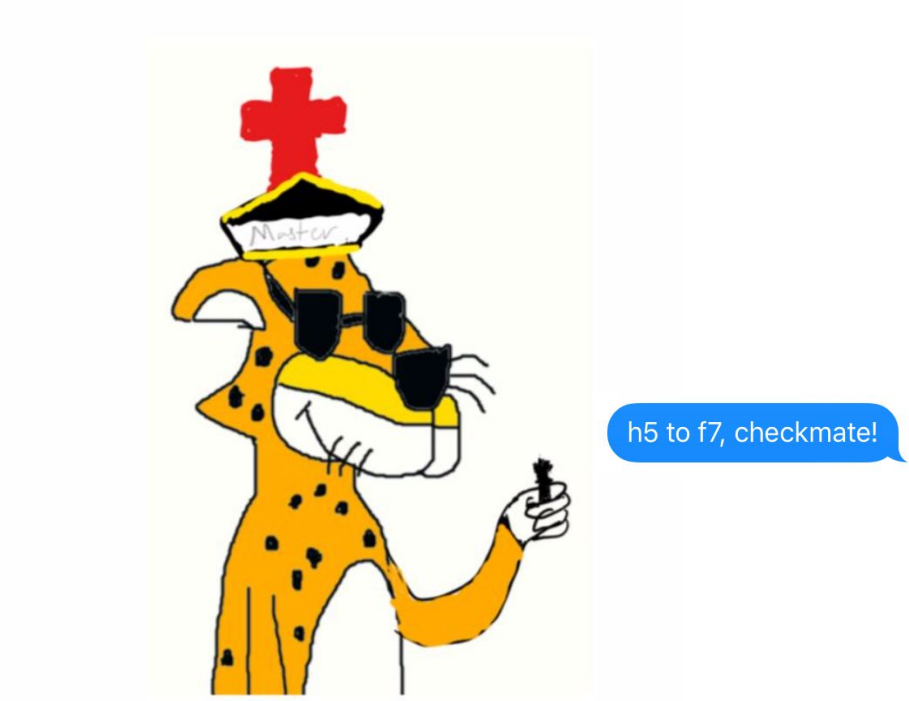


Chesster Chat

Software Specification Manual

An instant messaging application with an integrated chess game

Created by:

Jada Berenguer, *team manager*
Benny Lin, *presenter*
Marvis Nguyen, *recorder*
Daniel Pajulas, *recorder*
Armand Ahadi-Sarkani, *reflector*
Kevin Selda, *reflector*
Raiyan Bin Nasim, *reflector*

All affiliated project contributors are 2nd year computer engineering students at the University of California, Irvine.

Table of Contents

Glossary

Array: A data structure containing an amount of elements, each having an index in which it can be accessed.

Call: To use a function.

Client: The user that initiates a message by sending a request to the server.

Data type: A data item that is used to declare variables or functions. The type of data determines how much space is used in memory storage

Function/API: A group of statements created together to perform a certain task.

Initialize: To assign a value to a data object or variable.

Module: A single unit of source code that can be used with other units of source code.

Pointer: A variable that stores (points) to a memory address of another variable. A pointer can be used to allocate memory dynamically.

Structure: A structure is a user defined data type that can group different data types into one.

Server: The user that handles the client's request and passes that message to another user.

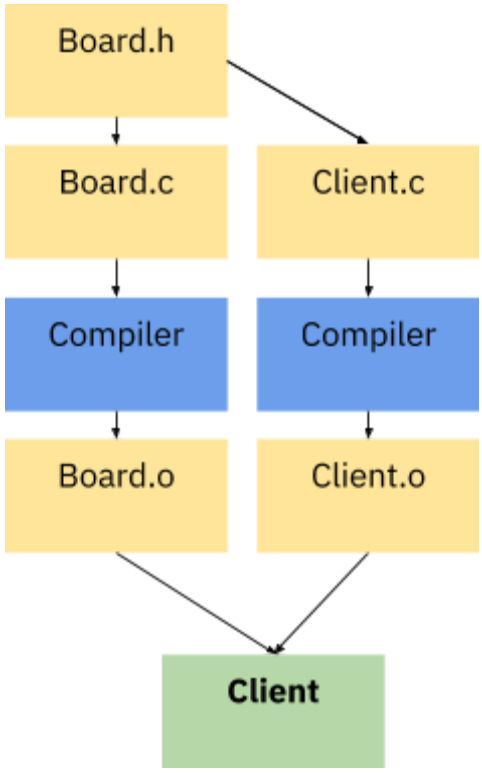
Variable: A variable is a specific stored space that the program manipulates.

1. Client Software Architecture Overview

1.1. Main data types and structures

The **User structure** includes information about the client's username, password, and friend list.

1.2. Major software components



1.3. Module interfaces

Modules:

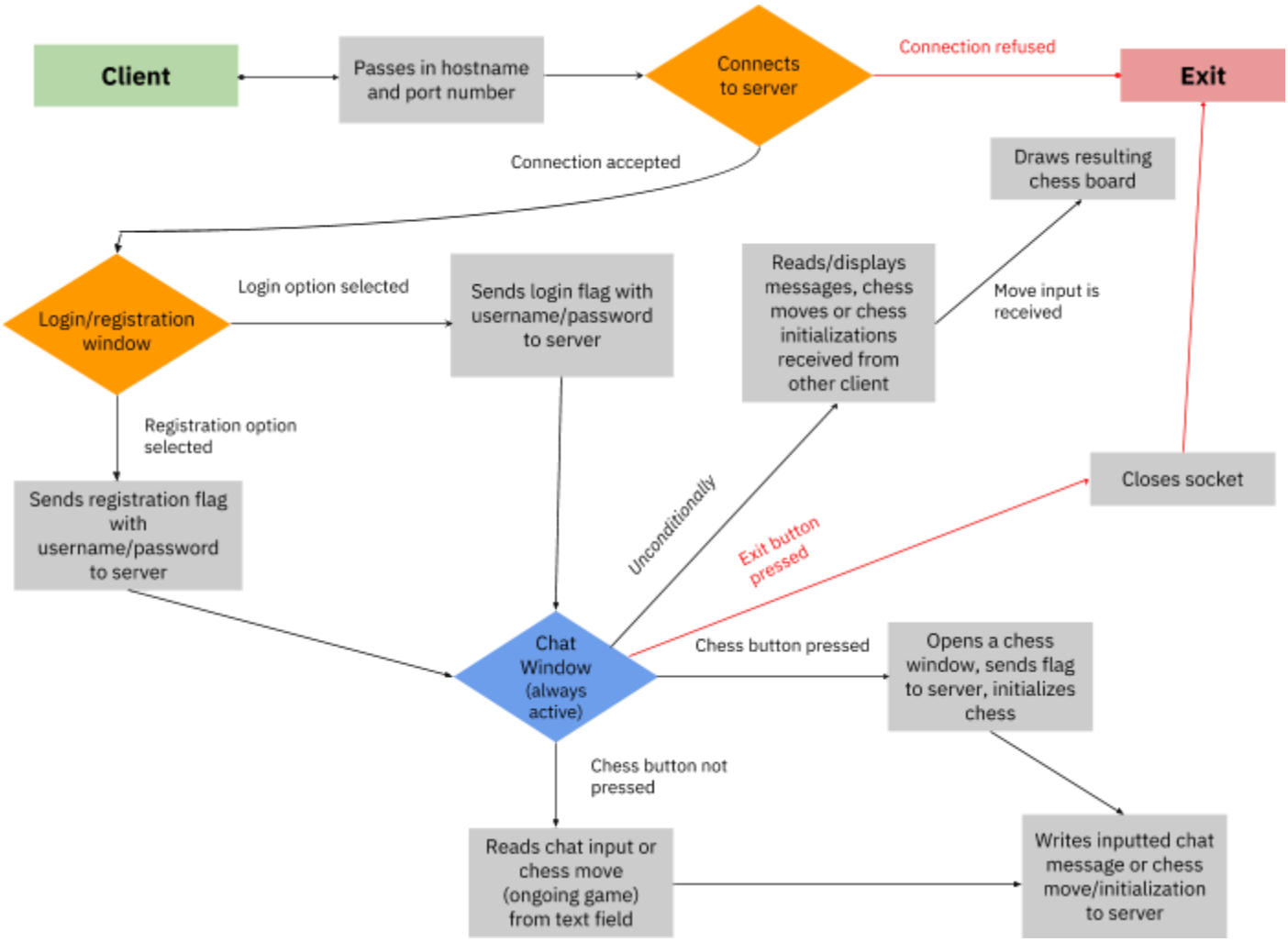
- **Client.c:** module that creates a client that allows communication with a server that will transmit their message to their specified destination client. Operates by using multithreading; once a socket is established, the relevant functions listen for both messages coming from other clients via the server and user inputs to send messages to other clients.
- **Board.c/.h:** Shared amongst Client.c and Server.c. Allows the user to play a chess game. *For more information about Board.c and Board.h, please see the Software Specification Manual for the Chesster chess game.*

- **Chat.c/.h:** Opens the login screen where the client will enter the username and password. The login screen has a register button which opens up the register screen. The register screen allows the user to create a username and password to be registered to the server.
- **Friend.c/.h:** Opens a window with the list of friends and allows the user to initialize the chat window. This window holds the buttons for adding and removing friends. This also gives the user the option to logout of their accounts.
- **ChatWindow.c/.h:** Opens a window where messages are exchanges. It holds a entry box, a button to send message and a table that displays the exchange.

Major network communication APIs:

- **FatalError:**
    - Input: const char *errormessage
    - Output: void
    - Description: Prints an error message
- **Receive:**
    - Input: void *sockfd
    - Output: void pointer
    - Description: Receives the message from the socket and prints it
- **Main:**
    - Input: command line arguments for port and hostname
    - Output: calls other functions (main program operations)
    - Description: runs the client-side program by receiving/sending messages

1.4. Overall program control flow

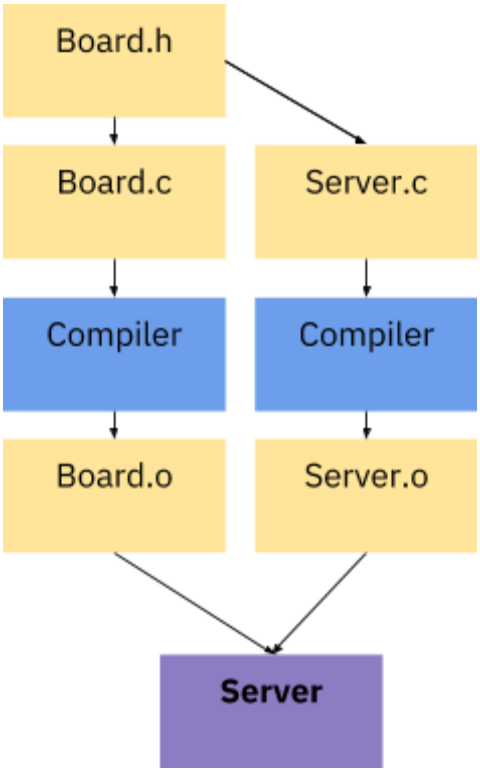


2. Server Software Architecture Overview

2.1. Main data types and structures

The **ChessInfo Structure** includes information about a client's current chess game with another client. Pointers are only dereferenced for this structure should a chess game be engaged by any client.

2.2. Major software components



2.3. Module interfaces

Modules:

- **Server.c:** module that creates a server that allows communication between two clients. Operates by using multithreading; once a socket is established, the relevant functions listen for both clients sending messages to the server and concurrently pushes those messages to their requisite destination clients.

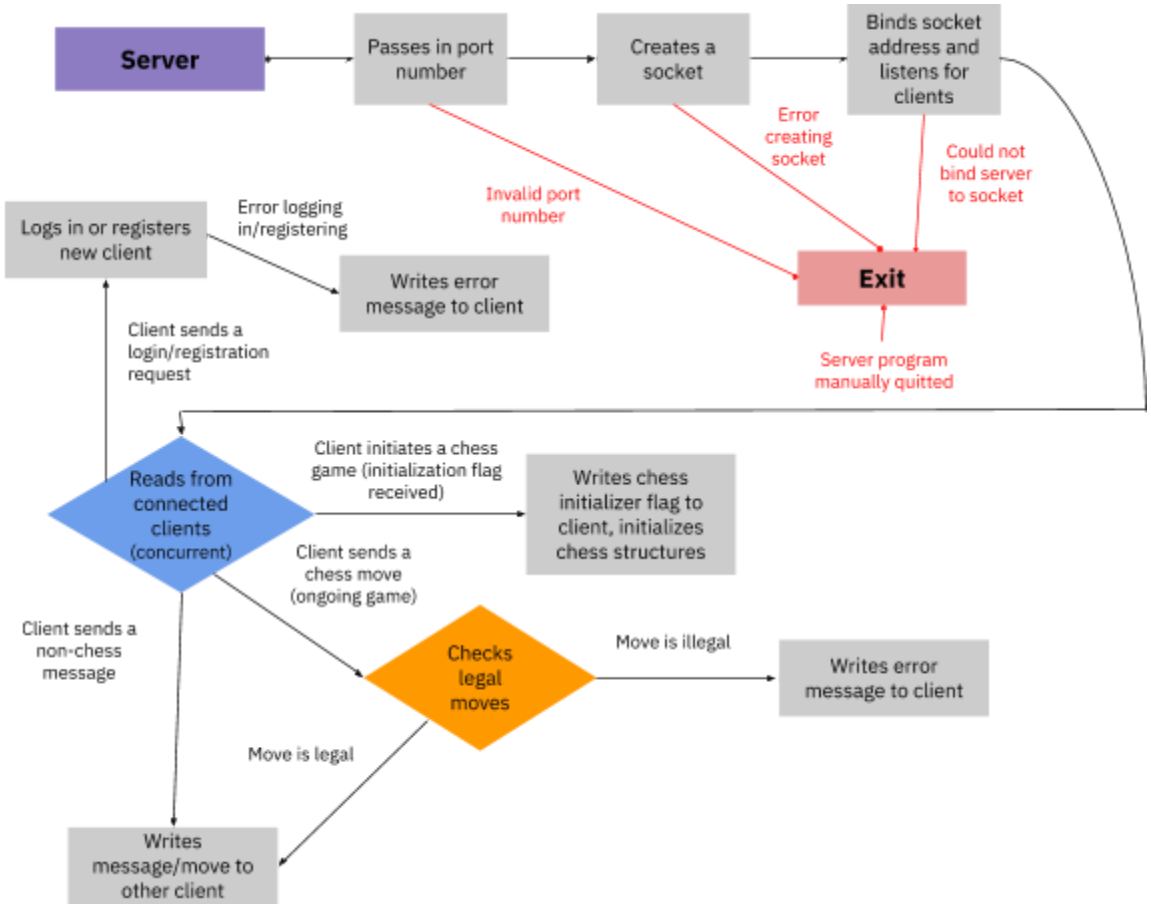
- **Board.c/.h:** Shared amongst Client.c and Server.c. Allows the user to play a chess game. *For more information about Board.c and Board.h, please see the Software Specification Manual for the Chesster chess game.*

Major network communication APIs:

- **FatalError:**
    - Input: const char *errormessage

    - Output: void
    - Description: Prints an error message
- **Send:**
    - Input: char *message, int sockfd
    - Output: void
    - Description: Sends a message to all users on the socket
- **Communicate:**
    - Input: void *sockfd
    - Output: void pointer
    - Description: Receives messages from each socket and if the message indicates that the user wants to play a chess game, it starts the chess game
- **MakeServerSocket:**
    - Input: uint16_t port
    - Output: int representing the socket
    - Description: Creates the server socket.
- **Main:**
    - Input: command line arguments for port only
    - Output: calls other functions (main program operations)
    - Description: runs the server-side program by receiving/sending messages and performing client management

2.4. Overall program control flow



3. Installation

3.1. System requirements and compatibility

- Linux environment with support for command line interface
- Processor with x86 architecture on server running in Linux environment
- Minimum free disk space: 1 GB
- Minimum memory: 512 MB

3.2. Setup and configuration

1. Open a window in your preferred Linux terminal program.
2. Using the command line, search through your filesystem to find the folder with the archive file. For example:
   `cd computer`
   `cd downloads`
3. Extract the software file from the archive file using the following command:
   `tar -xzvf P2_V1.0_src.tar.gz`

3.3. Building, compilation, installation

1. Enter the main directory:
   `cd ChatFinalDeveloper`
2. Compile the program:
   `make`
3. Run the server and/or client versions of the program (hostname must correspond to an IP address, and port must be greater than 2000):
   `cd bin`
   `./Server (port number)` and/or `./Client (hostname)(port number)`

4. Documentation of packages, modules, and interfaces

4.1. Detailed description of data structures

```
struct ChessInfo
{
    int socket;
    PLAYER *player;
};
```

The **ChessInfo** structure includes information about a client related to their current chess game on the Server.c module. It has an integer that represents the current client's socket, and a pointer to a PLAYER structure which has information about which player in the current chess game they are and what pieces they have *(for more information about the PLAYER structure, please see the Software Specification Manual*

for the Chesster chess game). This structure only has values assigned to it when one player initiates a chess game.

```
struct User
{
      char *Username;
      char *Password;
      char *Friends[100];
};
```

The **User** structure includes the information of each user registered into the program. It includes the user's username, password, and friends list (list of character pointers with maximum 100 friends), which is in the form of a character array.

4.2. Detailed description of functions and parameters

Server.c:

`void FatalError(const char *errormessage);`
Receives an error message string from whatever function is calling it, and displays that to the user and halts the execution of the program.

`int Authentication(char auth, char username, char password);`
Receives variables from the Communicate function if #REG# or #LOGIN# tag was used. Server checks the master user text file. If #REG# was used, the server searches for any existing user with the same name, and returns 0 if none are found. If #LOGIN# was used, the server searches for an identical username, and compares the registered password and the password given by the client. If both the usernames and the passwords match an existing user in the master text file, then the user is allowed access under that name.

`void Send(char *message, int sockfd);`
This function receives a message string and the socket file descriptor of the current user from the Communicate function. It then sends the message string to the other client that the current client is chatting with.

`void *Communicate(void *sockfd);`

This function passes in a socket and receives messages from that socket. If the message indicates that the user wants to play a chess game, it starts the chess game. If the message indicates that the user wants to either login or register, then it sends the line under variables “auth”, “username”, and “password” for each respectively to the Authentication function. Variable “auth” must be either “#REG#” or “#LOGIN#”. If this function returns 0, then the login/registration was a success. Any final messages that need to be sent to the other client which the current client (denoted by their socket file descriptor passed in) are sent by a function call to the Send function.

`int MakeServerSocket(unit16_t port);`
This function passes in a port number and creates the server socket. It first creates a server socket, binds it, and then listens on that socket.

`int *main(int argc, char *argv[]);`
The main server function takes in command line arguments and calls MakeServerSocket to create a socket. If the socket is successfully created, the function then concurrently calls Communicate (using multithreading) to receive messages from clients and handle them appropriately.

Client.c:

`void FatalError(const char *errormessage);`
Receives an error message string from whatever function is calling it, and displays that to the user and halts the execution of the program.

`void *receive(void *sockfd);`
This function accepts a socket file descriptor, which denotes the server. The function uses that file descriptor to read data that is coming from the server (and other clients via the server). It calls the DrawBoard function if a chess game is initiated by any client or a move is made on an existing chess game (and the current client receives a requisite chess flag from the server).

`FILE *CreateFriendList(char *username);`
Creates a friends list text file for a certain user by opening a new file named “<username>friendlist.txt”, appending the user’s username to the file to specify that it is theirs, and then returning that text file.

`FILE *AddFriend(char *requestusername,char *acceptusername, FILE *userfriendlist);`

The passed in parameter requestusername is the username of the user who is requesting another user to be their friend, acceptusername is the username of the user who has to accept the friend request, the file userfriendlist1 is the friend list of the requesting user, and the file userfriendlist2 is the friend list of the accepting user. This function opens the two files and appends each user's username. The requestusername will be appended to the acceptusername's friend list and the acceptusername will be appended to the requestusername's friend list.

`FILE *RemoveFriend(char *username,char *username2bedeleted, FILE *userfriendlist);`
The passed in parameter username is the username of the user who's friend list we want to remove a friend from, username2bedeleted is the username of the friend we want to delete, and the file userfriendlist is the user's friend list that we are editing. It opens the original friends list and reads each line until it finds the username that needs to be deleted, then copies the same file but without that line with the deleted username into a new file and returns the new file.

`int *main(int argc, char *argv[]);`
The main client function takes in command line arguments and calls MakeServerSocket to create a socket. If the socket is successfully created, the function then concurrently calls receive (using multithreading) to receive messages from the server or other clients (via the server) and handle them appropriately. This function also contains a while loop that writes messages that are inputted in the text field to the server.

Chat.c:

`void ShowPassword( GtkWidget *checkbutton, GtkWidget *entry );`
This function is connected to a checkbutton widget in CreateRegisterMenu and CreateLoginMenu. When the checkbutton widget is pressed, the password entry is passed through a signal connect and calls ShowPassword. Once ShowPassword is called, the text from the password entry becomes readable text.

`void screen_Register(void);`
This function calls CreateRegisterMenu which opens a new window. It checks to make sure that only one instance of the register window is open at any time.

`void screen_Login(void);`

This function gets the text from the entries from the Login window and checks if they are not empty. If the entries are not empty, then the Login window will be deleted and ChatMainMenu will be called which is from Friend.c.

`int ComparePasswords(const gchar *PW, const gchar *CPW);`
This function called in Register_function and checks two strings from entries to make sure that they are equal to each other. If they are equal, the function returns 0, else if they are not, the function returns 1.

`void CreateRegisterMenu(void);`
This function creates the register window which includes three text entries, three labels, two check buttons and one register button. Each text entry corresponds to a “Username”, “Password”, and “Confirm Password” input. The two checkbuttons call ShowPassword. The Register button calls Register_function when clicked.

`void CreateLoginMenu(void);`
This function creates the login window which includes two text entries, two labels, one check button and three buttons. Each text entry corresponds to a “Username” and “Password” input. The check button calls ShowPassword. The Login button calls screen_Login when clicked. The Register button calls screen_Register when clicked. The Exit button calls gtk_main_quit which will exit the program.

`int *main(int argc, char *argv[]);`
The main Chat function which initializes GTK and calls CreateLoginMenu.

Friend.c:

`void init_list(GtkWidget *list);`
This function stores and creates a list where the friend list is stored. It passes a list widget and initializes the widget list that contains the friend list.

`void add_to_list(GtkWidget *list, const gchar *str);`
This function adds new user names to the friend list that shows in the list tree widget in the friend list window. It passes in the list widget and a name to add new friends to the list.

`void on_changed(GtkWidget *widget, gpointer label);`
This function is used to change the selection of friends in the friend list tree. It takes in a widget and a label to switch the selection in the list of friends. This makes it possible

to see which friend is on the see which friend is selected in the list tree and change between the selection of friends.

`void ChatMainMenu(int argc, char *argv[]);`
The main Friend function creates the window that holds the list of friends, a button to initialize the chat window, a request button that opens a new window that shows all the friend requests for the user, a button to add new friends to the list and a remove button to remove a friend. This window also has a entry box that gets the user input to search through the username logs to find and add new friends.

ChatWindow.c:

`int main(int argc, char *argv[]);`
The main ChatWindow function which creates the chat window.The chat window has two sections: a scrollable window for displaying the chat log and, below it, a text entry box with a send button. When the user clicks the send button on the GUI, main() calls SendMessage, which will take the data from the text entry box on GUI and send it to the server. If there is no message, no message will be sent, and the program does nothing.

Static void SendMessage(GtkWidget *widget, gpointer entry);
When called, the function takes in a pointer to the text entry box. The function then calls TextEntryGet() to get the contents of the entry box. If the contents are empty, the function does nothing. If the contents are not empty, the string is printed in the scrollable window created from main() and sent to the server.

4.3. Detailed description of communication protocol

Login/Registration Procedure:

The GUI version of Client.c first greets the user with two text fields: one for the username, and one for the password. The user then enters enter a username and password in the corresponding text fields, and clicks one of the two buttons at the bottom of the screen to either register as a new user or login to an existing account. There is no separate screen for logging in and registering; rather, the client just clicks the desired button for the server to handle their account data how they wish.

One of the following flags is sent to the server to login or register (delimited with the new line character):

Client to Server:
`#REG#\n <username>\n <password>\n` (for registering a user, omitting <>)
`#LOGIN#\n <username>\n <password>\n` (for logging in a user, omitting <>)

The program will then send this data to the server, and the server will determine if it is valid. For registering, it will check against a central text file located alongside the server executable to ensure that the desired username the client has selected has not already been taken by another user. If it has, the server will send this error flag back to the client, which will display a user-comprehensible error message of its own:

Server to Client:
`#REGERROR#`

If the registration was a success, the server will send this flag back to the client to permit it to continue with the program:

Server to Client:
`#REGSUCCESS#`

For logging in, the server will check against the same text file and make sure that the username exists and the password matches that username. If either of these conditions are not met, the server will then send a generic error message to the client, which will in turn display a user-comprehensible error message of its own:

Server to Client:
`#LOGINERROR#`

If the login was a success, the server will send this flag back to the client to permit it to continue with the program:

Server to Client:
`#LOGINSUCCESS#`

Adding/Removing Friends Procedure:

Once a client is logged in or registered, the friends list window appears. It will present the client with a list of their friends. To add a friend, the client should be on on the friends list window, then they type in the username of the person they would like to add

and click the add button. To remove a friend, the client selects the username of the person (that exists on the friends list window) they would like to remove and then click the remove button.

One of the following sets of two flags are sent to the server for adding or removing friends:

Client to Server:
`#ADDFRIEND#`
`<username>` (for adding a friend, omitting <>)

`#RMFRIEND#`
`<username>` (for removing a friend, omitting <>)

Chat Procedure:

If a client wishes to chat with another active client, the client should select the user they would like to chat with from their friend list window and then click the adjacent chat button. The program will then present a chat window on which they are able to send messages with the selected user. The following flag set is sent to the server that specifies the client username they wish to chat with:

Client to Server:
`#CHAT#`
`<username>` (omitting <>)

Chess Procedure:

If a client wishes to initiate a chess game with another client, they push the chess button in the chat window, and a board will appear on another window. To make a move on on the chess board, the client will type in a move in the chat window text field and then a board with the updated move will appear.

The program sends the following flag to initiate a game of chess:

Client to Server:
`Chess`

If it is determined that the text input of a client is in the form of a chess move, the server will handle that string input and make the requisite move on the server-side data structures, and then send these flags/strings back to each of the clients (including the client that made the move) to make that move on the client-side board:

Server to Client:
`#CHESS#`
`<move>` (omitting <>)

Exiting Procedure:

If any client wishes to quit the program, the client will press on the 'x' button on the top right of the friend list window or click on the log out button.

The client will send the following flags to the server to update its active status:

Client to Server:
`#OFFLINE#`
`<username>` (omitting <>)

5. Development plan and timeline

5.1. Partitioning of tasks

Week 6: Research
- GUI Team:
    - Rough draft of GUI for the chat
        - Implementing buttons and text boxes
- Client/Server Team:
    - Research on Sockets and connections

Week 7: Continue Research and Beginning of Integration
- GUI Team:
    - Continuation of creating program windows
- Client/Server Team:
    - Implementation of Multiple Sockets and Friends List

Week 8: Alpha Release Preparation
- Teams Merge
    - Get the GUI to behave according to the code

Week 9: Debugging for Final Release

- All Hands: Adjust and debug code accordingly.

Week 10: Final Release

- All Hands:
    - Update/Finalize User Manual and Software Specification
    - Final debugging process

5.2. Team member responsibilities

GUI Team: Benny, Marvis, Raiyan

- Create a working and visually pleasing graphical user interface that will serve as the main point of interaction for clients
- Link GUI elements with specific functions
- Link GUI with client/server modules
- Possibly create a GUI for the server

Client/Server Team: Armand, Daniel, Jada, Kevin

- Create robust, reliable client-server communication to send and receive messages between at least two clients connected to a central server
- Enable support for usernames, passwords, and friends lists.
- Enable chess functionality so that two clients can play chess against each other, hosted by the server
- Work with GUI team to help integrate GUI elements into program functions and modules

Copyright

References

GNOME Human Interface Guidelines, www.developer.gnome.org/gtk2/stable/ch02.html.

Bodnar, Jan. “ZetCode.” *JavaScript Snake Tutorial - Creating Snake Game in JavaScript*, www.zetcode.com.

Index

Variable	3