. Began pondering design and protocol alterations for separate Client/Server classes, along with proper maintainance of message, channel, and user states.
- 06/03 – 06/08: Implemented the network protocol and state-sharing functionality between Client and Server, fixing bugs and pitfalls along the way. Merged the `overflow` branch (from the previously logged step) and implemented scroll buffer features to enable a cohesive user experience. Reassessed scope to fit time frame. Implemented Catpuccin colors anyways because they're neat :)
- 06/09: Cleaned up code for improved organization and cohesion in design, updated and rearranged UML diagram from the umpteenth (and hopefully final) time to reflect current classes, documented protocol in detail, began writing plot for planned demonstration skit.
- 06/10 (planned): implementation of final features; script-writing for the demonstration, which I hope to integrate a skit into the demonstration portion, with a broad overview of the code structure, pitfalls along the way, and my personal favorite functions.

4 UML Diagram



5 Manual

PRC encompasses two separate programs, a Server and Client program. In both programs, `/help` will print out a description of available slash-commands when they are available.

5.1 Server Mode

The server will attempt to start a TCP server on port 2510. If this fails, it will increment upward until a working port is found. If the program fails to find a port, it will exit().

The program will then display an entry screen from which the administrator can use the `/quit` command to disconnect all clients. It will also list the usernames given by user logins, along with the channels that have been created. If the DEBUG boolean is enabled, additional logs will be displayed on screen.

5.2 Client Mode

The user will be prompted enter a server address/port to connect to, in the `host:port` format. On successful connection, the user will be registered to a guest username, and join the `#general` chat by default.

Using the `/nick` command, the user can register a username, and using the `/join` command, they are able to change to a different channel, or create their own. Like the server, their screen will display a list of connected users and channels. They can use `/quit` to gracefully exit from the program.

User colors are generated from a palette using a deterministic algorithm that guarantees that the same user with the same nickname on the same host IP address will have the same color as displayed in chat.

Once logged into a channel, the user is allowed to enter their message into the bottom chatbox, and press enter to send it to the channel. This message will only appear when the user has `/switch`ed to the correct channel. Messages in other channels are hidden until the user switches to them.

Using the up and down arrow keys, the user may scroll through chat history, akin to pagers like the `less` command in many Unix environments.