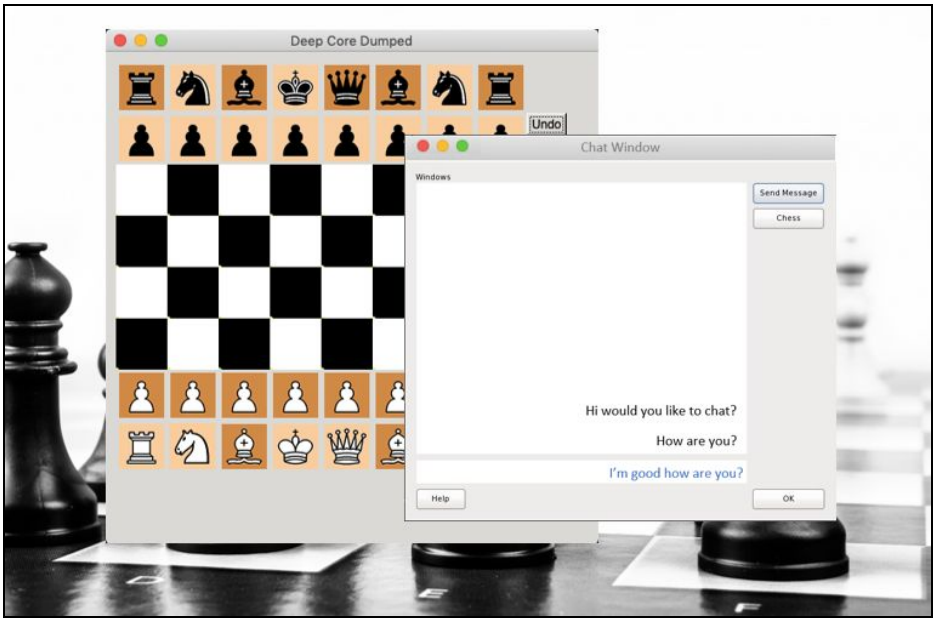
Chat & Chess

Software Specification Version 1.0



https://www.publicdomainpictures.net

Software Engineering In C
EECS 22L

Instructor
Quoc-Viet Dang

University of California, Irvine
Team # 4 Deep Core Dump
Ameya Pandit
Matthew Dunn
Richard Duong
Yuming Wang
Yunhe Shao
Xingjian Qu

Table of Contents

Table of Contents Continued

Glossary

- **Account:** A record of the users information, e.g. passwords, chat history, friends
- **Chess Window:** A window for the chess game between the user and other clients
- **Client:** The user of the program
- **Client Program:** The program that the client side will use
- **Graphical User Interface:** A.K.A. GUI, The graphical and visual aspect of the program allowing the user to interact
- **Message Window:** A window showing the chat between the user and other clients
- **Login Window:** The window the user first sees to login or make an account
- **Online Contacts Window:** The window holding the information for other clients
- **Port Number:** The port that is needed to connect to correct user
- **Register Window:** The window for registering an account to access the main program
- **Server:** The program hosting the chat program for the client
- **Server Program:** The program that the server side uses

1 Client Software Architecture Overview

1.1 Main Data Types and Structures

The client program uses a linked list to store the friends or contacts of a user. There are 2 main data structure components, Friend list and Message list. These data structures encapsulate the variables that keep track of the user data. The hierarchy of the data structure is shown in **Figure 1.**

1: Friend list. This is a linked list that stores the other contacts that the user is connected with. New contacts can be appended to this list at any time. Each friend entry allows for variables to keep track of that contacts user name as well as a chess game struct for playing chess (see chess software spec for details on the chess game struct).

2: Message List. This is a linked list that stores the message history between a user and another contact. The message list exists under each friend contact. The message entries under this list are between the user and the contact at the head of the message entry.

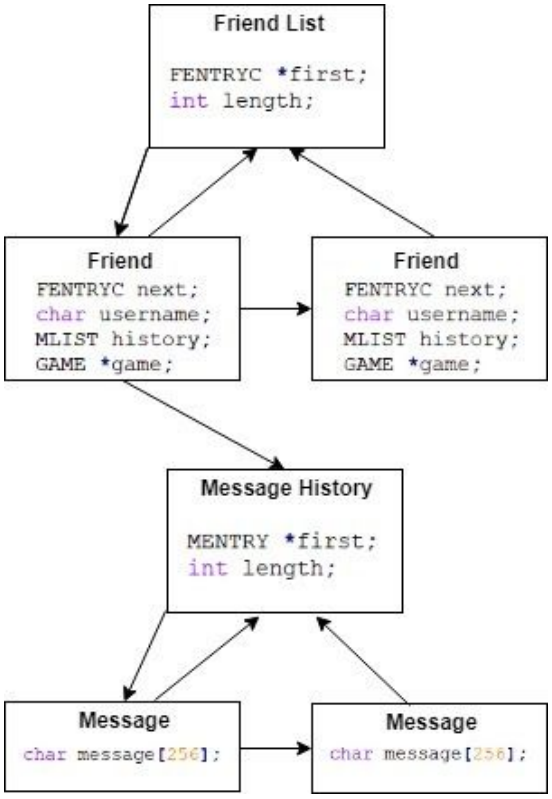


Figure 1: Client data structure

1.2 Major software components

The client program is decomposed into several modules which are compiled as object files. These files and the C library are then linked when the executable file is compiled. Each module has a header file and a source code file. The source code file ending in .c is the file that contains all of the functions that relate to the module. The header file ending in .h is the file that contains the function headers for all of the functions. The respective header file is included in each module were these functions are to be called. The hierarchical composition of the chat program is shown in **Figure 1.2**.

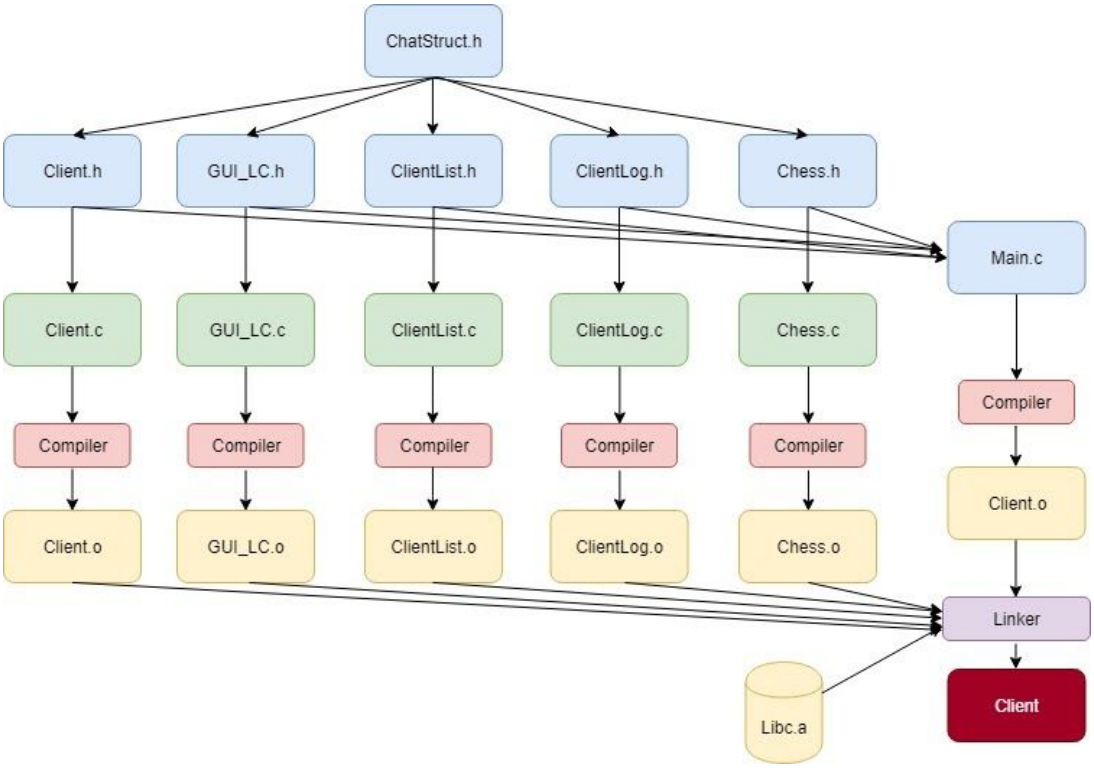


Figure 1.2: Client module hierarchy

1.3 Module Interfaces

1: Client.c

This module contains functions and other communications elements for the client side that allow it to recieve, process, and send data.

```
int Parsing(char *input, int length);
```

Input: Char pointer pointing to the input and the length of it
Output: returns an int that GTK can function with
Description: Parses the message that is inputted into it

```
void SendFReq(char *friend);
```

Input: char pointer to friend
Output: none
Description: Sends a friend request to a client through the server

```
Void Deletefriend(char *friend);
```

Input: Char pointer to a friend
Output: none

Description:Sends a message to the server to delete a friend

```
void SendMes(char *friend, char *message);
```

Input: char pointer to friend and char pointer to message
Output: none
Description: Sends message to a friend through the server

```
void ReceiveMes(CLIENT *client1, char *message, char *friend);
```

Input: CLIENT pointer to the client, char pointer to friend and char pointer to message
Output: none
Description: Receives message from a friend through the server

```
void AppendMes(CLIENT *client1, char *message, char *friend);
```

Input: CLIENT pointer to the client, char pointer to friend and char pointer to message
Output: none
Description: Appends the outgoing message to the message history

```
void ReqChess(char *friend);
```

Input: Char pointer to friend
Output: none
Description: Sends a request to play chess to a friend

```
void SChessMove(char *friend, char *pmove, char *move);
```

Input: Char pointer to a friend, and char pointers of both the previous and next move
Output: none
Description: sends the chess move to the other player

```
int ChessReq(char *friend);
```

Input: Char pointer to friend
Output: an Integer
Description: Receives a chess request and ask client to accept or decline

```
void RChessMove(char *friend, char *pmove, char *move);
```

Input: Char pointer to a friend, and char pointers of both the previous and next move
Output: none

Description: Receives the chess move of the other player

```
void RecFreq(char *friend);
```
Input: Char pointer to friend
Output: none
Description: Receives a friend request from a friend

```
Char *update();
```
Input: none
Output: Char pointer to a buffer
Description: Checks for updates from the server and puts it in a buffer

```
void strcpyn(char *str1, char *str2);
```
Input: Char pointer to first string and char pointer to second string
Output: none
Description: Like strcpy but ends at "\n" as well

```
void Writetoserver(char *message);
```
Input: Char pointer to message
Output: none
Description: Sends the message to the server

```
char *Readfromserver(char *buffer);
```
Input: Char pointer to buffer
Output: Char pointer
Description: Checks for any messages from the server and places into a buffer

2: GUI_LC.c

This module contains the GUI functions for the chat windows, chess board and all of the functions to handle signals from the user input.

```
void InitBoard()
```
Input: none
Output: none
Description: Initialize the chess board

```
void DrawBoard()
```
Input: none
Output: none

Description: Function use link all Graphic content with the pieces in board array.

```
void GUIMove(int s_x, int s_y, int g_x, int g_y)
```

Input: s_x, s_y: Mouse click location
Input:g_x,g_y:Mouse release location
Description: Function that determine which pieces that is selected and where the piece is moving.

```
void CoordToGrid(int c_x, int c_y, int *g_x, int *g_y)
```

Input: int s_x, int s_y: Mouse click location
Input: intg_x, int g_y:Mouse release location
Output: none
Description: Function that takes mouse click action and change the output pointer to coordinate value.

```
void chang_background(GtkWidget *widget, int w, int h, const
gchar *path)
```

Input: widget pointer, int height int width and gchar picture path
Output: none
Descripton: Changes the background of a specific window

```
Void hide (GtkWidget *widget)
```

Input: GtkWidget pointer to a widget
Output: none
Description: Hides the window associated with the widget

```
gint select_click(GtkWidget *widget, GdkEvent  *event,
                  gpointer  *data)
```

Input: GtkWidget pointer to a widget, GdkEvent pointer to an event, gpointer to data
Output: integer
Description: Detects the position of mouse click/release.

```
GtkWidget *Create_Board()
```

Input: none
Output: none
Description: Function that create a working chess board GUI window.

```
void Login_on_button_clicked (GtkWidget* button,gpointer data)
```
Input: GtkWidget pointer to button, gpointer to data
Output: none
Description: Function that read which button was clicked and a int data to specify which action should be taken for that button.

```
GtkWidget *Create_login_window()
```
Input: none
Output: GtkWidget pointer returned
Description: Function that create a working Login GUI window.

```
GtkWidget *Create_register_window()
```
Input: none
Output: GtkWidget pointer returned
Description: Function that create a working Rgisteration GUI window.

```
GtkWidget *Friend_list()
```
Input: none
Output: GtkWidget pointer returned
Description: Function that create a working Friend list GUI window.

3: ClientList.c

This module contains the functions to perform the needed tasks on the client data structures.

```
CLIENT *CreateClient(void);
```
Input: none
Output: Pointer to a CLIENT struct
Description: Creates the CLIENT struct and returns a pointer to it

```
FENTRYC *CreateClientFriend(char *username);
```
Input: char pointer to username
Output: pointer to a FENTRYC struct
Description: Creates a FENTRYC struct using the username

```
MENTRY *CreateMessageEntry(char *message);
```
Input: char pointer to message
Output: Pointer to MENTRY struct
Description: Creates a MENTRY struct and places the message in it

```
Void DeleteMessageEntry(MENTRY *entry);
```
Input: MENTRY pointer to entry
Output: none
Description: Deletes the MENTRY that entry points to

```
Void DeleteMessageList(MSLIST *list);
```
Input: MSLIST pointer to list
Output: none
Description: Delete the MLIST that list points to

```
Void DeleteClientFriend(FENTRYC *entry);
```
Input: FENTRYC pointer to entry
Output: none
Description: Deletes the FENTRYC that entry points to

```
Void DeleteClientFlist(FLISTC *list);
```
Input: FLISTC pointer to list
Output: none
Description: Delete the FLISTC that list points to

```
Void DeleteClient(CLIENT *client);
```
Input: CLIENT pointer to client
Output: none
Description: Deletes the CLIENT that client points to

```
FLISTC *CreateClientFList(void);
```
Input: none
Output: FLISTC pointer
Description: Creates a list for friends and returns a list pointer to the list created.

```
MLIST *CreateMessageList(void);
```
Input: none
Output: Pointer to MLIST
Description: Creates a MLIST that lists messages

```
void AppendClientFriend(FLISTC *list, FENTRYC *frnd);
```
Input: FENTRYC pointer to friend that is to be added, FLISTC pointer to list the friend is to be added to

Output: none
Description: Appends a new friend to the end of the list

```
Void AppendMessage(MLIST *list, MENTRY *message);
```
Input: MLIST pointer to list, MENTRY pointer to message
Output: none
Description: Appends a message entry to the message list

4: ClientLog.c

This module contains the functions to save the client data in a text file when client is shut down.

```
void Login (char *username, char *password, ULIST *list);
```
Input: char pointer to username, char pointer to password, pointer to userlist
Output: none
Description: sends login data to the server to login a user

```
void sign_up(char *username, char *password, ULIST *list);
```
Input: char pointer to username, char pointer to password, pointer to userlist
Output: none
Description: sends login data to the server to login a user

```
void logout(char *username, char *password, ULIST *list);
```
Input: char pointer to username, char pointer to password, pointer to userlist
Output: none
Description: sends logout request to the server and deletes all of the client

```
int AddFriendList(CLIENT *client, char *input, int increment);
```
Input: CLIENT pointer to client, char pointer to input, and an int
Output: an integer that GTK will use
Description: Adds friends based on the input to the client

```
void add_history(CLIENT *client, char *input);
```
Input: CLIENT pointer to client, char pointer to input
Output: none
Description: Adds the message history based on the input to the client

```
int AddUserList(CLIENT *client, char *input, int increment);
```
Input: CLIENT pointer to client, char pointer to input, and an int

Output: an integer that GTK will use
Description: Adds to the userlist based on input to the client

```
int AddMessageList(CLIENT *client, char *input, int increment);
```
Input: CLIENT pointer to client, char pointer to input, and an int
Output: an integer that GTK will use
Description: Adds the message list based on the input to the client

```
int DelFriendList(CLIENT *client, char *input, int increment);
```
Input: CLIENT pointer to client, char pointer to input, and an int
Output: an integer that GTK will use
Description: Deletes friends based on the input to the client

```
Void GetUsername(char *str1, char *str2);
```
Input: two char pointers
Output: none
Description: Obtains the username from the second string and places it in the first string

5: Main.c

```
int main(void)
```
Input: none
Output: none
Description: Main function for client contains loop for server connection and GUI functions

1.4 Overall Client Program Control Flow

Control flow on the client side starts with initialization of the data structures from a text file. The user is then allowed to login. After a successful login the data needed from the server is passed to the client to initialize needed information. Control is then passed to the GUI. Each time the user selects a feature that requires server contact the send receive process is executed. If the user does not make a request the receive timer loop will connect to the server periodically to receive updates. If the user logs out or timeout occurs then the client program will log out and the exit sequence will run to save user data.

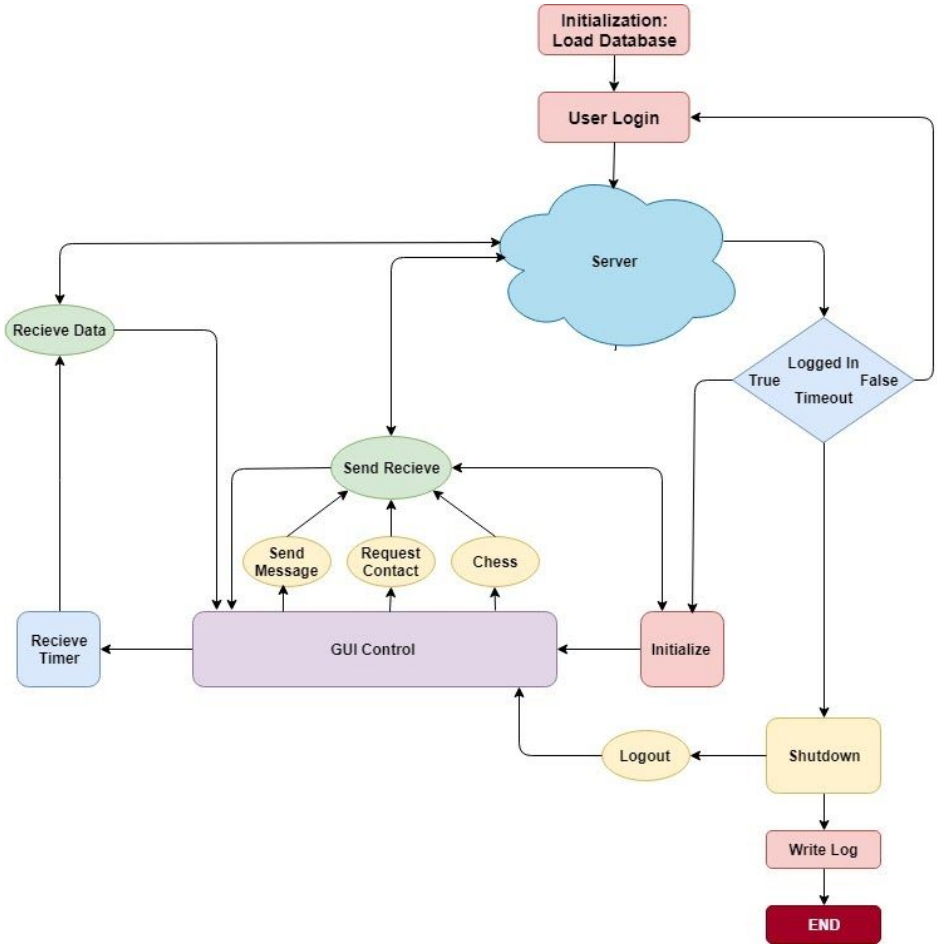


Figure 1.3: Client control flow

2 Server Software Architecture Overview

2.1 Main Data Types and Structures

The server program uses a linked list to store the accounts for each user. Each user account stores the variables that are needed for for login, the account user's friend list and the message history with each of the account holder's friends. There are 3 main data structures used to store this information. Account list stores the main accounts for each user. Friend list stores the contacts that the user has connected with. Message list stores the message history between a friend and the account holder.

1: Account List. Account list is a linked list that contains pointers to all of the accounts that are registered for the message application. The list contains a pointer to

the first account. Each account has variables to store the user data associated with that account such as password, username, the outgoing buffer for communicating with the user and the user friend list.

2: Friend list. This list stores the account holder's friends and the data associated with them. Each entry in the list stores the username of that contact and a message list for message history between the contact and the account holder.

3: Message list. The message list contains the message history between the account holder and the contact at the head of the message list. This list can be appended to whenever there is a message exchanged between that particular contact and the account holder.

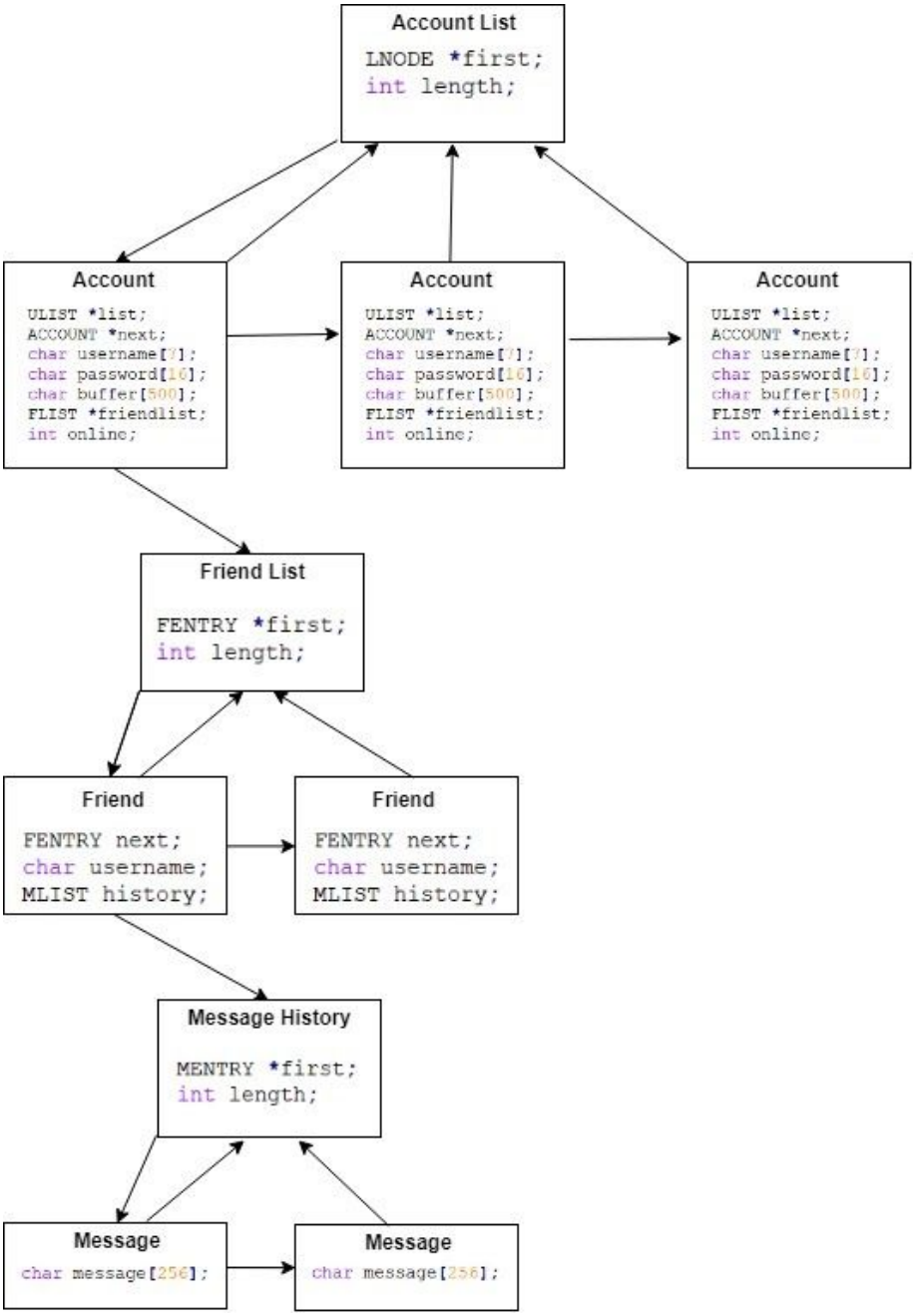


Figure 2: Server data structure

2.2 Major software components

The server program is decomposed into several modules which are compiled as object files. These files and the C library are then linked when the executable file is compiled. Each module has a header file and a source code file. The source code file ending in .c is the file that contains all of the functions that relate to the module. The header file ending in .h is the file that contains the function headers for all of the functions. The respective header file is included in each module were these functions are to be called. The hierarchical composition of the chess game is shown in **Figure 2.2**.

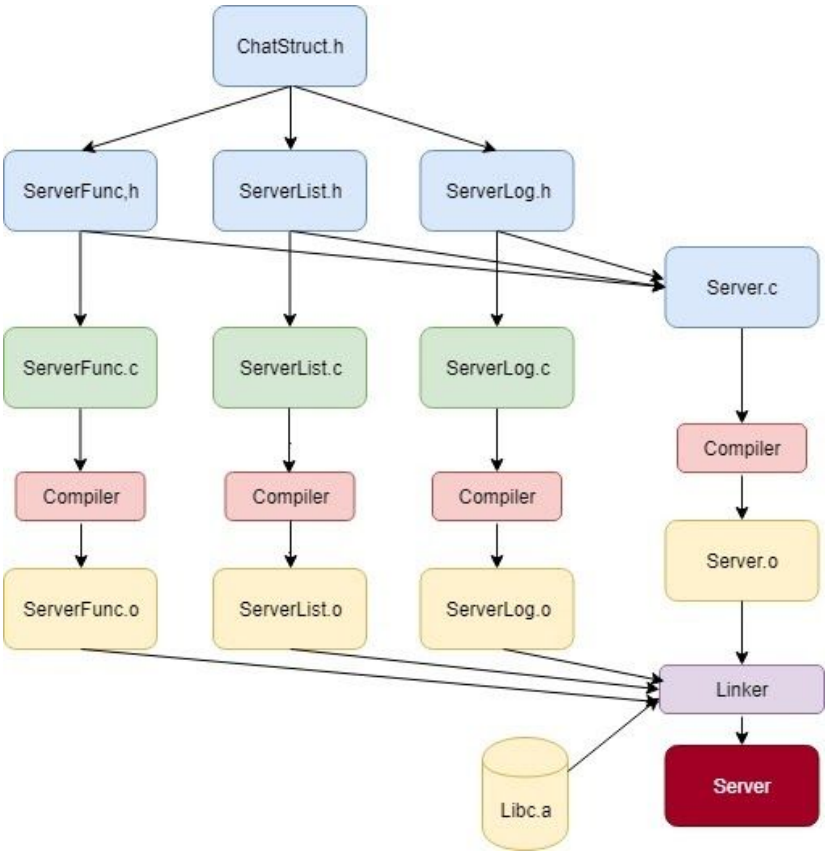


Figure 2.2: Server software modules

2.3 Module Interfaces

1: ServerFunctions.c

This module contains the functions that handle user requests from the server.

```
int LoginUser(ULIST *userlist, char *username, char *password,
char *outbuffer);
```

Input: Pointer to userlist, username and password coming from the client and the send buffer.

Output: int output returns 1 if FAIL and 0 if SUCCESS

Description: Checks the user login information and sends confirmation back to the client

```
int LogOutUser(ULIST *userlist, char *username, char
*outbuffer);
```

Input: Pointer to userlist, username and coming from the client and the send buffer.

Output: int output returns 1 if FAIL and 0 if SUCCESS

Description: Logs out a user and sends confirmation back to the client

```
void ProcessInStream(char *recieve, char *send, int length,
ULIST *userlist);
```

Input: ULIST pointer to user list and pointer to username of client to be processed, char pointers to the incoming and outgoing server buffer and length of the buffer.

Output: none

Description: Process a clients outgoing data that is stored in their buffer

```
void UserUpdate(ULIST *userlist, char *senderUN, char
*outbuffer);
```

Input: ULIST pointer to user list and pointer to username of client to be processed and outbuffer

Output: none

Description: Add data to clients buffer when client asks for update

```
void SendFriendList(ULIST *userlist, char *recieverUN, char
*send);
```

Input: ULIST pointer to user list and pointer to username of client to be processed and send buffer

Output: none

Description: Send the friend list to the user

```
int BufferMessage(ULIST *userlist, char *senderUN, char
*recieverUN, char *message);
```

Input: ULIST pointer to user list and pointer to username of client to be processed, pointer to username of the client the message is going to and send buffer
Output: int value, 1 for FAIL, 0 for SUCCESS
Description: Send the friend list to the user

2: ServerList.c

This module contains the functions for the server data structures, creating lists and accounts.

```
FLIST *CreateFriendList(void);
```

Input: none
Output: FLIST pointer
Description: Creates a list for friends and returns a list pointer to the list created.

```
void DeleteFriendList(FLIST *list);
```

Input: FLIST pointer to list to be deleted
Output: none
Description: Deletes a friend list and its sublist of messages

```
void AppendFriend(FENTRY *friend, FLIST *list);
```

Input: FENTRY pointer to friend that is to be added and FLIST pointer to the list to which the friend is to be added
Output: none
Description: Appends a new friend to the end of the list

```
void AppendAccount(ACCOUNT *account, ULIST *list);
```

Input: ACCOUNT pointer to the account that is to be added and ULIST pointer to the list of users to which the account is to be added
Output: none
Description: Appends a new account to the end of the list

```
void DeleteAccountList(ULIST *list);
```

Input: ULIST pointer to list to be deleted
Output: none
Description: Delete a account list and its sublists.

3: ServerLog.c

This module contains the functions on server-side to save the client data in a text file when client is shut down.

```
void loadLog(ULIST *list);
```

Input: ULIST pointer to list to be loaded

Output: none

Description: Loads the log to update the server-side to the conditions as they were before the server shut down.

```
void saveLog(ULIST *list);
```

Input: ULIST pointer to list to be saved

Output: none

Description: Saves a log of data so that the server can be shut down and the conditions before shutdown can be replicated.

```
void conversionSpace(char *string);
```

Input: char string to be manipulated

Output: none

Description: converts space characters in string to "~" characters.

```
void conversionSpaceBack(char *string);
```

Input: char string to be manipulated

Output: none

Description: converts "~" characters in string back to space characters.

```
void conversionNewline(char *string);
```

Input: char string to be manipulated

Output: none

Description: converts "\n" characters in string to "`" characters.

```
void conversionNewlineBack(char *string);
```

Input: char string to be manipulated

Output: none

Description: converts "`" characters in string to "\n" characters.

```
void spaceRemove(char *string);
```

Input: char string to be manipulated

Output: none

Description: removes extraneous space characters (in beginning of string and end of string).

4: Server.c

This module contains the main function for the server program and calls the other functions needed for server operation.

```
int MakeServerSocket(uint16_t PortNo);
```

Input: port number for socket creation.
Output: int
Description: Creates a server socket on desired port number

```
void ProcessRequest(int DataSocketFD, ULIST *userlist);
```

Input: int value for File descriptor, pointer to the userlist
Output: none
Description: Processes the request for an incoming client.

```
void ServerMainLoop(int ServSocketFD,ClientHandler
HandleClient,ULIST *userlist,int Timeout)
```

Input: int value for File descriptor of server socked, pointer to the function called to handle the client and int value for timeout.
Output: none
Description: Server main loop using select and timeout.

```
int main(void);
```

Input: none
Output: none
Description: Main function for server contains call to main server loop.

2.4 Overall Program Control Flow

Control flow for the server program starts with initialization of the data structures from a text file. Next the server opens a socket and starts listening to the socket. The server has a loop that allows for multiple users to make requests. Each time the user connects to a client the data is received, sent and then the client socket is closed. The clients are in a queue and the server uses a select to pick a client to service. The server then processes the client request according to the type. This loop occurs as

long as the server does not receive a shutdown request. Once shutdown is received the server writes all the data to a text file and then shuts down.

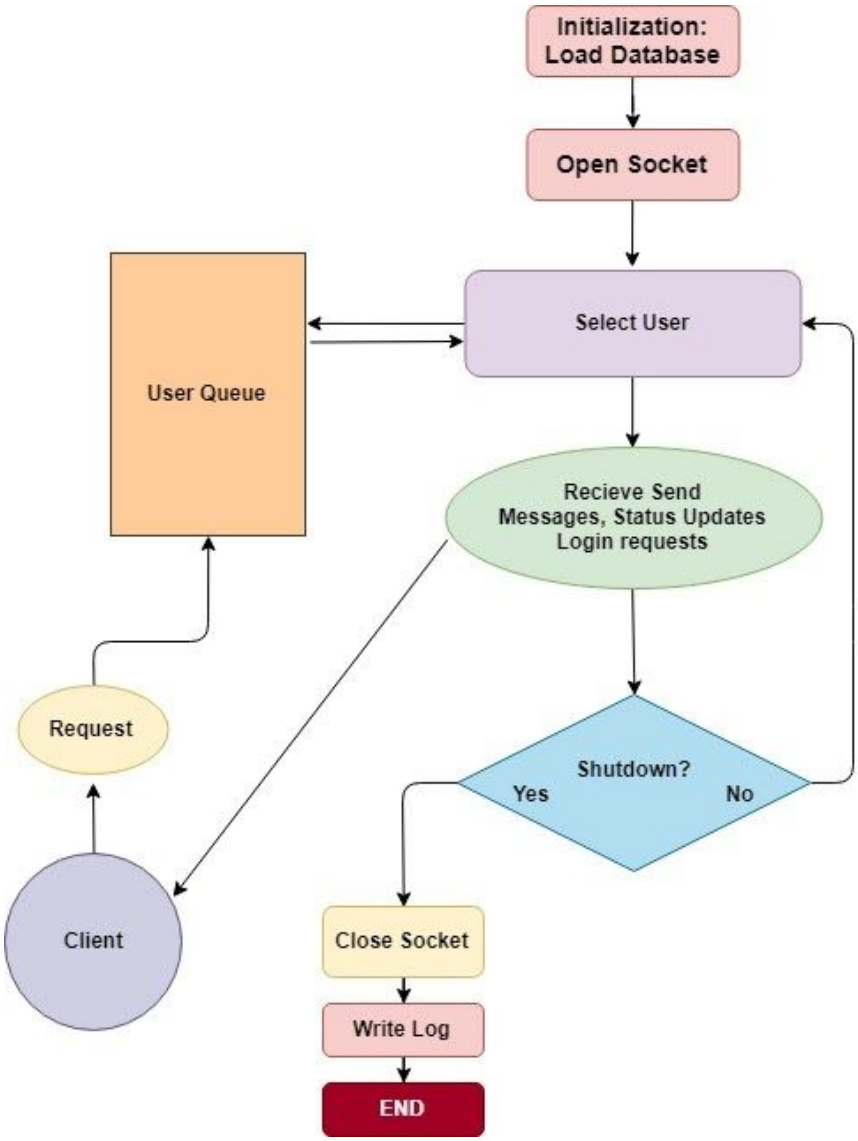


Figure 2.3: Server control flow

3 Installation

3.1 System Requirements, Compatibility

Hardware: Linux Server
Software Linux: CentOS release 6.10
More, the Linux application: MobaXterm

3.2 Setup and Configuration

The first step to setting up the program is to download the zip file.

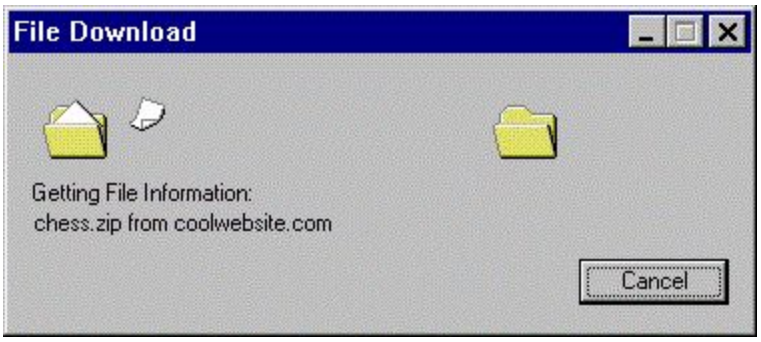


Figure 3: Downloading the zip file

Once the file has finished downloading, locate the zip file, for most web browsers the zip file will be placed in the download folder. Next, using an application to view zip files, open and extract all of the contents of zip file. Locate where the zip file was extracted. Once the zip file is extracted, there will be a Chat Program file, we copy and paste the file into our terminal, which is Linux, using the following command: "`gtar xvzf chatprogram.tar.gz`". Then switching to the directory /chat. Then type the "make" command to compile the code file to execute.

When the client side program is launched, the login window will appear and allow the client to login or register an account. When the server side program is launched, there will be a message saying that the server is up and running properly.

3.3 Building, Compilation, Installation

The zip file consist of : Makefile, Client.c, GUI_LC.c, ClientList.c, ClientLog.c, ServerLog.c, Chess.c, ClientMain.c, Server.c, ServerList.c, ServerLog.c, ServerMain.c and all of their respective header files. All the compilation is done within the makefile that is included in the zip file. When the command "`make`" is entered into the console, the object file for each c file is made and will link all the object files and make the executable file chat program.

Before launching the program, ensure that the MobaXterm program is running. If not, the gui may not launch and cause other problems when trying to launch the game. Running the executable file can be done with the command " `./Chat`". At this point the program will launch and the login window will be shown. To exit the chat program, simply close the window or click on "logout" in the menu.

To remove the unnecessary files from the directory where the zip file was copied to, type the command "`make clean`". All the object files and the executable file will be removed, leaving only the files extracted from the zip file.

4 Documentation of Packages, Modules, Interfaces

4.1 Detailed Description of Data Structures

```
typedef struct UserList{
    ACCOUNT *first;
    int length;
} ULIST;
```

Figure 4: User account list

The UserList struct contains the account list of users that are registered in the chat application. *first is a pointer to the first account in the list. Length is a variable that contains the length of the list.

```
typedef struct FriendList{
    FENTRY *first;
    int length;
}FLIST;
```

Figure 5: Friend list struct

The FriendList struct stores the client friend list and is used only in the server program. It contains the length of the list and also a pointer to the first friend entry in the list.

```
typedef struct MessageList{
    MENTRY *first;
    int length;
}MLIST;
```

Figure 6: Message list struct

The MessageList struct is used for storing the message history from friends and is used in the client program as well as the server program. It contains a pointer to the first message entry in the list and also the length of the message list.

```
typedef struct FriendListClient{
    FENTRYC *first;
    int length;
}FLISTC;
```

Figure 7: Client friend list struct

The FrientListClient struct contains the friend list used in the client program. It contains a pointer to the first entry in the client friend list and the length of the list.

```
typedef struct Account{
    ULIST *list;
    ACCOUNT *next;
    char username[7];
    char password[16];
    char buffer[500];
    FLIST *friendlist;
    int online;
}ACCOUNT;
```

Figure 8: Account struct

The Account struct stores the account information and friend list for each of the account holders. The *list pointer points to the head of the list, *next pointer points to the next account in the list, username and password contain the registered username and password for the account holder, and *friendlist points to the user's friend list. The buffer is the location where all communication that is outgoing is stored until the user connects to receive. Online is a int for the user status, 0 for logged out, 1 for logged in and 2 for inactive.

```
typedef struct FriendEntry{
    FENTRY next;
    char username;
    MLIST history;
}FENTRY;
```

Figure 9: Friend list entry struct

The FriendEntry struct is for storing entries in each user friend list and is only used in the server program. The *next pointer points to the next friend in the list, username

contains that friends username information. *history is a pointer to the message history list between that friend and the account holder.

```
typedef struct FriendEntryClient{
    FENTRYC *next;
    char username;
    MLIST *history;
    GAME *game;
    int Online;
}FENTRYC;
```

Figure 10: Friend list entry client side

The FriendEntryClient struct is for storing entries in each user friend list and is only used in the client program. The *next pointer points to the next friend in the list, username contains that friends username information. *history is a pointer to the message history list, *game is a pointer to a chess game struct and is used if the client wants to play a game of chess with the user. Online is a integer that stores the friends login status, 0 for logged out, 1 for logged in and 2 for inactive.

```
typedef struct MessageEntry{
    char message[256];
}MENTRY;
```

Figure 11: Message entry

The MessageEntry struct stores a message in the message history list. The struct contains one variable, a char array that stores a message up to 256 charachters.

4.2 Detailed Description of Functions and Parameters

GUI Functions

```
void InitBoard()
```
This is the function used to initialize the whole board. It will set up the color of each square and initialize the position of all pieces. No parameter or return value will be created by this function.

```
void DrawBoard()
```
This function is used to draw out the board. When called, the function will used pictures that are stored under chess_icon folder to get all pictures of pieces as well as the black and white square.

```
void GUIMove(int s_x, int s_y, int g_x, int g_y)
```
This function takes in the position of Mouse Click and Mouse release and change the related board section (delete previous board piece and add the piece to the new position).

```
void CoordToGrid(int c_x, int c_y, int *g_x, int *g_y)
```
This function is used to change the coordination of mouse click/release and change it to the relative position on board. (8*8 coordinate).

```
void chang_background(GtkWidget *widget, int w, int h, const
gchar *path)
```
This is a “modulus” function used to change the background of any GTK window. It takes into the widget pointer and path of the picture. All pictures were stored in chess_icon.

```
Void hide (GtkWidget *widget)
```
This function is used to hide the window if that widge need to be closed after certain action. For instance, it is used to close the login page after the user successfully login or choose to register.

```
gint select_click(GtkWidget *widget,GdkEvent *event,gpointer
data)
```

This is the function that takes the click action of the mouse to determine which piece in the board that the user want to choose to move. In respect, when user choose to release the button, that position where release action happen is the destination of this move.

`GtkWidget *Create_Board()`
This is the module function section for the board window and integrated all necessary functions to create a working board GTK window. When called, it will create a Board window on screen.

`void Login_on_button_clicked (GtkWidget* button,gpointer data)`
This is the function that used to determine all buttons' related function for the Login Page. Int 1 is for clear button, int 2 for login, int 3 is the "editable" box and int 5 is for register.

`GtkWidget *Create_login_window()`
This is the module function for the Login window that integrated all necessary function needed to cleated a working login in GTK window.

`GtkWidget *Create_register_window()`
This is the module function for the Register window that integrated all necessary function needed to cleated a working Register GTK window.

`GtkWidget *Create_chat_window()`
This is the module function for the chat window that integrated all necessary function needed to cleated a working login in GTK window.

`GtkWidget *Friend_list()`
This is the module function for the Friend List window that integrated all necessary function needed to cleated a working Friend List GTK window.

Client Authentication Protocol:

`void Login (char *username, char *password)`
This function is to let a client login their account. In this function, the client will be asked to type the username and the password. Then the client said sent the request

asking for the server side authentication of whether this account is exist or not. If there doesn't find this database on the server side, then this client will be asked to sign up a new account.

`void sign_up(char *username, char *password)`
This function is inside the login function, which helps a client create an account. The client send the request and the server side accept, the client write and the server read and then put the username and the password into the account structure.

`void logout(char *username)`
This function is to ask the client to logout the account he currently use. On the client side, he send the request of refresh the database and the offline statues buffer. The server side accept and read, then write the changes.

Server Authentication protocol:

`void Login(char *username, char *password, ULIST *list)`
This function is to determine whether the username and password that client side send to you is right or not. If the username right but the password not, it will send the warning that type the password again, otherwise it will send the massage of sign up a new one.

`void LogOut(char *username, ULIST *list)`
This function will log out the client on the server side and then communicate with the client that they are logged out.

Client Side Communication:

`void ReceiveFList(char *username, char* buffer);`
This function will be called when the client side program begins to receive a friends list from the server. The buffer will hold the friends list and set up the GUI for the specified username. The buffer holds the list of friends, their current status and a message log of each friend to a certain extent.

`void SendFReq(char *username, char *friend);`
This function will send a friend request to a the specific friend. The username will allow the server to determine who the friend request is being sent from.

`void SendMes(char *username, char *friend, char *message);`

This function will send a message to the specified friend from the username through the server. The message can hold up to 256 characters and will be sent through the server to the friend.

```
void ReceiveMes(char *username, char *friend, char *message);
```
This function will receive a message from the specified friend to the username through the server. The message can hold up to 256 characters and will be sent through the server to the username.

```
void FStatusChg(char *username, char *friend, int status);
```
This function will change the status of the specific friend of the username based on the status. Whenever the status of a friend changes, this function will be called to update the friends list of the current user and all friends of those on the client that had their status change.

```
void PlayerVPlayer(FENTRYC *friend)
```
This function will handle the chess game that is being played between a client and another user. The board for the game is stored under the friend entry passed to the function. The function will make the move on the board and then send the information to the server.

```
FLISTC *CreateFriendList(void);
```
This function creates a friend list in the client program and returns a pointer to the list created.

```
void DeleteFriendList(FLISTC *list);
```
This function will delete a friend list and the corresponding message list that is located inside each of the friend entries.

```
void AppendFriend(FENTRYC &friend, FLISTC *list);
```
This function will append a new friend to the end of the friend list. The corresponding message list is created inside the friend entry.

Server functions

```
int MakeServerSocket(uint16_t PortNo);
```
Creates a port for socket communication.

```
void ProcessRequest(int DataSocketFD, ULIST *userlist);
```

This function receives the input from the client and sends the output.

```
void ServerMainLoop(int ServSocketFD,ClientHandler
HandleClient,ULIST *userlist,int Timeout)
```
This function provides the server main loop using select and timeout. It selects between incoming clients and then calls the processing function to handle them.

```
int LoginUser(ULIST *userlist, char *username, char *password,
char *outbuffer);
```
This function logs in the user and sends the information such as friendlist, message history and userlist to the user.

```
int LogOutUser(ULIST *userlist, char *username, char
*outbuffer);
```
This function logs out the user in the server database.

```
void ProcessInStream(char *recieve, char *send, int length,
ULIST *userlist);
```
This function parses the input coming from the client and calls the appropriate functions for processing the data.

```
void UserUpdate(ULIST *userlist, char *senderUN, char
*outbuffer);
```
This function adds the data to the buffer when a client requests an update.

```
void SendFriendList(ULIST *userlist, char *recieverUN, char
*send);
```
This function sends the friendlist to the user.

```
int BufferMessage(ULIST *userlist, char *senderUN, char
*recieverUN, char *message);
```
This function buffers messages between users and adds the message to the message history log.

```
FLIST *CreateFriendList(void);
```
This function creates a friend list in the server program and returns a pointer to the list created.

```
void DeleteFriendList(FLISTC *list);
```

This function will delete a friend list and the corresponding message list that is located inside each of the friend entries.

```
void AppendFriend(FENTRY *friend, FLIST *list);
```
This function will append a new friend to the end of the friend list. The corresponding message list is created inside the friend entry.

```
void AppendAccount(ACCOUNT *account, ULIST *list);
```
This function will append a user account to the end of the user list. The account is passed to this function as an argument.

```
void DeleteAccountList(ULIST *list);
```
This function will delete an account list and all the entries in the list. The account friend list and corresponding message list will also be deleted when this function is called.

Server log functions:

```
void saveLog(ULIST *list);
```
This function saves the server-side data in event of a server shutdown so that the conditions before shutdown can be replicated, in effect limiting any data loss that might occur with shutting a server down.

```
void loadLog(ULIST *list);
```
This function loads the server-side data after a shutdown so that conditions before the shutdown are replicated, in effect creating a system with memory.

```
void conversionSpace(char *string);
```
This function converts the space characters in the input string to “~” characters. This is done as to avoid any issues with transmitting space characters.

```
void conversionSpaceBack(char *string);
```
This function converts the “~” characters in the input string to back to space characters.

```
void conversionNewline(char *string);
```
This function converts the newline characters in the input string to “`” characters. This is done as to avoid any issues with transmitting newline characters.

```
void conversionNewlineBack(char *string);
```
This function converts the “`” characters in the input string to back to newline characters.

```
void spaceRemove(char *string);
```
This function removes the extraneous space characters in the input string, removing these characters in the beginning and end of the string.

4.3 Detailed Description of the Communication Protocol

```
void ParseBuffer(char *inbuf, ULIST *list);
```
Once the connection is made to the server and the data is received either by the client or the server the ParseBuffer function is called to parse the data according to keywords.

The parsing is accomplished using keywords. Once a keyword in a stream of data is reached there will be a space and then the information pertaining to the keyword. If there is no information to convey after a keyword the keywork V_OID will be after the next space and the parsing command will move to the next keyword.

Keywords:

R_E_G: this keyword is used to register an account.

L_I: lets the server know that the client wants to login the next expected data will be username and password with \n for the ending character

L_O: lets the server know that the client wants to logout, the next expected data will be username and password using \n as the ending character

M_S_G: the information that comes next will be the message that the user will send.

F_N_R: this is the flag for a friend request the information that comes next will be the username of the friend that the client want to connect with.

D_F_N: this is the flag for deleting a friend.

F_N_A: the keyword for the accepted friend request

C_S: the information that comes next will be a chess game request.

C_M: the information that comes next will be a chess move. The expected format for the move is move from row,column move to row,column

U_P_D: lets the server know that the client will receive messages

S_N_D: the information that comes next will be the username of the client

R_C_V: the information that comes next will be the username of the friend the client wants to communicate with.

C_F_L: the next information will be the client's friend list sent from the server. The status is appended to the name as in username_ONLINE.

U_N: to send the username required for message history from the server to the client.

M_H: to send message history from the server to the client.

SUCCESS: this is the message that the server will send when communication is successful

FAIL: this is the message that the server will send when the communication fails

A typical communication looks similar to the following examples:

Login success message:
SUCCESS\nC_F_L\nUsername1\nUsername2\nM_H\nUsername\nS_N_D\nusername\nM_S_G\nHistory\nE_N_D\nM_H\nUsername\nS_N_D\nusername\n

Send client FriendList:
C_F_L\nusername_status\nusername_status\n.....\nE_N_D\n

Message for history :
M_H\nU_N\nusername\nmessage\nmessage\n.....U_N\nmessage\n.....

Send list of users: F_U_L\nusername_status\nusername_status\n.....\nE_N_D\n

Send message history: M_H_S

Friend request between clients:

Friend Request sent to server: F_N_R\nsenderusername\nrecieverusername\n This is the server command coming from a client that wants to request a friend. The server will receive this command and put it in the other clients buffer to send to them the next time they connect.

Friend request sent to client: F_N_R\nrecieverusername\n the client will receive the username from the client that accepted

Friend Accept: F_N_A\nsenderusername\nrecieverusername\n this message will be send from the user that accepts a friend request. The friend will be added to the clients list.

Login request: L_I\njames67 \npass677\n once the user is logged in the message of SUCCESS will be sent to the user. If the login fails then FAIL will be sent to the user.

Logout request: L_O\nclientusername\n if the logout is successful the message of SUCCESS will be sent to the user.

New registration: R_E_G\nclientusername\nclientpassword\n once the user is registered the message of SUCCESS will be sent to the user. If the registration fails then FAIL will be sent to the user.

Receive: U_P_D\nclientusername\n This is the message that is send to the server so that the client can receive an update.

Message sent to server from client: S_N_D\nclientusername\nR_C_V\nsusan566\n M_S_G\n Hi how are you?\n

Message sent to client from server: S_N_D\nclientusername\nM_S_G\n Hi how are you?\n

Chess game initialization between clients:

Message sent to the server for the chess request:
C_S\nclientusername\nfriendusername\nb\n for a game where the user plays black
C_S\nclientusername\nfriendusername\nw\n for a game where the user plays white

Message sent to the client for the chess request: C_S\nfriendusername\nb\n or C_S\nfriendusername\nw\n

Message sent to the server for the chess accept: C_A\nclientusername\nfriendusername\n

Message sent to the client for the chess accept: C_A\nfriendusername\n

Message sent to the server for the chess move: C_M\na3d6\nclientusername\nfriendusername\n :the first data received after the command for chess move is the starting row, starting column, then ending row and ending column followed by receiver username

Message sent to the client for the chess move: C_M\na3d6\nfriendusername\n :the first data received after the command for chess move is the starting row, starting column, then ending row and ending column followed by receiver username

Multiple communications can also be sent at the same time and the server or client will be able to receive the message and perform the appropriate actions.

5 Development Plan and Timeline

5.1 Partitioning of Tasks & Team Member Responsibilities

Table 1:

	Ameya Pandit	Matthew Dunn	Yunhe Shao	Richard Duong	Yuming Wang	Xingjian Qu
Software Development	Backup Logs Database Initialization	Data Structures, Server Side Communication	Main GUI	Client Side Communication	Account Management, Server Authentication	Main GUI
Team Roles	Presenter	Manager	Reflector	Reflector	Reflector	Recorder

Ameya is responsible for the development of the server-side backup log and database initialization infrastructure. This is a backup for the all connections and communication between all the clients connected to the server. The importance of this

functionality is that in the case the server is shut down, this log keeps track of all the connections and messaging that is lost; once the server is rebooted, this log will enter all the data that was stored, in effect creating a system with memory. Furthermore, Ameya is the presenter of the team; he is the liason for communication between the team he is part of and the instructor and other teams, ensures every member of the team has an input before seeking input from outside sources, and will manage crafting the presentation to showcase the game. Ameya also manages the team Github account, and offers assistance if teammates have issues with the service.

Matthew is responsible for developing the data structures and functions for creating lists and other functions related to data structures which are needed to execute this software application and functions that will take care of client messages, games being played, etc. Furthermore, Matthew will handle server side communication and request handling. The importance of this functionality is that clients and the server can communicate, which is the emphasis of the project. Matthew is also the manager of the team; he makes sure the team is prompt and focused, is responsible for time management, and values input from every single team member.

Yunhe and Xingjian are responsible for developing the GUI and the main function. The GUI is the user interface of this software application, and it is imperative that the user has an easy time interacting with the software. Developing the GUI is, in essence, making sure the user has a smooth time interacting with the other functionality this software provides. Yunhe and Xingjian are also updating the main function, ensuring that application runs smoothly and is void of any problems in execution. Yunhe is one of the group reflectors; he makes sure the team is unified, consistent, and thorough with their solutions, and observes and comments on general team dynamic so that the team can achieve as much as possible. Additionally, Xingjian will be the team recorder; he will log all the discussion comments when the team is sharing ideas and will take note of the important concepts every member of the team has learned in the development of this game.

Yuming is responsible for the client-side login, logout, registration. Furthermore, on the server side, Yuming is responsible for user authentication - are the login credentials provided valid. These two responsibilities are regarding the task of account management. The importance of this functionality are evident; it is important to help users utilize this service by creating accounts, and equally as important to assure that their account is protected through authentication. Additionally, Yuming is another one of the group reflectors. His responsibilities, as aforementioned, are to make sure the

team is unified, consistent, and thorough with their solutions, and to observe and comment on general team dynamic so that the team can be more efficient.

Richard is responsible for client-side communication. His responsibilities concern how the client receives data and how the client transmits this information to the server. This client also parses how the information provided should be parsed; what is the information provided useful for? The importance of this functionality is that is is paramount that the client and server can communicate, but it is also important that the client and server can understand what to do with the data provided. Richard and Yuming work with Yunhe as the reflectors of the team. By making sure the team is consistent with their solutions and monitoring the team dynamic, the trio helps keep every team member on the same page so that there is less confusion and no animosity, making the team efficient and the working environment healthy.

Although the team has elected to split tasks, it is imperative to note that every single team member will be working together and assisting one another when need be. For instance, although Ameya is responsible for building the presentation, members of the team will assist him with their respective parts. Every team member will assist each other with development and debugging their code. Evidently, although this section partitions the responsibilities of the project between team members, all team members will work together and help each other in order to develop the project and any documentation.

5.2 Timeline of Development, Testing, Releases

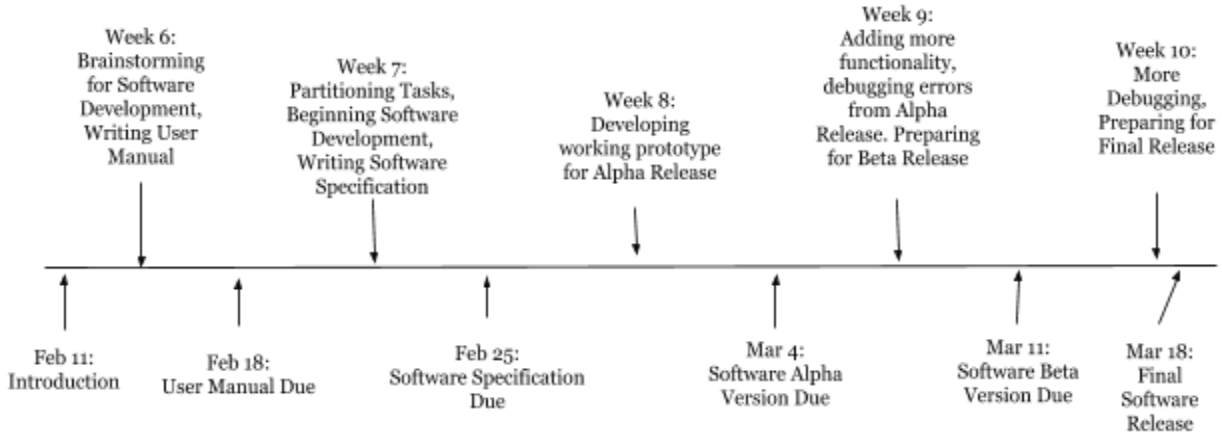


Figure 12: Timeline of our development

February 11: The team was introduced to the challenge after completing Project 1.
Week 6: The team began brainstorming for the development of the project. After brainstorming began the development of the User Manual.
February 18: The User Manual was due.
Week 7: The team partitioned the tasks needed to complete the application amongst team members and thus, software development could begin. The Software Specification was written.
February 25: The Software Specification was due.
Week 8: A functional version of the application was being developed. This prototype would be the Alpha Release.
March 4: The first release, the Alpha release, was due.
Week 9: Any inconsistencies and issues that were in the Alpha release were addressed. This week was dedicated to debugging any issues that were present. Objective was to have a fully functional product.
March 11: The second release, the Beta release, was due.
Week 10: Any issues arising from Beta release were to be handled. Fixing these issues yielded a completed product.
March 18: Final Software release was due.

Copyright

References

http://www.linuxhowtos.org/C_C++/socket.htm

Index