ZotChat

Software Architecture Specification Version 1.0.0

University of California, Irvine - EECS 22L SOFTWARE ENGINEERING PROJECT IN C LANGUAGE - Winter 2019

TEAM 5 - 4OUR

HE, JINGTIAN
RAMOS, ROY
TANG, JOHNATHAN
TRINH, BRYAN
ZHU, KEVIN
ZOU, ZIHAO

February 20, 2019

TABLE OF CONTENTS

GLOSSARY

2D Array:

Also called a **2-dimensional array**. Similar to a 1D array, a 2D array is a collection of data, however, it is organized as a matrix with rows and columns, which is why it has two dimensions.

Blocking:

A type of network that has fewer transmission paths than is required if all users were to communicate simultaneously (b) is used to reduce equipment requirements, (c) is used because not all users require service simultaneously, and (d) makes use of certain statistical distributions that apply to the patterns of user communications demand.

Client:

A laptop, desktop computer or workstation that is capable of obtaining information and applications from a server via the internet, intranet, extranet etc.

Define:

When something is defined, it means that it is created and exists in the program file and a part of memory is allocated for it. It is similar to declarations and initializations, but differs from the former by having memory allocated and the latter by not having an initial value.

Enum:

Also called an **Enumeration**. Enum is a user-defined data type in C, mainly used to assign names to integral constants - the names make a program easy to read and maintain.

Graphic User Interface:

Also called **GUI**. It is a form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators, instead of pure text-based user interfaces.

Hard-Coding:

Hard-Coding is the practice of fixing the data/parameters in a program such that it would run only that certain way, as opposed to obtaining data from other external sources at run-time.

Include:

A request to use a particular header file, written as “#include” in C. This ties in with header files as the file contains C declarations and macro definitions to be shared between multiple source files.

Instant Messaging:

A type of online chat that offers real-time text transmissions over the Internet for two or more than two people.

Internet Protocol Address:

Also known as an *IP address*, is a numerical label assigned to each device connected to a computer network that uses the Internet Protocol for communication. Serves two functions: host or network interface identification and location addressing.

Initialization:

Initialization is the assignment of an initial value for a data object or variable. The manner in which initialization is performed depends on the programming language, as well as type, storage class, etc., of an object to be initialized.

Macros:

A macro in computer science is a rule or pattern that specifies how a certain input sequence should be mapped to a replacement output sequence according to a defined procedure. This fragment of code is given a name, and whenever the name is used, it is replaced by all of the contents of the macro. The 2 types of macros are object-like macros, which resemble data objects, and function-like macros, which resembles function calls.

Output:

Output is a collection of data that is generated by a computer, including data produced at a software level, such as a generated calculation, or a physical level, such as a printed document.

Pointer:

A pointer is a programming language object that stores the memory address of another value located in computer memory. A pointer references a location in memory, and obtaining the value stored at that location is known as dereferencing the pointer.

Register:

One of a small set of data holding places that are part of the computer processor. Registers may hold an instruction, a storage address, or any kind of data.

Repository:

A central location in which data is stored and managed.

Send:

Cause (a message or computer file) to be transmitted electronically to another user

Server:

A computer or computer program which manages access to a centralized resource or service in a network.

String:

A string is a sequence of characters as a lateral constant or in a form of a variable (that is able to be changed after creation)

User-Input:

Also simplified to just **Input**. User-Input is the collection of information or data sent to the computer for processing through the use of an input device. Its counterpart is **Output**.

1.0 Client Software Architecture Overview

1.1 Main Data Types and Structures

Integer:

Integers are an important aspect of creating a server. Using the socket() function in Linux to create a socket, it returns an integer which is basically a file descriptor. These file descriptors help identify socket resources. These socket descriptors, a type of file descriptor, are what allow users to connect with each other when using other Linux functions such as send() and recv().

Structures:

The structure sockaddr_in is an important structure that was included using a preprocessor macro. This address specifies the type of socket created and is important when binding the server into a specific port so that other clients can connect to it. After multiple clients binding to the master server socket, the use of the address of the master server socket and the socket descriptor of the client can help track the IP and who disconnected from the server. This is an invaluable tool for debugging and helping create a server.

Strings:

The server and the client communicate with each other through strings. The server would constantly be listening to the clients. Using the socket descriptor, it would read the incoming message as a string. The string would then need to be parsed and the final part of the message would need a null character to be added on to it.

1.2 Major Software Components

Libraries:

- Takes in the 'Chess' Library that contains a working chess game
- Uses the socket.h library which is Linux's set of files that contain information regarding socket building, binding, and sending and receiving messages
- Uses the netdb.h library to manage network database operation

Client Login:

- The server stores information regarding the client's login information, friends list, and their conversation logs
- The client can make a new account if they do not already have an account

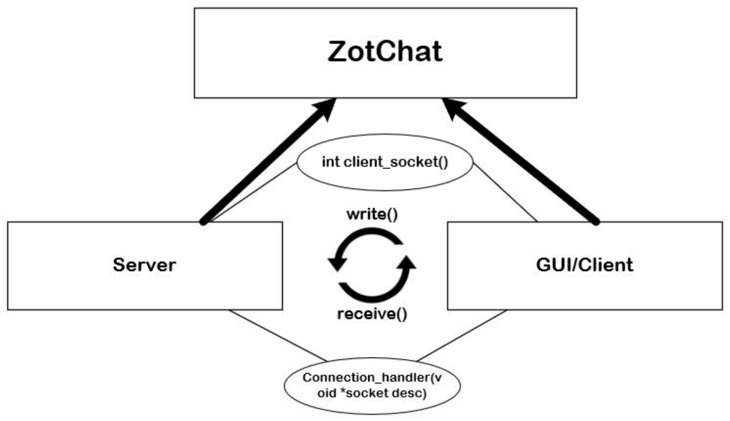
Communication between Clients:

- Multiple clients can connect on the server and talk to each other
- Clients have to wait for each other to respond in order to proceed to the next message
- Clients are able to add each other on their friend list and start a game of chess

GUI Display:

- Accepts user input and communicates the input to the rest of the program
- GUI will also handle the chess game

The program itself contains all of the above and follows game logic refer to **Section 1.4** detailing the game logic and program flow. It will communicate with the GUI receiving and sending values to help update the display. Clients, after connecting to the server, will be able to play games of chess, talk to one another, and log into the server.



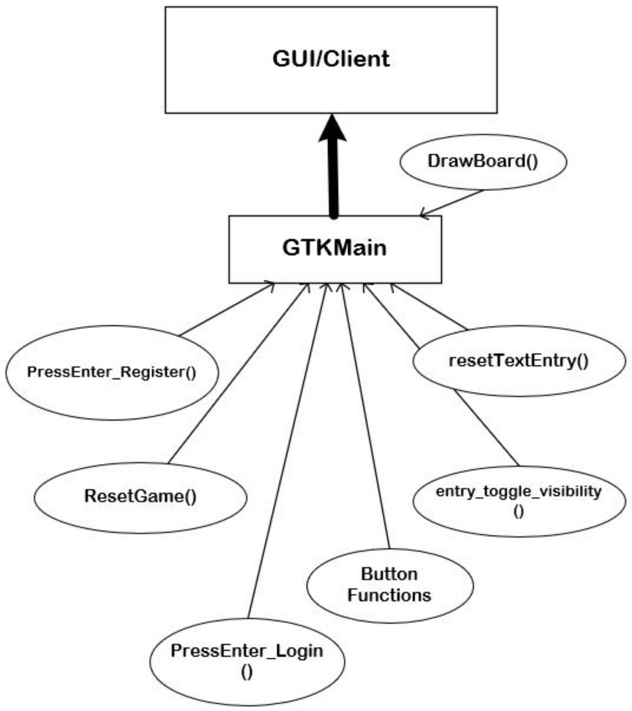


Figure 1:

Module Hierarchy of Client

1.3 Module Interfaces

API of Major Module Functions

void gtk_widget_hide **(GtkWidget** *widget**)**

- Input: GtkWidget
- Output: None
- Description: takes a shown widget and hides it, along with all of its children. Mostly used for closing up windows after opening up new ones, but is also used to hide certain widgets on the window.

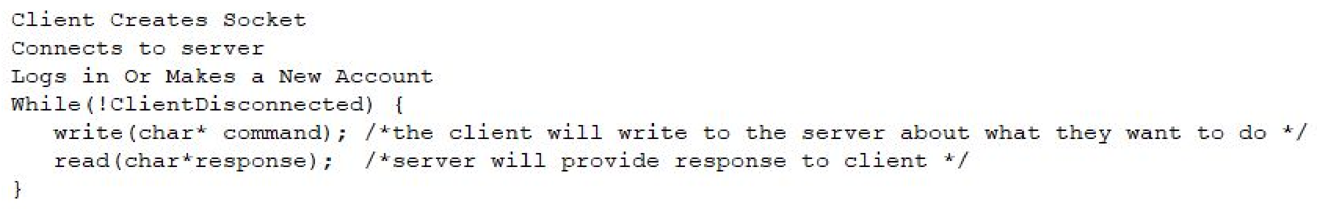
void gtk_widget_show_all **(GtkWidget** *widget**)**

- Input: GtkWidget
- Output: None
- Description: takes a hidden widget and displays it, along with all of its children. Mostly used for opening up new windows, but can also be used to display a widget within a widget

Int* ParseString**(char*** signal**)**

- Input: a string with the communicated message from the client or the server
- Output: array
- Description: Parses the command from the client or server. This function translates the message to be done in terms of an array

1.4 Overall Program Control Flow



```
Client Creates Socket
Connects to server
Logs in Or Makes a New Account
While(!ClientDisconnected) {
   write(char* command); /*the client will write to the server about what they want to do */
   read(char*response);  /*server will provide response to client */
}
```

Figure 2:

Pseudocode of overall program control flow

The server must be started before any users can connect to it. After creating a socket, it must bind to a port, then it will listen for any signals. If there is a new connection, it will join the new user to the server so that they can communicate with each other.

The ZotChat program welcomes its users with a title screen that asks whether or not the user would like to register an account or log in. The registration window will take in inputs of username, password, and the re-entering of the password for confirmation to create an account. If the passwords are not the same or any of the 3 entries are not filled in, ZotChat will not allow the user to create an account. After creating an account, or skipping this step if the account is already created, the user will find themselves at the title screen yet again, offering a registration button or a login button. Once the login button is pressed, a new window will pop up, allowing users to enter their username and password. If the password and username match a set in the database, the user will be allowed to enter their home menu.

The home menu consists of a list of the user's contacts, a button to add new users, and a button to check for incoming friend requests. If the button to add new users is clicked, a pop-up will appear to allow the user to enter the username of another client. If the username does not exist, the user will be notified. If the request is successful, the user will also be notified. If the button to show incoming friend requests is clicked, a list of users who have sent a friend request will show up. If a user on the list is clicked, a pop up will appear which allows for the acceptance or declination of the friend request, and the sender will be notified. The list of usernames in the home menu can be clicked, and if a friend is clicked, a new window will open to show the message history with that friend, as well as the opportunity to chat and play chess with them.

The message screen will display all the messages that have been sent between the two users, and will allow the user to send additional messages to the destination user. There are also 2 buttons, one on the chat window, one on the local home window to start a new chess game. If there is no game in progress, an error message will be sent to the user.

Additionally, ZotChat has a server that pulls data from all users. Clients will be pushing and pulling whatever data is given to them on the server.

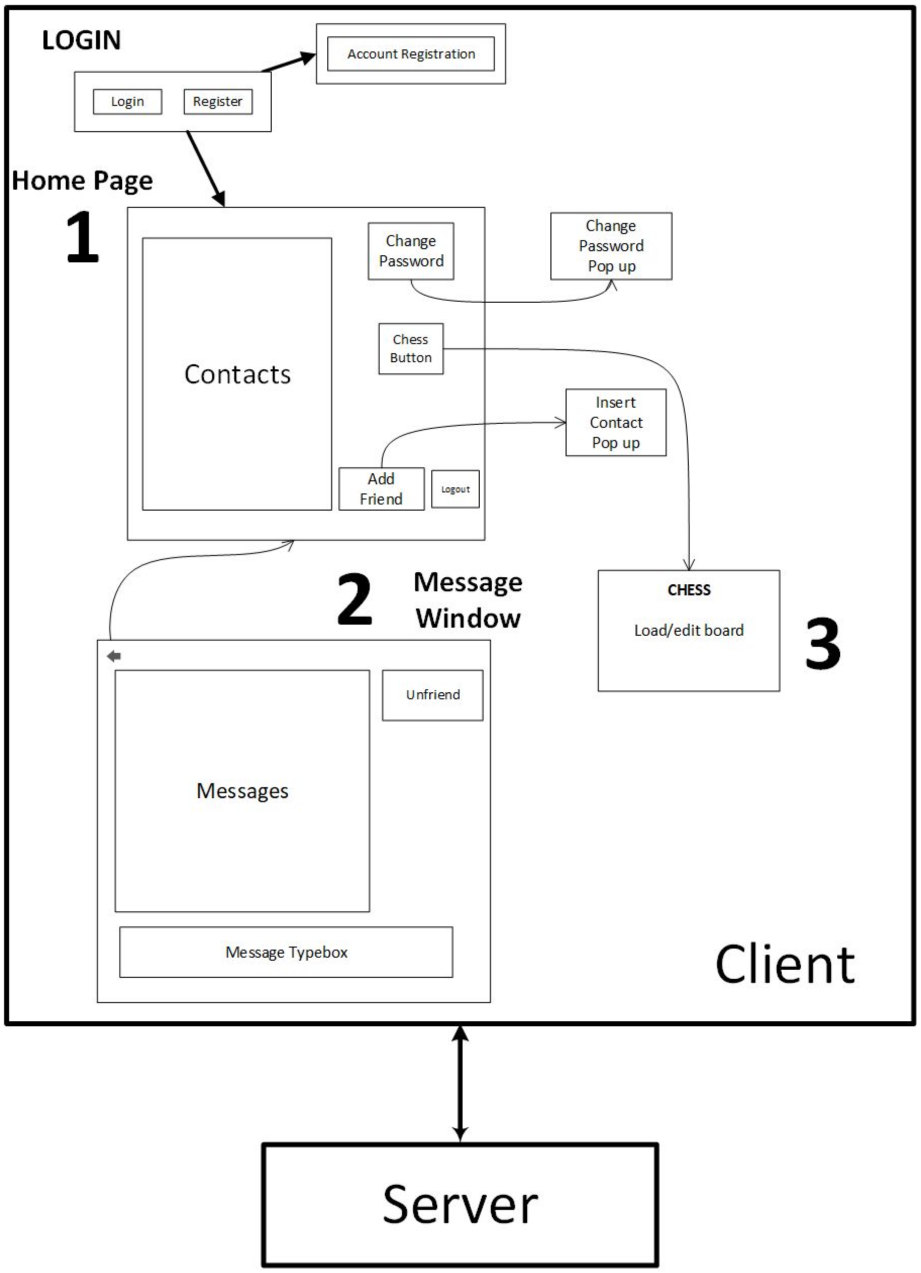


Figure 3:

Flow Chart of Client to Server Interaction

2.0 Server Software Architecture Overview

2.1 Main Data Types and Structures

Integers:

The server will use integers, but more specifically an integer array to store an accepted socket connection to allow communication between clients. As long as the array is not yet full and the recipient a valid target of the message, the message will be sent to the server to be pulled to the destination in a later cycle of read/write.

Structures:

The server will use pthread_rwlock_t as a structure for read/write lock. This ensures that readers and writers of the same priority won't starve each other, as in both will not attempt to no new readers will get a section if a writer is waiting for the section or vice versa.

2.2 Major Software Components

Libraries:

- Uses the pthread.h library that contains a threading module to manage and handle all the possible connections
- Uses the sys/socket.h library which is Linux's set of files that contain information regarding socket building, binding, and sending and receiving messages for servers
- Uses the netinet/in.h library for basic internet protocols

Client Login:

- Server stores information regarding the client's login information, friends list, and their conversation logs
- The client can make a new account if they do not already have an account

Communication between Clients

- Multiple clients can connect on the server and talk to each other

● Clients are able to add each other on their friend list and start a game of chess

The program itself contains all of the above and follows game logic refer to **Section 2.4** detailing the program flow. It will communicate with the client receiving and sending values to help update the display. Clients, after connecting to the server, will be able to play games of chess, talk to one another, and log into the server. Refer to **Figure 2** for additional information on client-server communication.

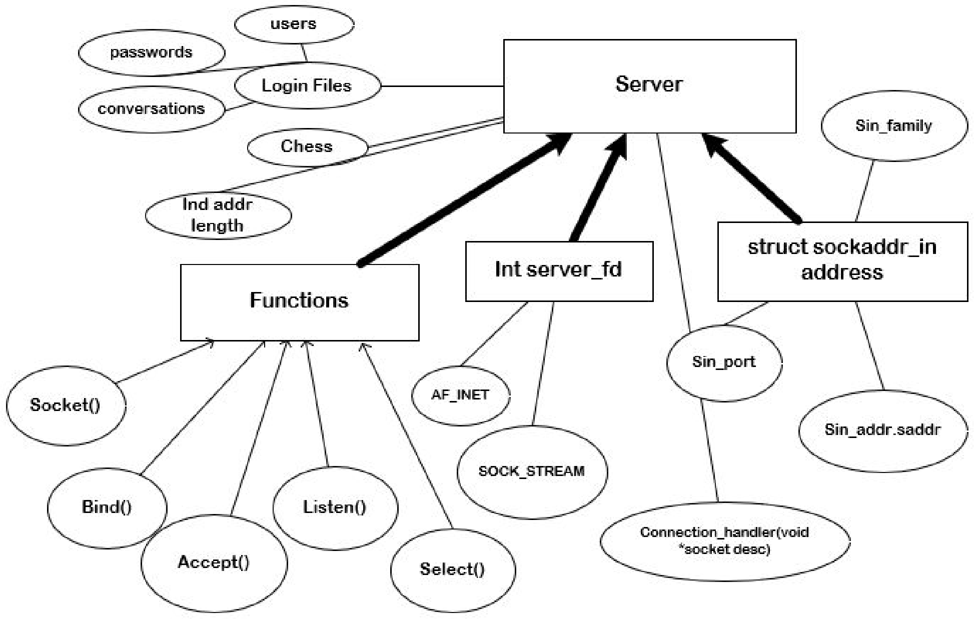


Figure 4:

Module Hierarchy of Server

2.3 Module Interfaces

API of Major Module Functions

Int Login **(char *** username**, char ***password**, int** new**)**

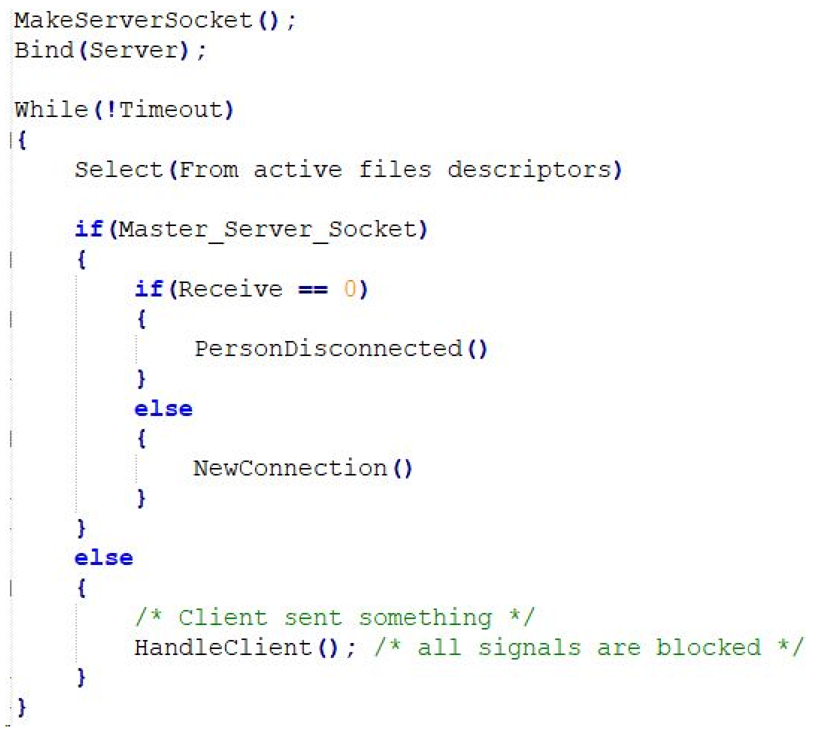
- Input: string with the username and password, or if the user wants to create a new login ID
- Output: int

- Description: creates or verifies a login ID. The user must do this before being able to fully connect to the server.

Int* ParseString**(char*** signal**)**

- Input: a string with the communicated message from the client or the server
- Output: array
- Description: Parses the command from the client or server. This function translates the message to be done in terms of an array

2.4 Overall Program Control Flow



```
MakeServerSocket();
Bind(Server);

While(!Timeout)
{
    Select(From active files descriptors)

    if(Master_Server_Socket)
    {
        if(Receive == 0)
        {
            PersonDisconnected()
        }
        else
        {
            NewConnection()
        }
    }
    else
    {
        /* Client sent something */
        HandleClient(); /* all signals are blocked */
    }
}
```

Figure 5:

Pseudocode of overall program control flow

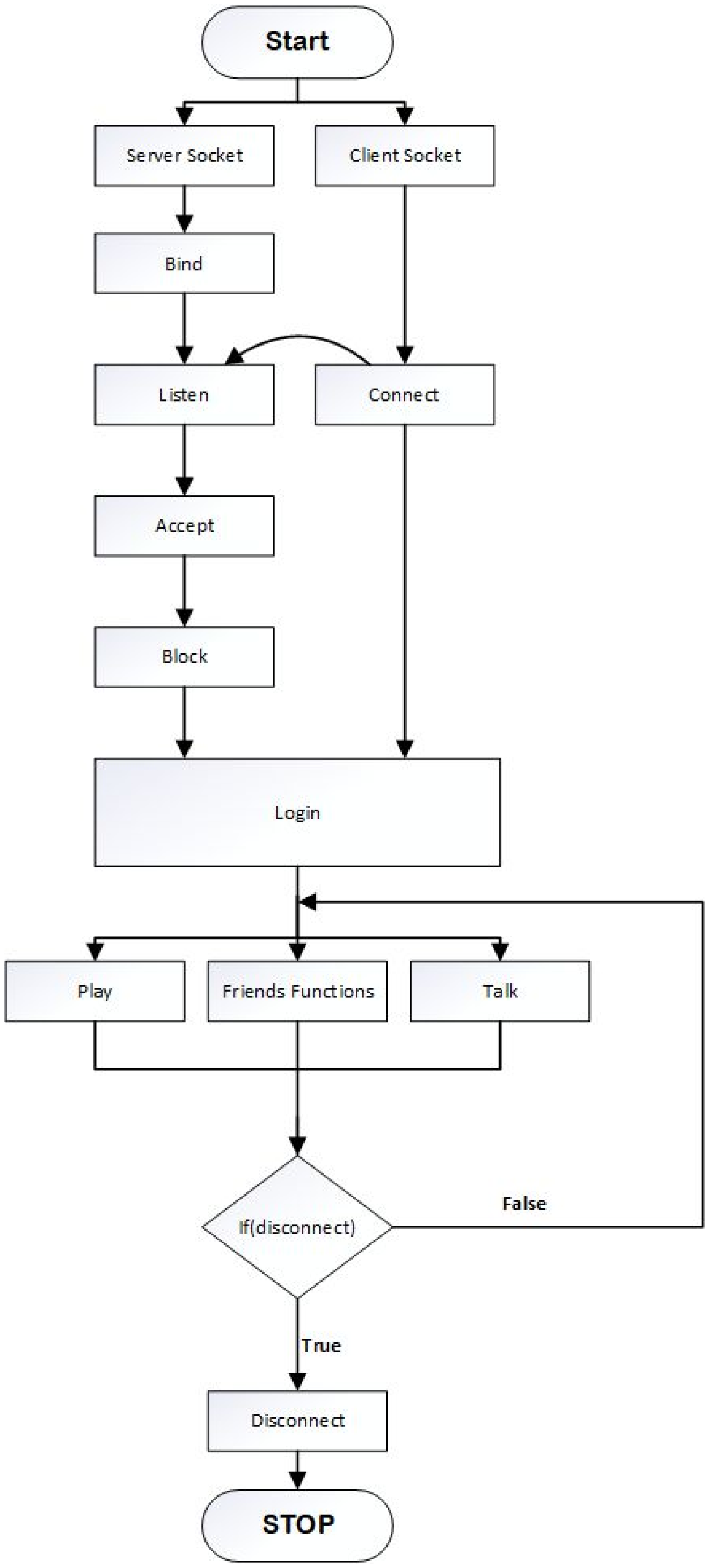


Figure 6:

Flow Chart of overall program control flow

The server must be started before any users can connect to it. A socket must be bound to a port after it is created. It will listen for any signals. If there is a new connection, it will join the new user to the server so that they can communicate with each other.

To log in from client to server, the server must receive a connection request from the client and the server must send back a validation of the request back to the client to initiate the connection. The Login() function allows the server to scan its database of account information in order to verify the correct account information to log in. If the information is correct, then the client will be able to log on and the server has to send the client the logged in command. If the client inputted incorrect account information, the server will send a message to the client saying that the account information is incorrect and to try again.

The server will have cycles of reading and writing to ensure that no users will starve each other. Starving means that one side will infinitely wait for their turn to come. For example, if a sender has equal or lower priority than the receiver, the receiver will always be pulling information and have data be streamlined towards it, even though the sender does not have the chance to send with its equal or lower priority. If a sender can not get their message to the server for the receiver to pull, since the receiver in this theoretical case is always pulling, then the sender will be starved and not send any information due to an equal or lower priority. To resolve this issue, there are read and write cycles that the server goes through in order to verify that no one is being starved. If the server is on read lock, then no one can write to the server, and if the server is on write lock, then no one can read from the server. This allows that no request can be neglected by being put on infinite hold.

The server will send out commands to the clients in forms of updating messages or requests and has to use ParseString() in order to tokenize the information for the client to read. After being made into a string, the client will then have to “decode” the information on the string and apply necessary updates on the client’s end.

3.0 Installation

3.1 System Requirements, Compatibility

ZotChat requires little processing power, yet succeeds to present a classic game that is meant to provide a fun-filled experience for its users. It does not only take 4 GB of Random Access Memory to run ZotChat but it also only requires 4 GB of Hard Drive space. Either an Intel Core i3- 3210 3.2 GHz or an AMD A8-7699 APU 3.1 GHz is required alongside an Nvidia GeForce 700 Series or an AMD Radeon Rx 200 Series Graphics Processing Unit(GPU) in order for ZotChat to be launched. The machine must run at least a 32-bit GNU/Linux Operating System. However, a 64-bit version of the Linux Operating System is recommended. Nevertheless, ZotChat is built and designed to run flawlessly with such provided system specifications.

In order to achieve mind-boggling performance, it is recommended that the machine must run using an Intel Core i7-8700 3.2 GHz Central Processing Unit. Another recommendation is for the machine to have 16GB of DDR4-2666 RAM with the company of an Nvidia GeForce RTX 2080 GPU. Lastly, ZotChat' optimal performance is deemed best when saved in a Solid State Drive(SSD). In this case, A 480GB M.2 SSD would allow for the fastest bootup times.

3.2 Setup and Configuration

The installation process of ZotChat was designed to be as feasible as possible for its users. The first step is to download the tar.gz file. Once the downloading process has finished, untar the tar.gz file using an untar application. After following the provided steps, the user is able to view the contents of the tar.gz file and is ready for the compilation process.

3.3 Building, Compilation, Installation

ZotChat' tar.gz file comes with everything that the user needs for the setup process. The developers of ZotChat has made it possible for the user to spend as little time as possible installing the program. The Makefile in the untarred folder allows for the compiling process to commence. By simply typing “make all” and pressing enter, the compiler through of the Linux system is prompted to compile the modules necessary. Once the compilation is successful, the installation process is done. Running the executable file would then start up ZotChat.

If the user chooses to uninstall ZotChat, the user must first close the application window. If necessary, the folder where the contents of the tar.gz file were unpacked is to be deleted. Otherwise, deleting the tar.gz file would finalize the uninstallation of ZotChat.

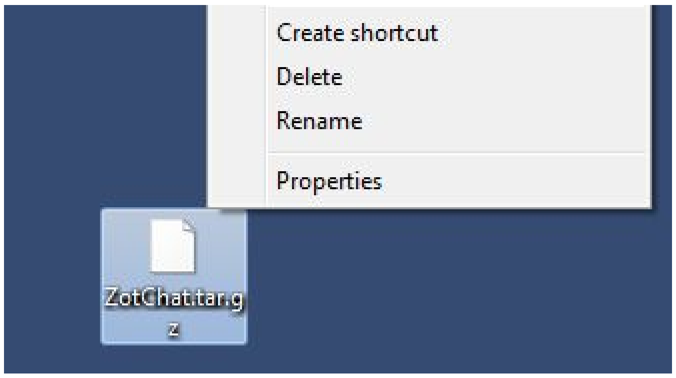


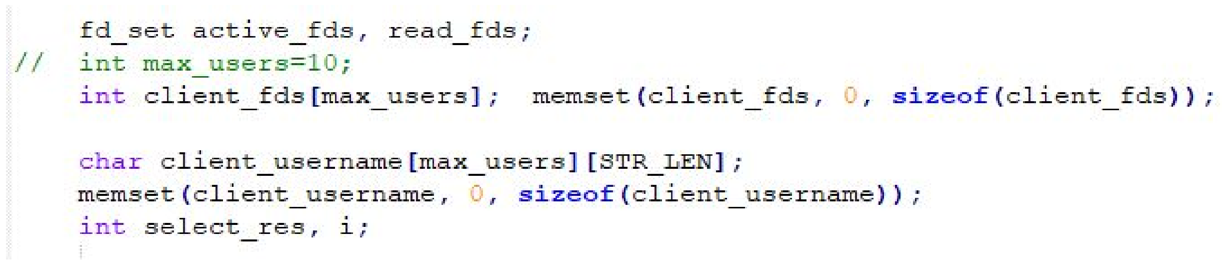
Figure 7:

The snippet of .tar.gz file for ZotChat

4.0 Documentation of Packages, Modules, Interfaces

4.1 Detailed Description of Data Structures

The user system in the server is built with arrays.

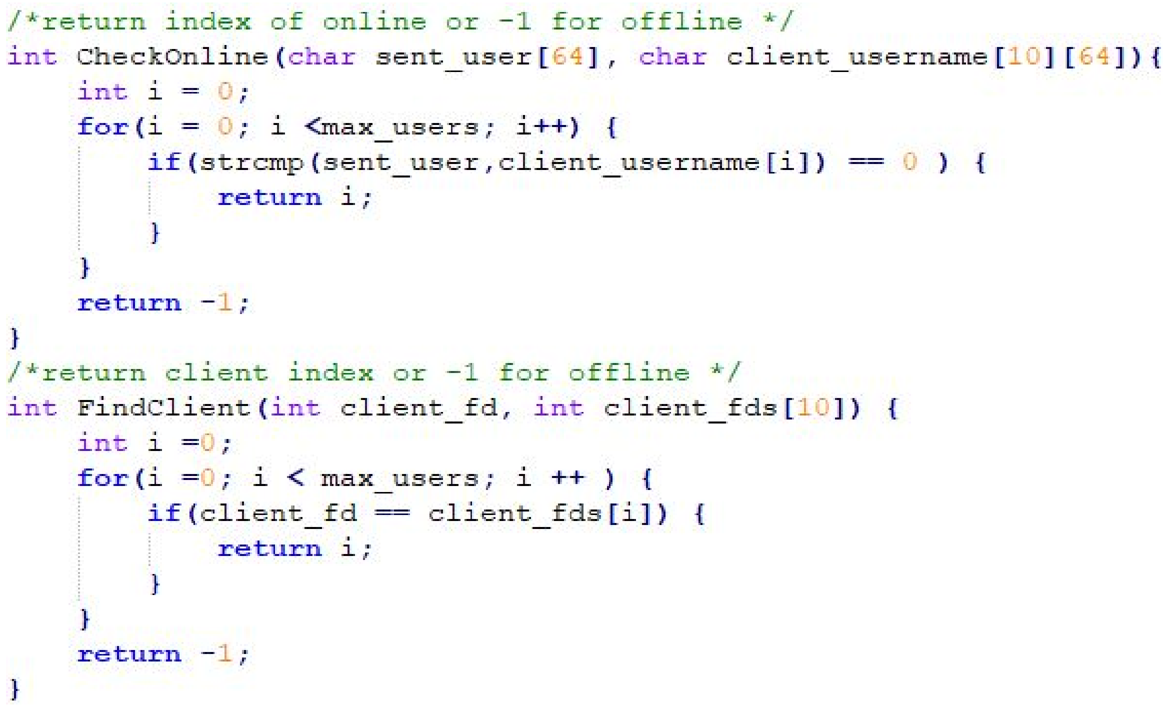


```
    fd_set active_fds, read_fds;
//  int max_users=10;
    int client_fds[max_users];  memset(client_fds, 0, sizeof(client_fds));

    char client_username[max_users][STR_LEN];
    memset(client_username, 0, sizeof(client_username));
    int select_res, i;
```

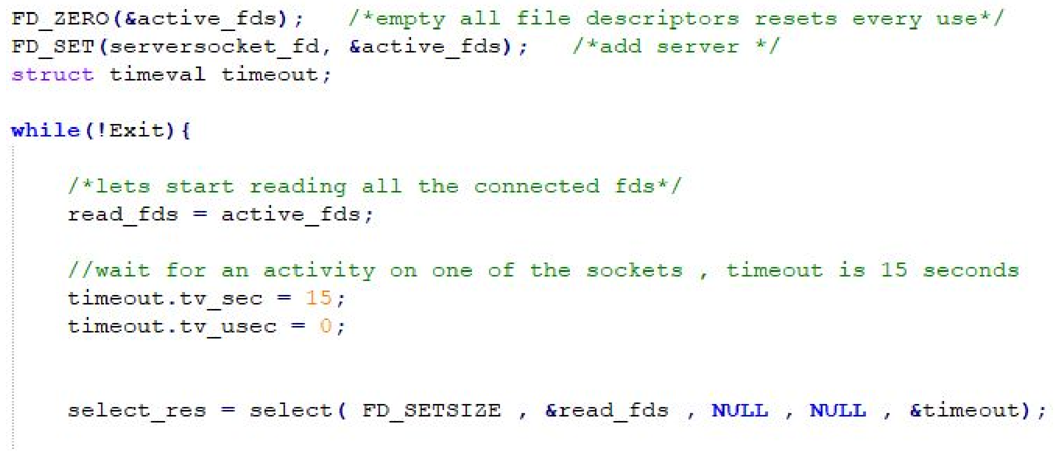
Figure 8:

The snippet of structure for Server Information

These arrays are stored in the server. The max_users, a macro, is defined to be 10 users. This will also not allow other users to join if there are already 10 users on the server and kick them out. When logging in, their username and file descriptor are synced inside the array ie. they will have the same index. Functions were created to help find the username given.



```
/*return index of online or -1 for offline */
int CheckOnline(char sent_user[64], char client_username[10][64]){
    int i = 0;
    for(i = 0; i <max_users; i++) {
        if(strcmp(sent_user,client_username[i]) == 0 ) {
            return i;
        }
    }
    return -1;
}
/*return client index or -1 for offline */
int FindClient(int client_fd, int client_fds[10]) {
    int i =0;
    for(i =0; i < max_users; i ++ ) {
        if(client_fd == client_fds[i]) {
            return i;
        }
    }
    return -1;
}
```

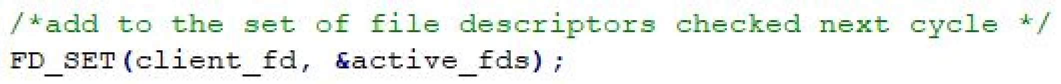


```
FD_ZERO(&active_fds);   /*empty all file descriptors resets every use*/
FD_SET(serversocket_fd, &active_fds);   /*add server */
struct timeval timeout;

while(!Exit){

    /*lets start reading all the connected fds*/
    read_fds = active_fds;

    //wait for an activity on one of the sockets , timeout is 15 seconds
    timeout.tv_sec = 15;
    timeout.tv_usec = 0;


    select_res = select( FD_SETSIZE , &read_fds , NULL , NULL , &timeout);
```

Figure 9:

The snippet of the main server loop

The select function blocks and when it detects activity, it will select one of the active file descriptors, including itself. The server will always continue running, unless it times out, which the default is 15 seconds (25 seconds for the GUI).

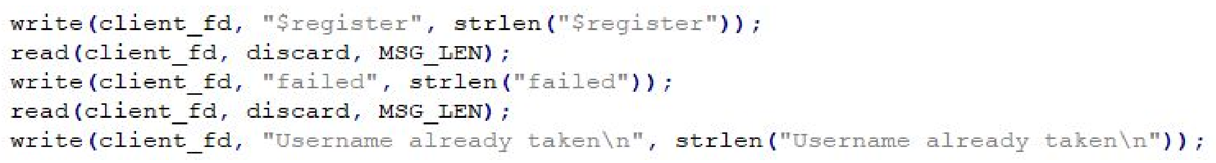


```
/*add to the set of file descriptors checked next cycle */
FD_SET(client_fd, &active_fds);
```

Figure 10:

The snippet of function to how the server adds new clients

The client will be added to the read fds, active_fds. After the next cycle, this new client will be added.



```
write(client_fd, "$register", strlen("$register"));
read(client_fd, discard, MSG_LEN);
write(client_fd, "failed", strlen("failed"));
read(client_fd, discard, MSG_LEN);
write(client_fd, "Username already taken\n", strlen("Username already taken\n"));
```

Figure 11:

The snippet of function to create read/write blocking

Both the server and the client are blocked. The recv and read functions block by default meaning that are synchronous and will wait for input before continuing. Because the client code is probably faster to execute than the server code, the signals must be blocked with read(), then write(), then read(), until the server and client finish talking to each other. The select() function also blocks all inputs until then.

4.2 Detailed Description of Functions and Parameters

Client

void PressEnter_Login(**GtkWidget** *widget, **GtkWidget** *entry)

- *widget refers to the text entry box where enter is pressed
- However *widget and *entry are not important as any text entry box can trigger this event on the log-in window.
- This event is triggered by pressing the "enter" key when either of the text entry boxes is selected. The text within both entry boxes is then converted into a string and compared to an empty string to check whether or not the text entry boxes are empty. If not, then the two strings from the entry boxes are searched up in the account information .txt file and scanned to see whether or not the account login information matches. If not, an error message will display, if there is a match, then the client will log in to the system.
- The .txt file is received from the server, in which it would turn on a flag to indicate that a successful connection has been made and will then read from the server for the username and password combination.

void resetTextEntry()

- Makes all of the text entry boxes become blank, through the use of editing globals.
- This event is called whenever a new window opens that contains a text entry box. It will wipe out all text inside ALL text entry boxes in the program, even the text entry boxes that are not shown on the current window.

- This event purely occurs in the client and there are no transfer of data from the client to the server or vice versa.

void entry_toggle_visibility(**GtkWidget** *checkbutton, **GtkWidget** *entry)

- *checkbutton refers to a button that is being pressed while *entry refers to the text inside the password entry box
- This function makes the text inside the password entry box to be hidden. All characters will become dots instead. It uses a GTK function to "turn on" text visibility and "turn off" text visibility. This is because text visibility is defaulted to be on by default. By turning it on and turning it off, the text will no longer be visible.
- This event purely occurs in the client and there is no transfer of data from the client to the server or vice versa.

gulong g_signal_connect(**GtkWidget** *widget, **char** *condition, G_CALLBACK(**void** *event), **void** *default);

- *widget refers to the widget that the signal is being connected to, *condition refers to the condition where the signal will be activated, *event refers to the event that will occur once the conditions are met, and *default can be a GtkWidget or can be NULL, depending on the circumstances of usage.
- This function connects a signal to a widget that performs a particular function.
- This event purely occurs in the client and there are no transfer of data from the client to the server or vice versa.

void gtk_widget_hide (**GtkWidget** *widget)

- This function passes a GtkWidget through its parameters to hide that widget. Hiding the widget makes it invisible to the user. If any parent widget is hidden, all of its child widgets are also hidden too by default. This widget can be a window, a button, a text box, etc.
- This event purely occurs in the client and there are no transfer of data from the client to the server or vice versa.

void gtk_widget_show_all (**GtkWidget** *widget)

- This function passes a GtkWidget through its parameters to show that widget. By showing the widget, it recursively displays all child widgets within the widget, such as a window with all of the widgets inside of it. This widget can be a window, a button, a text box, etc.
- This event purely occurs in the client and there are no transfer of data from the client to the server or vice versa.

void DrawBoard()

- This is used for ZotChess inside of ZotChat, where the chessboard is drawn using image files onto a table. This table is then uploaded as a widget onto ZotChess window.
- This event purely occurs in the client and there is no transfer of data from the client to the server or vice versa.

Server

Int LoginUser**(int** client_fd, **int** message_size, **char** client_username[10][64], **int** max_c**)**

- Takes a string with the username and password, or if the user wants to create a new login ID
- If the user wants to make a new account, then it will first loop through the Users/UserList.txt file and search if the username is taken, if not then it will create the username and create a text file for the user containing the password
- If it is an existing user, it will try to the username and attempt to log in
- 0 is returned when login failed, -1 will be returned if password not found, -2 will be returned if the username already exists and 1 will be returned if successful
- Note: The user must log in before they do anything else
- This occurs on the client side initially and will pull information from the server to validate whether certain conditions are met or not. If a new user is created, the

client will push the new username and password combination for the server to receive.

Int VerifyUserName(**char** *username)

- Opens "./Users/UserList.txt" and scans for matching username entry with database
- If there is no match, function returns 0, else 1.
- Close file

Int VerifyPassword(**char** *username, **char*** password)

- Opens "./Users/UserList.txt" and first scans for username entry match
- If the next line of matched user, the password for that user, does not match, returns error message that a username was found but password string does not match.
- In the case of both matching username and password, return 1, else 0.
- Close file

Void AppendUser(**char** *username, **char** *password)

- Takes newly created username and password entries and append to database "./Users/UserList.txt"

int* ParseString(**char*** signal**)**

- Takes a string with the communicated message from the client or the server
- Uses string tokenizer to separate the string by spaces
- The first part of the string will then be compared using strcmp()
- Certain commands such as message, login, register, play, will be recognized
- Returns an int* with an int detailing the action and the string converted into an int

int Initialize()

- Read all the user data from a .txt file.
- The file is constructed in order like "username$password$friend1$friend2\n"
- All the online condition and socket will be set to 0

- Takes in a .txt file as input and outputs a 1 if successful
- This event purely occurs in the server and there are no transfer of data from the client to the server or vice versa.

int AddFriend();

- Takes in a string as input from the text entry box
- AddFriend function allows users to add friends by inputting username
- Return 1 if succeed, 0 if fail
- If the AddFriend() function is activated, the client will send a string up to the server to check the list of available users. If there is a targeted username that has not been added to the client's friend list yet, the server will set a 1, if not the server will send back a 0.

4.3 Detailed Description of Communication Protocol

Please note that the communication of client-server is still in the works. You must run server first, then run client. Afterwards you must create the client first, and the server after for the program to work. The program does not have a way to exit out of communication as of this version, please use Control+C to suspend the program and exit out from there.

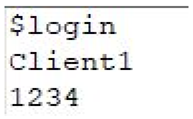
String Parsing Protocols

When the server and client communicate, they send and receive strings with each other. However, it isn't obvious what these strings mean or how either will interpret them. Thus, a string parsing protocol is made so that the client and server can understand each other. As of the Alpha Version, these are the commands that are supported.

Client To Server

$login

In order for the client to chat, play chess, or talk to friends, it must first log in to the server. The client will first type in “$login” and then hit enter. Then he/she will type in his username, press enter and enter his/her password. This is an example of what the client would type if he had a username of “Client1” and a password of “1234”.



```
$login
Client1
1234
```

Figure 12:

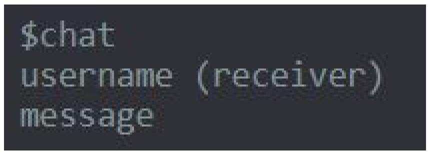
Example login input

The server will wait until it receives all of the inputs before doing something else. It will wait indefinitely for the user to finish typing their login information. The server will receive this information and then send the user some messages back regarding how the login process went. Please refer to **Server to Client** section below for more details.

$chat

The user first must have logged in before he can chat. The chat format starts off by taking the argument “$chat ” and the user is expected to hit Enter. The user will then type in the message that he wants the other client to see, and hit Enter. After doing so, the server will expect the username of the client he wants to talk to.

Assume the situation that both “Client1” and “Client2” are logged in and those are their usernames. An example of what the Client should type is shown in Figure 13 below.

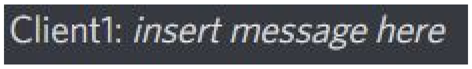


```
$chat
username (receiver)
message
```

Figure 13:

Example chatting input

Client2 would see this message:



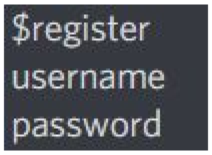
```
Client1: insert message here
```

Figure 14:

Example receiving end output

$register

In order for the client to be able to log in, the client must first register. The client will first type “$register” and then hit enter. Doing so would prompt the server to initiate the registration process. The user must then type in his username, hit enter, and then type in the desired password.



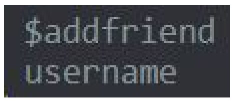
```
$register
username
password
```

Figure 15:

Example of register input

$addfriend

ZotChat supports an add friend function. Recall that a user can only message another user if they are friends. The client must first type “$addfriend” to initiate the adding friend process. After hitting enter, the client is expected to enter the username that is desired to be added.



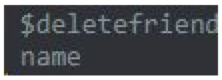
```
$addfriend
username
```

Figure 16:

Example of addfriend input

$deletefriend

ZotChat supports a delete friend function. The client must type “$deletefriend” to initiate the deleting process. After hitting enter, the user must enter the name of the friend that is to be deleted. ZotChat allows the delete friend process to be initiated offline.



```
$deletefriend
name
```

Figure 17:

Example of deletefriend input

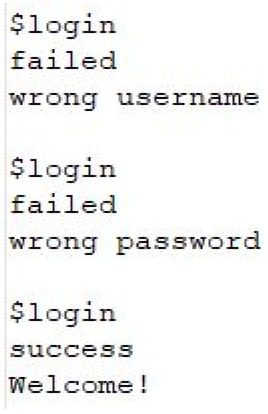
$pull

The “$pull” command reaches out to the person’s friends list, and verifies if each person on the list is online. The pull command is responsible for updating the count and concatenating the usernames of the online and offline users.

Server to Client

$login

The server will receive the login details from the client. Recall that the server will end up receiving three separate messages. One for the command “$login” (followed by newline), “username” (followed by newline), “password” (followed by newline). After typing in $login, the server will expect a username and password. It will wait indefinitely for these things. After receiving a username and password, it will put these in a char array. The server will call the Login() function. It will browse through its database and try to find the appropriate username in its UserList. If it appears, then the username is registered that it will have its own text file. It will go to that username’s file and then compare the password. It does both the username searching and password confirming by using getc() to grab the character and comparing the strings when a “\n” or “\r” is found. Login returns an integer, which will also help it decide which error message it would send (refer to section 4.2 for details). Some examples of what the server would send to the client are:



```
$login
failed
wrong username

$login
failed
wrong password

$login
success
Welcome!
```

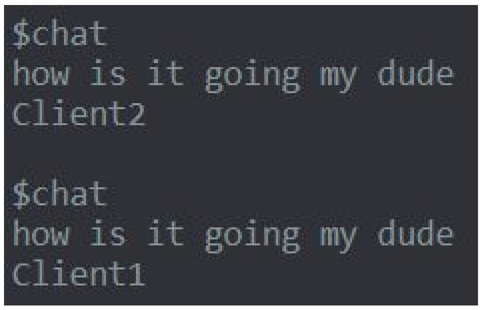
Figure 18:

Example of failed login

The client will also receive $login after attempting to log in. This is then followed by “success” or “failed” for whether the login succeeded or failed. If the login is failed, then the user must try again. If the login works, then the user can begin to use other functions. The next recv() statement the user will receive why they were able or unable to log in. If they were able to log in, they would receive the message “Welcome!” The client will handle this.

$chat

The program can detect for friends list. A user can only message another user if they are friends. The program does not currently handle offline messaging. If the user is online, the server will send the receiver the message in the same format it received it, except instead of showing the receiver it will show the sender. If Client1 were to send Client2 messages, the server would send the message below the second $chat text.



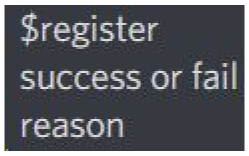
```
$chat
how is it going my dude
Client2

$chat
how is it going my dude
Client1
```

Figure 19:

Example of “returned” message

$register

The server will receive the registration details from the client. The server will end up receiving three separate messages. One for the command “$register” (followed by newline), “username” (followed by newline), “password” (followed by newline). After typing in $register, the server will expect a username and password. After receiving a username and password, it will put these in a char array. The server will call the Register() function. The server will then add the appropriate username in its UserList. If it appears, then the username is registered that it will have its own text file. Once the registration process is done, the user must then log in using the newly-registered credentials. The template of what the server would send to the client is portrayed through the photo below.



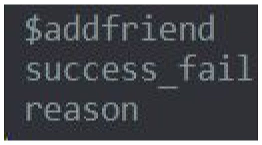
```
$register
success or fail
reason
```

Figure 20:

Template of “returned” message for register

$addfriend

Once the server receives the addfriend details necessary to add a friend. The server receives the command “$addfriend” and it also receives the “username” of the friend that is to be added. After receiving the necessary details, the server will call AppendingFriend() function which handles the adding of friends to the friendlist. Regardless of whether the addfriend process is successful or not, the server would send the client information about the status of the $addfriend command.

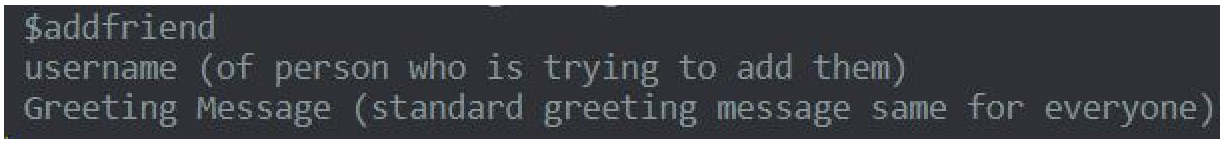


```
$addfriend
success_fail
reason
```

Figure 21:

Template of “addfriend” message for register

Another action that the server supports for the “$addfriend” command is a message sent by the server that notifies the newly-added user about his/her addition to another user’s friendslist. The server would send “$addfriend” (followed by newline), “username” of the person who is trying to add them (followed by newline), and a Greeting Message.



```
$addfriend
username (of person who is trying to add them)
Greeting Message (standard greeting message same for everyone)
```

Figure 22:

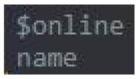
Example template of what the added client user receives from the server

$refresh

The server is now forked, so the refresh protocol allows the server to aid the refreshing of the GUI. This command sends the client a message called refresh which allows the Client to sleep occasionally and refresh for new updates.

$online

The “$online” enables the Client to verify if their friend is online. Another use for this protocol is when a newly-added friend turns online once he/she is added to the database. The server would send “$online” (followed by newline), “name.”



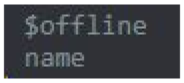
```
$online
name
```

Figure 23:

Example template of what the added client user receives from the server

$offline

The “$offline” enables the Client to verify if their friend is offline. Another use for this protocol is when a newly-deleted friend turns offline once he/she is deleted as a friend. The server would send “$offline” (followed by newline), “name.”



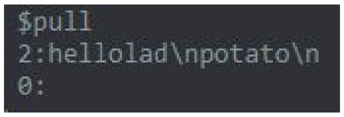
```
$offline
name
```

Figure 24:

Example template of what the added client user receives from the server for the $offline protocol

$pull

The pull protocol supports the online/offline handling of the Clients. The “online” and “offline” strings both show the number of people followed by a colon. Each name is separated by a newline. This shows all their friends, both online and is offline. The most common instance of “$pull” is called after logging in.



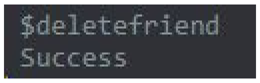
```
$pull
2:hellolad\npotato\n
0:
```

Figure 25:

Example what the Client receives after the $pull command is called

$deletefriend

Once the server receives the deletefriend details necessary to add a friend. The server receives the command “$deletefriend” and it also receives the “username” of the friend that is to be deleted. Regardless of whether the deletefriend process is successful or not, the server would send the client information about the status of the $deletefriend command.



```
$deletefriend
Success
```

Figure 26:

Example of what the Client receives when the $deletefriend call is successful

Server to Client

Server to Client is defined as the interactions between server and client that involves the transferring of data. The server will manage everyone’s data, even when the clients are offline. The server will actively pull data from clients and store them in files. The clients, however, will push and pull data, pushing data whenever a request is needed to be sent, such as a request of contact or a message. The server is always pulling at certain intervals, while the clients are always pushing and pulling at certain intervals, or whenever a request is needed to be urgently sent.

Initially, the client attempts to make a connection with the server when logging in. If there is no server or the server is busy, the client will experience a connection failure after a timeout. Basically, the client will request a connection to the server and the server will accept (or decline or ignore) the request. If the server accepts the request, it will send back a message to the client saying that the request has been accepted and that the client can log in, if and only if the client inputted the correct account information. Once in the home window, the client will routinely push and pull data from the server to see if there are any new information available to the client. The server will pull data that is pushed from the client.

The client compresses their information into multiple strings and sends them to the server. Within the information, there is a list of the friend's list to be updated, all messages in the conversation, as well as any pending friend requests, all stored under the client's profile in the .txt file in the server. All usernames have specific and unique information available to only them. This is the synchronous updating system implemented in ZotChat, where the information pushed to the server is pulled by the server, and then pulled by the client in order to forward information. The client will send an updated list of all of the information as a "save state" it currently has to the server in order for the server to update and additionally pull information to check for any updates. This way the client and server will both be up to date with the most relevant information possible.

To send information such as accepting a request, the client will send the flag to indicate that the request has been accepted to the server, and the server will update it's .txt file for the two clients and add each other onto the friend's list. The clients on both sides will pull this new information and have an updated friend's list on both sides. By updating automatically after the client attempts to communicate through the following, but not limited to: messaging, accepting a friend request, sending a friend request, sending a chess request, etc. the client will have no delay to push the information, rather than waiting for the next clock cycle to send. After the client sends the information, the server will still pull at it's clocked time, but by sending the newer bits of information, there is a lesser chance of collision and loss of packets when transmitting data to the server.

GUI to GUI

GUI to GUI based on client and server is the interactions of handling on the GUI including, but not limited to, clicking a button, submitting text entries, etc. If text entry is submitted, it is first processed by the GUI on the client side and then pushed to the server to store. Note that the GUI is not fully linked to the client/server interactions yet, but this will change by the final release.

The PressEnter_Login function reads all of the text entry boxes on the login window and checks with a .txt file received from the server to verify that the account information is all present when the user tries to log in. If the account information is not there, then the user will not be able to log in.

The resetTextEntry function resets all of the text entry boxes to be blank upon entering a window where there is a text entry box(es) present.

The entry_toggle_visibility function simply targets a particular widget and makes all the text inside of the widget to be hidden as dots. In this case, the particular widget is the password text entry box on the login window. Anything typed in the password text entry box will be marked as dots, however, the terminal can still read

The g_signal_connect function connects a widget to a function and connects a trigger to the function, with an additional widget that can be connected as collateral. The function takes in the first widget, the main widget, as the widget that will trigger the event. The trigger is the event that happens to the widget in order to execute the function. The function is up to the user to implement and program so that the widget will perform the desired algorithm when it is triggered. An additional widget can be connected as part of the function since the function can use another widget as a parameter to perform whatever that is implemented.

GUI to Board

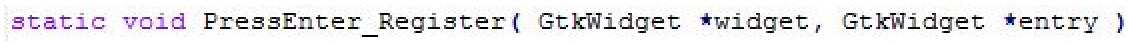
GUI to Board is defined as the interactions when the user clicks on the board part of the GUI. GUI to GUI interactions are all the other interactions that should be handled in the GUI including but not limited to clicking a button, starting a new game, and scrolling through the moves recorded.

This is only an occurrence in the ZotChat implementation of ZotChess, which is not included in this version.

When the user makes two inputs on the board, these coordinates will be converted into coordinates on the chessboard. A coordinate (321, 1093) may be converted to e2 because that is the position on the chessboard. The two inputs represent the two positions that the user will be inputting. The program will then concatenate the two inputs together into a string such as ‘e2 e4’ with a space in between. In case of promotion, the input would be ‘e2 e4 q’. This input will then be handled by the board.

The board handles the inputs and will receive the input from above. The code is written in C and uses a string tokenizer (stok in the string class) to parse the elements. These things will be converted into readable coordinates for the 2D array. The ReadInput function then returns a pointer with the first coordinate and second coordinate.

After checking if the move is valid and if there is check or not from the move, the reading log file function will take these inputs in as well as the array from reading the input. It will find the piece and then store the move in standard chess notation. The initial position is dropped because that is not standard chess notation. A brief explanation through an example is the pawn moving from e2 to e4. The program will store e4 and not e2 or e4. At the end of the game, the log will automatically be saved as a text file called ‘Logs.txt’. The moves will be logged with the move number followed by White and then Black’s move. In case if the game was imported, it will start at the specified turn. If for whatever reason, it’s black’s move, white’s move will be logged as ‘...’ (standard chess notation).

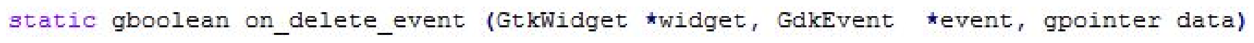


```
static void PressEnter_Register( GtkWidget *widget, GtkWidget *entry )
```

Figure 27:

The snippet of function to allow the client to create their accounts

Where *widget* is a GtkWidget struct. However *entry* is not relevant in this function, but Gtk functions require a second parameter. This function reads all of the text entry boxes on the register window and runs them through an algorithm to determine whether or not the user can create their account based on identical usernames and re-entry of password for confirmation.



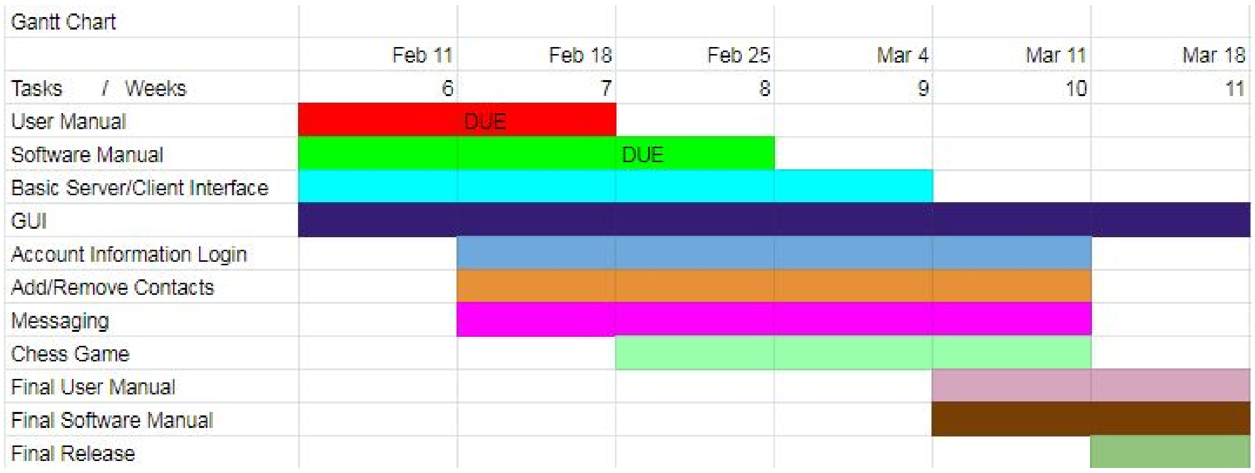
```
static gboolean on_delete_event (GtkWidget *widget, GdkEvent  *event, gpointer data)
```

Figure 28:

The snippet of function to exit out of the program

There will be many windows that are manipulated in ZotChat, so having a function to easily quit out of all the windows and exit the program is necessary.

5.0 Development Plan and Timeline



Gantt Chart						
	Feb 11	Feb 18	Feb 25	Mar 4	Mar 11	Mar 18
Tasks / Weeks	6	7	8	9	10	11
User Manual	■	■ DUE				
Software Manual	■	■	■ DUE			
Basic Server/Client Interface	■	■	■	■		
GUI	■	■	■	■	■	■
Account Information Login		■	■	■	■	
Add/Remove Contacts		■	■	■	■	
Messaging		■	■	■	■	
Chess Game			■	■	■	
Final User Manual					■	■
Final Software Manual					■	■
Final Release						■

Figure 29:

Gantt chart, dividing up the days to work on certain aspects

5.1 Partitioning of Tasks

Graphical User Interface: Bryan, Jingtian, Johnathan, Roy
Data Structures - Client/Server: Kevin, Zihao

Each of the six developers is a contributor to the User Manual and Software Manual aspect of the project.

5.2 Team Member Responsibilities

Manager: Kevin Zhu
Recorder: Johnathan Tang
Presenter: Bryan Trinh - *Documentation Formatter*
Reflector: Jingtian He - *Submission, Documentation Formatter*
Reflector: Roy Ramos - *GitHub Manager*
Reflector: Zihao Zou

6.0 Copyright

University of California, Irvine
The Henry Samueli School of Engineering
EECS 22L - SOFTWARE ENGINEERING PROJECT IN C LANGUAGE
E4130 Engineering Gateway, Irvine, CA 92697
+1 (949) 824-5333

7.0 References

1. Walia, Dipesh. “Data Structures in C Programming Language.” *Conio.h | Programming Simplified*, Creative Commons Attribution, www.programmingsimplified.com/c/data-structures.

2. Chaudhary, Shashank. “Artificial Intelligence 101: How to Get Started.” *HackerEarth Blog*, 25 Sept. 2018, www.hackerearth.com/blog/artificial-intelligence/artificial-intelligence-101-how-to-get-started/.

3. Gtk. “The GTK+ Project.” *About GTK+*, The GTK+ Team, www.gtk.org/.

4. “Getting Started with GTK+.” *GNOME Human Interface Guidelines*, The GNOME Project, developer.gnome.org/gtk3/stable/gtk-getting-started.html.

5. Gale, Tony, and Ian Main. “GTK+ 2.0 Tutorial .” *GNOME Human Interface Guidelines*, developer.gnome.org/gtk-tutorial/stable/.

6. “Widget Gallery.” *GNOME Human Interface Guidelines*, developer.gnome.org/gtk2/stable/ch02.html

8.0 Index