



h5 to f7, checkmate!

# 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

<b>Glossary</b>	<b>3</b>
<b>1. Client Software Architecture Overview</b>	<b>3</b>
1.1. Main data types and structures	3
1.2. Major software components	4
1.3. Module interfaces	4
1.4. Overall program control flow	6
<b>2. Server Software Architecture Overview</b>	<b>6</b>
2.1. Main data types and structures	6
2.2. Major software components	7
2.3. Module interfaces	7
2.4. Overall program control flow	9
<b>3. Installation</b>	<b>9</b>
3.1. System requirements and compatibility	9
3.2. Setup and configuration	10
3.3. Building, compilation, installation	10
<b>4. Documentation of packages, modules, and interfaces</b>	<b>10</b>
4.1. Detailed description of data structures	10
4.2. Detailed description of functions and parameters	11
4.3. Detailed description of communication protocol	15
<b>5. Development plan and timeline</b>	<b>18</b>
5.1. Partitioning of tasks	18
5.2. Team member responsibilities	19
<b>Copyright</b>	<b>19</b>
<b>References</b>	<b>19</b>
<b>Index</b>	<b>20</b>

# 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.

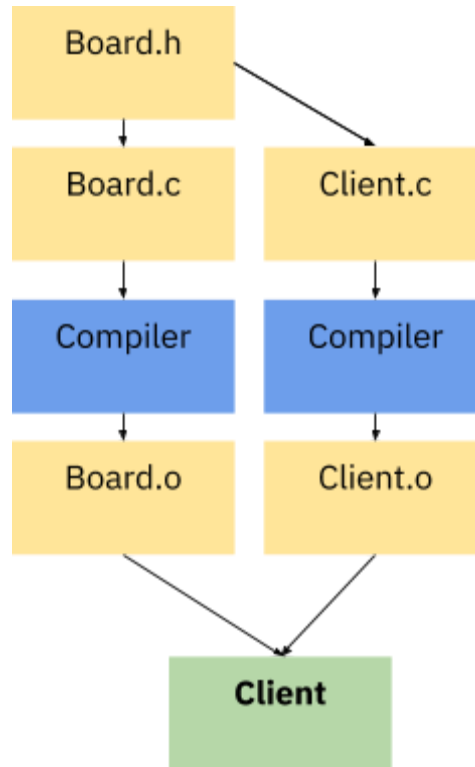
**Variable:** A variable is a specific stored space that the program manipulates.

## 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:

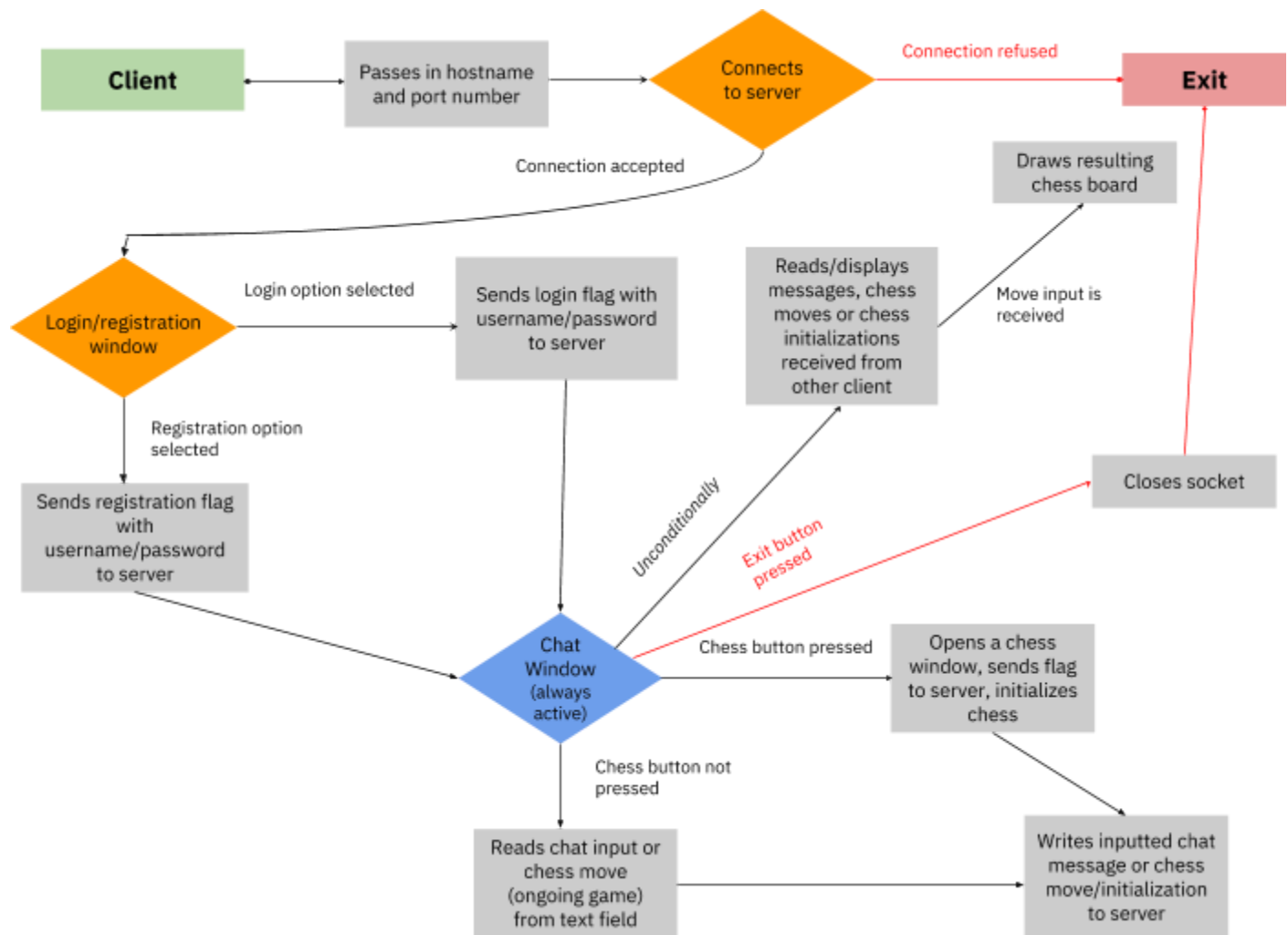
- **Client.c:** Module that creates a client that allows communication with a server that will transmit their message to their specified destination client. Operates by using multithreading; once a socket is established, the relevant functions listen for both messages coming from other clients via the 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.*

- **FriendsList.c:** Module that contains functions to update a user's friend list and friend request list like adding and removing friends and viewing both lists.
- **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 exchanged. 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

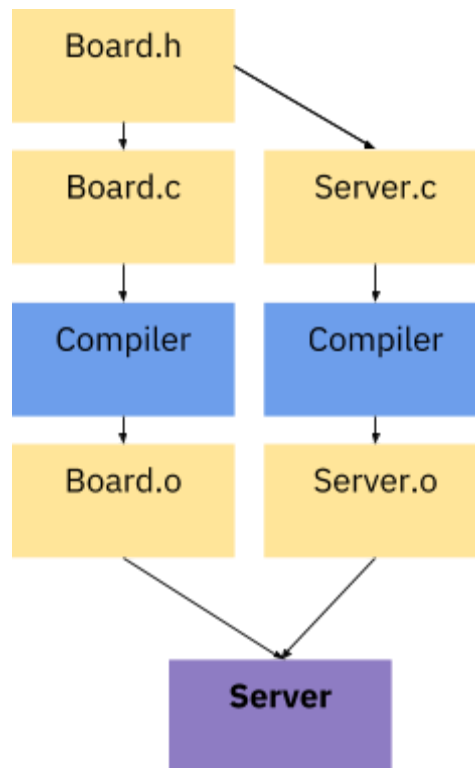


## 2. Server Software Architecture Overview

### 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.*

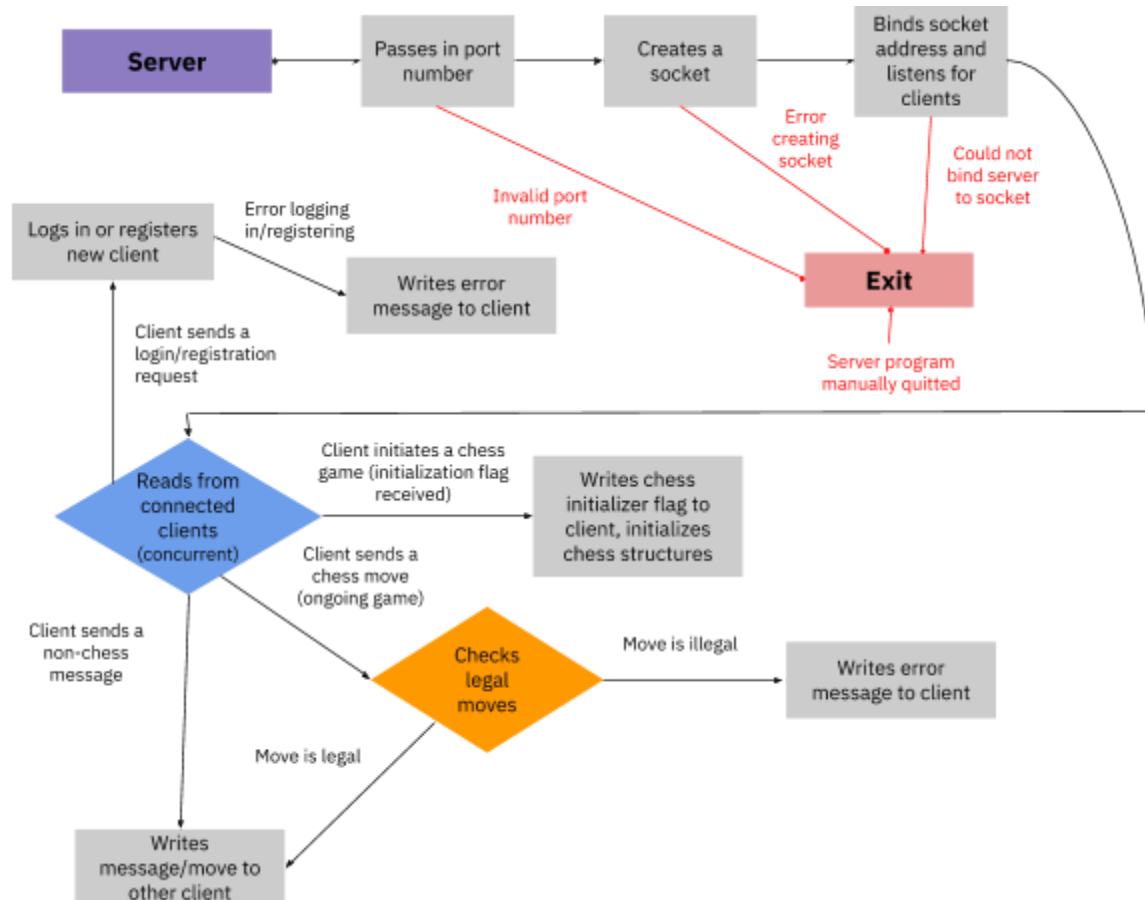
### Major network communication APIs:

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

- Output: void
  - Description: Prints an error message
- **SendtoAll:**
  - 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
- X11 window system installed (such as XMinig on Windows or XQuartz on macOS)
- Support for GTK+ 2.0 and Pango font library (see <http://macappstore.org/pango/> to install on macOS)
- 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. Enter the folder with all the executable files:  
**cd bin**
4. Run any executable as follows:  
**./GUIClient [hostname] [port number]** to launch the client program in GUI, from the same directory as the server  
**./Client [hostname] [port number]** to launch the client program in the command line interface, from the same directory as the server  
**./RemoteGUIClient [hostname] [port number]** to launch the client program in GUI, from any directory  
**./RemoteClient [hostname] [port number]** to launch the client program in the command line interface, from any directory  
**./Server [port number]** to launch the server program

# 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 Data {
    GtkWidget* table;
    GtkWidget* chat_entry;
    GtkWidget* main_window;
    GtkWidget* time_table;
    int current_location;
    int user;
    int Shutdown;
    int socket;
    char *Name;
    int colorflag; // 0 = std. color, 1 = green
};

```

The **Data** structure contains information useful for the ChatWindow.c module, including GTK data that is passed between functions in order to generate the GUI chat window screen. It also contains an integer representation of the current location of a chat message, and other user data, such as their socket number.

```

struct menuData
{
    GtkWidget *window;
    GtkWidget *entry_Login;
    GtkWidget *entry_PW;
    int sockfd;
};

```

The **menuData** structure contains information useful for the initial login menu. It contains GTK pointers for the window, login entry field, password field, and the socket of the client.

```

struct registerData
{
    GtkWidget *window;
    GtkWidget *entry_username;
    GtkWidget *entry_PW;
    GtkWidget *entry_CPW;
    int sockfd;
};

```

The **registerData** structure contains information useful for the initial registration menu. It contains GTK pointers for the window, login entry field, password field, confirm password field, and the socket of the client.

```

struct friendList
{
    GtkWidget *list;
};

```

The **friendList** structure contains a GTK pointer for a widget that represents a current client's list of friends.

```

struct sendData
{
    GtkTreeSelection *selection;
    GtkWidget *entry;
    char *name;
    int sockfd;
};

```

The **sendData** structure contains a GTK pointer for a username selection on the friends list tree view, a GTK pointer for an entry field (to add and remove a friend), a string for the name of a friend to add, and the socket of the current client.

## 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 Register(char \*username, char \*password, int socket)**

Receives variables from the Communicate function if the #REG# tag was used. Server checks the master file to search for a username with the same as the client input. If the program returns 0, then the user does not exist and the client is able to register under this new name.

**int Login(char \*username, char \*password, int socket)**

Receives variable from the Communicate function if the #LOGIN# tag was used. Server checks the master file to search for an identical username, and then compares the registered password to the password entered by the client. If the username does not exist in the file, then the function returns 2. If the client entered the wrong password, then the function returns 1. If both the usernames and the passwords match an existing user in the master file, then the user is allowed access to the program under that username.

**void SendtoAll(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 SendtoAll function.

**int MakeServerSocket(uint16\_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).

**void LoginFunction (const char username, const char password, int sockfd);**

Receives variables from the main client function if the client indicates that they want to login. The function saves the variables under the #LOGIN# flag, then sends the information to the server. The function then receives an integer regarding success.

**void Registration(const char username, const char password, int sockfd);**

This function allows the user to register by creating a username and password by using #REG# flag. Server checks the master file to search for a username with the same as the client input. If the program returns 0, then the username is already taken and the client is able to register under this new name.

**void MessageClient(const char username, const char password, int sockfd);**

This function allows users to message one another based on the socket that they are using.

**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.

## **FriendList.c & RemoteFriendList.c:**

**void CreateFriendListFile(char \*username);**

This function creates a unique friend list text file based on the passed in username by first creating the correct file name by combining the passed in username and the extension "friendlist.txt" then creating a new file by opening a new file based on that file name.

**void ViewList(char \*username, char \*list);**

This function displays the passed in user's friend list or friend request list by printing every character of the file.

**char GetFriendListName(char \*username);**

This function returns the name of the user's friend list text file by concatenating the username to the extension "friendlist.txt".

**int AddFriend(char \*friendusername, char \*yourusername, FILE \*friendsfile, FILE \*yourfile);**

This function adds both user's to each others friends lists if one user has accepted a friend request from another user by opening each passed in file and appending the friendusername or yourusername to it based on who's file is being edited. It returns a 0 if they have both been added to each other's lists successfully.

The file **RemoteFriendList.c** does not check if the user that the client adds exists.

**int RemoveFriend(char \*removeusername, char \*yourusername, FILE \*friendslist);**

This function removes a user from their friend list if they do not want to be their friend anymore. It does this by first copying each username in the file to a character array and then deleting the original file. It then gets the original name of the file by calling on the function GetFriendListName. It then creates a new file of the original name and copies each username in the character array unless it was the username of the user to be deleted. It returns a 0 if it has removed that user successfully.

## **chatGUI.c:**

**void ShowPassword( GtkWidget \*checkboxbutton, GtkWidget \*entry );**

This function is connected to a checkbox widget in CreateRegisterMenu and CreateLoginMenu. When the checkbox 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 checkboxes 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.

## **list.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.

**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 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.

For adding and removing friends, no flags are sent to the server. If a user indicates that they want to add or remove a friend, the functions are simply called on the client side.

Once the add button is clicked, the function AddFriend is called and adds both users to both of their friends lists. Once the remove button is clicked, the RemoveFriend function is called and deletes both users from both of their friends lists.

**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 server program sends the following flag to initiate a game of chess (when one client pushes the chess button):

**Server to Client:**

#STARTW# to the client that initiates the chess game (automatically the white player)

#STARTB# to the other client (automatically the black player)

If it is determined that a 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

© **2019 Chesster.** The rights to this software and its documentation belongs to the members of this team: Jada Berenguer, Benny Lin, Armand Ahadi-Sarkani, Raiyan

Nasim, Kevin Selda, Marvis Nguyen, Daniel Pajulas. All members of this team are enrolled as undergraduate computer engineering students in the Henry Samueli School of Engineering at the University of California, Irvine.

## References

*GNOME Human Interface Guidelines*,  
[www.developer.gnome.org/gtk2/stable/ch02.html](http://www.developer.gnome.org/gtk2/stable/ch02.html).

Bodnar, Jan. "ZetCode." *JavaScript Snake Tutorial - Creating Snake Game in JavaScript*,  
[www.zetcode.com](http://www.zetcode.com).

## Index

### A

API	3
Array	3

### B

Board.c	5, 7
Board.h	5, 7

### C

Call	3
Chat.c	5, 12
Chat.h	5,
ChatWindow.c	5, 13
ChatWindow.h	5
ChessInfo Structure	6
Client	3, 4, 5, 9, 10, 11
Client.c	4, 5, 10
Communicate	7, 10
Copyright	13

## **D**

Data Type 3

## **F**

FatalError 5, 8, 11, 12

Friend.c 5, 12

Friend.h 5, 8

Function 3, 10

## **I**

Initialize 3

Input 5, 8, 12

Installation 9

## **M**

Main 4, 6, 11

MakeServerSocket 8, 11

Module 3, 4, 7

## **O**

Output 5, 8, 12

## **P**

Pointer 3

## **R**

Receive 5, 11

## **S**

Server	3, 6
Server.c	7, 10
SendtoAll	8, 11
Structure	3, 4, 10

## **U**

User Structure	4, 10
----------------	-------

## **V**

Variable	3
----------	---