The Best Chess Game Ever V2.0

Version 2 (Software Specification)

Team:

Team Charlie

Members:

Allison Cornell, Adrien Guillaume, Vivian Lam, Devin Pham, Michael Yang

EECS 22L (Winter)

University of California, Irvine

Table of Contents:

Glossary:

Bishop: May only move diagonally as many squares as long as the pathway is not blocked.

Castle: A special move which involves moving the king and rook simultaneously. This will be the only time where two pieces can be moved in the same turn. There exists two types of Castle:
-*Castle Long*: *Queenside Castling*
-*Castle Short: Kingside Castling*
Both types of Castling consist of moving the King two squares either right or left depending on the type of Castling, and moving the rook on the square besides the King closest to the center. Keep in mind that this move can only be done if there are no other chess pieces in between the King and Rook, if neither pieces (King and Rook) have already moved, and if the King does not move out of a Check, into a Check, or over a Check. This special move allows the King to be in a safer position while giving the Rook a powerful position in the middle of the board.

Check: the act of attacking the opponent's King. When in Check, the player can call out "Check" to inform his opponent of the threat. In this case, the opponent must either move his King in a safe square, capture the attacking piece, or move another piece in between the King and the attacking piece.

Checkmate: the act of attacking the opponent's King such that it cannot escape by any means. The game is then over, and the attacking side wins.

Client: Service requester. In this application, the user is a client.

En Passant: from the French term "in passing". This move only occurs after a pawn moves two squares from its starting position, and passes an enemy pawn. The enemy pawn then has the opportunity to capture the passing pawn as if it had only moved forward one square. The right to capture "en passant" must be made immediately during the next move by the enemy's pawn, or else it will be lost in the following moves.

Fork: an attack where a piece threatens two or even three enemy pieces at the same time.

IP Address: An Internet Protocol address is a unique numerical label assigned to each device connected to a computer network that uses the Internet Protocol for communication.

King: Can only move 1 step in the surrounding squares.

Knight: May only move in the shape of a ‘L’. For example, 2 steps forward and 1 step right.

Pawn: For each pawn, the first move may either be 1-2 steps forward. After the first usage of that Pawn, they may only move 1 space forward at a time. The Pawn may also move diagonally forward 1 step to capture the opponent’s piece.

Port Number: A 16-bit unsigned integer ranging from 0 to 65535 to identify each process or application on a computer.

Queen: Can move 1 step in the surrounding squares. Can also move vertically, horizontally, and diagonally as many spaces as long as the pathway is not blocked.

Rook: May move vertically or horizontally as many squares in the horizontal or vertical direction as long as the pathway is not blocked.

Server: Service provider which stores the information of all users and responds to clients’ requests.

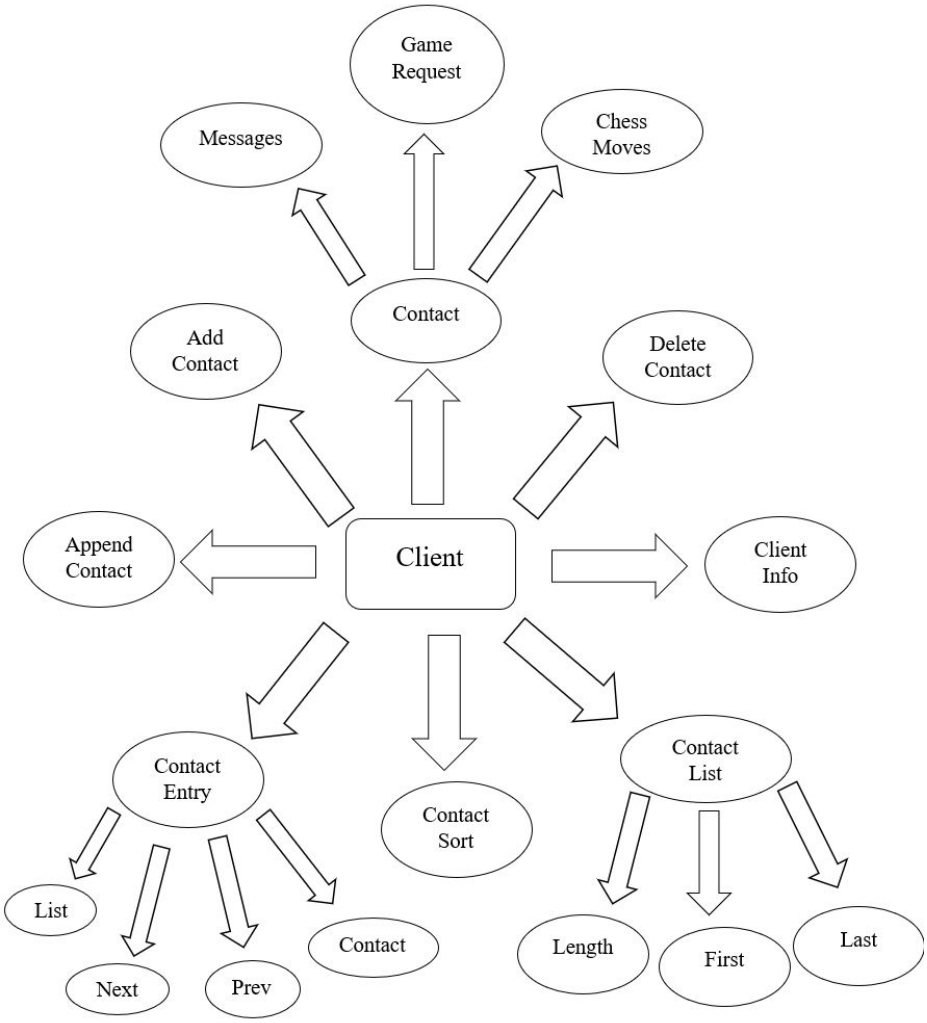
Support: The name of the server in the user interface.

1. Client Software Architecture Overview

1.1 Main Data Types and Structures

The main data types for the client are file descriptor variables that will be used to send a message through the correct socket as well as receive a message from the server. Data structures will be used for contacts. There will be three data structures, one for contact list, contact entry and contact. The client is still a work in progress so more data types will be added as we begin to further develop the client.

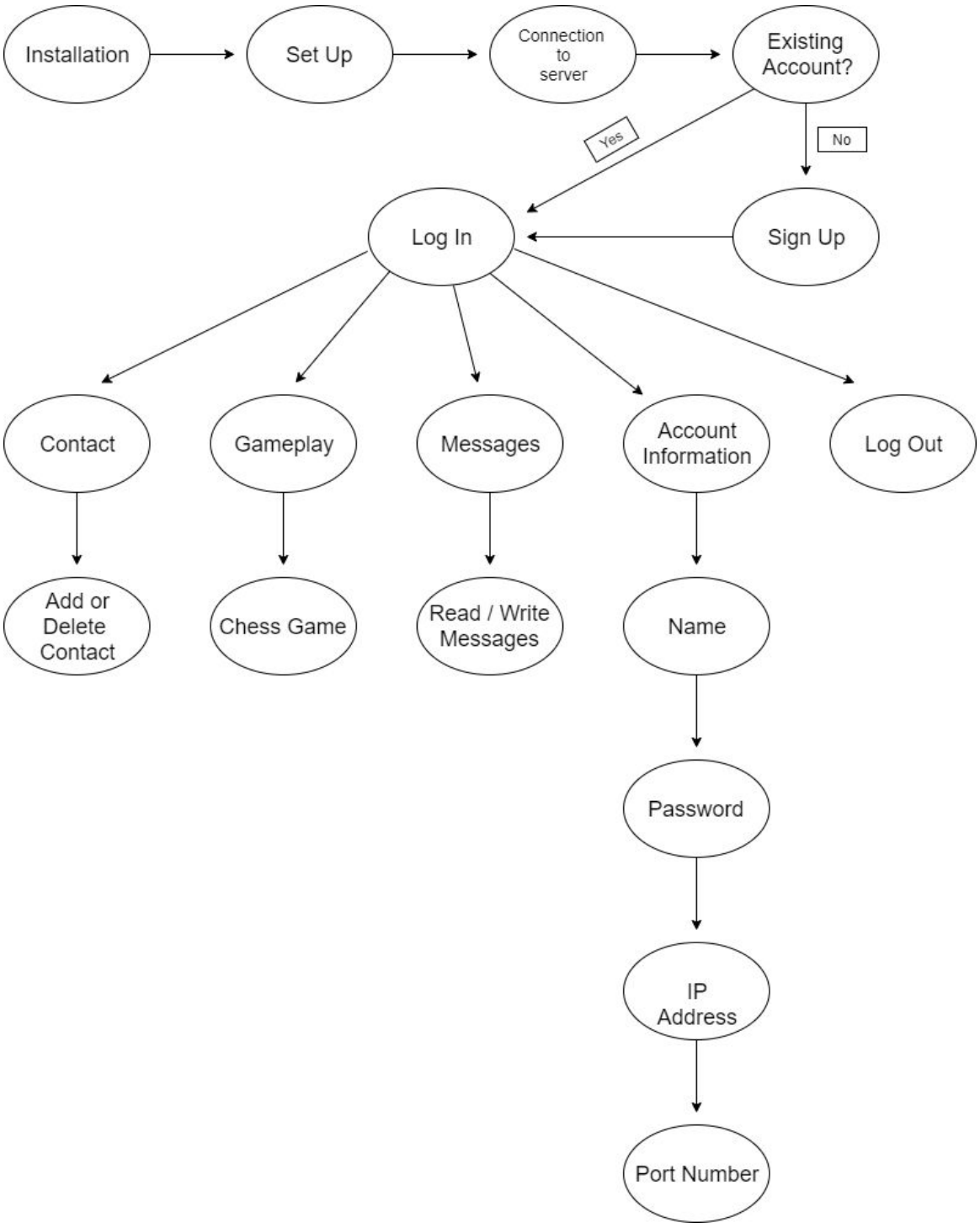
1.2 Major software components



1.3 Module interfaces

- Client.c
    - int client(int argc, char *argv[ ]);
    - struct Contact
    - struct ContactEntry
    - struct ContactList
    - CONTACT *addContact(char *Name)
    - void DeleteContact(CONTACT *c)
    - int ContactSort(CONTACT *c1, CONTACT *c2)
    - void AppendContact(LIST *l, CONTACT *c)
    - int recv_message(int socket, Message *msg)
    - void FatalError(const char *Program, const char *ErrorMsg)

1.4 Overall program control flow

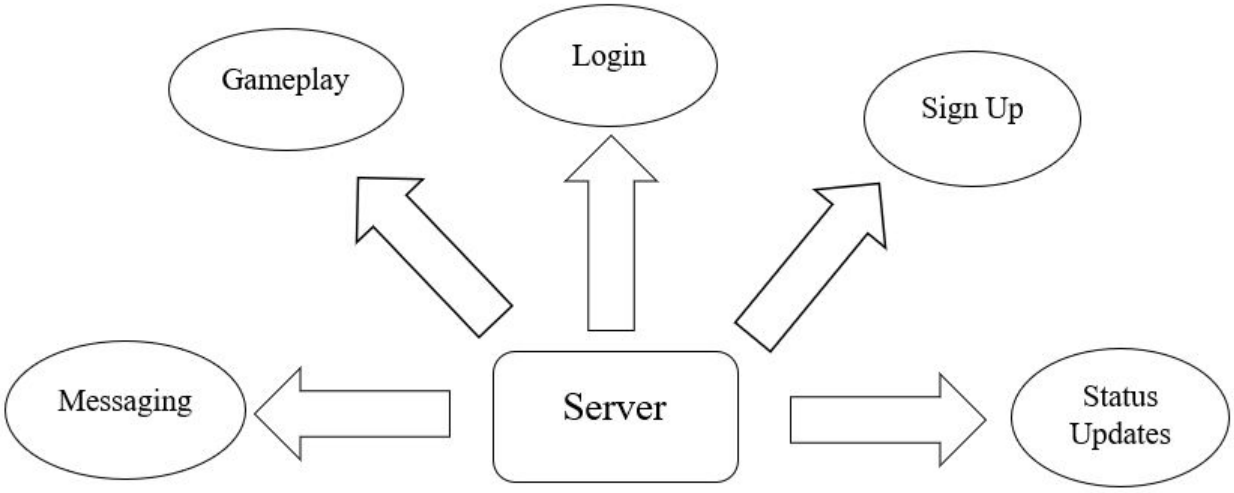


2. Server Software Architecture Overview

2.1 Main Data Types and Structures

The main data types for the server are arrays to store the username and password for each user and file descriptor variables in order to open a connection. This is still a work in progress so other data types will be added as we begin to further develop the server.

2.2 Major software components



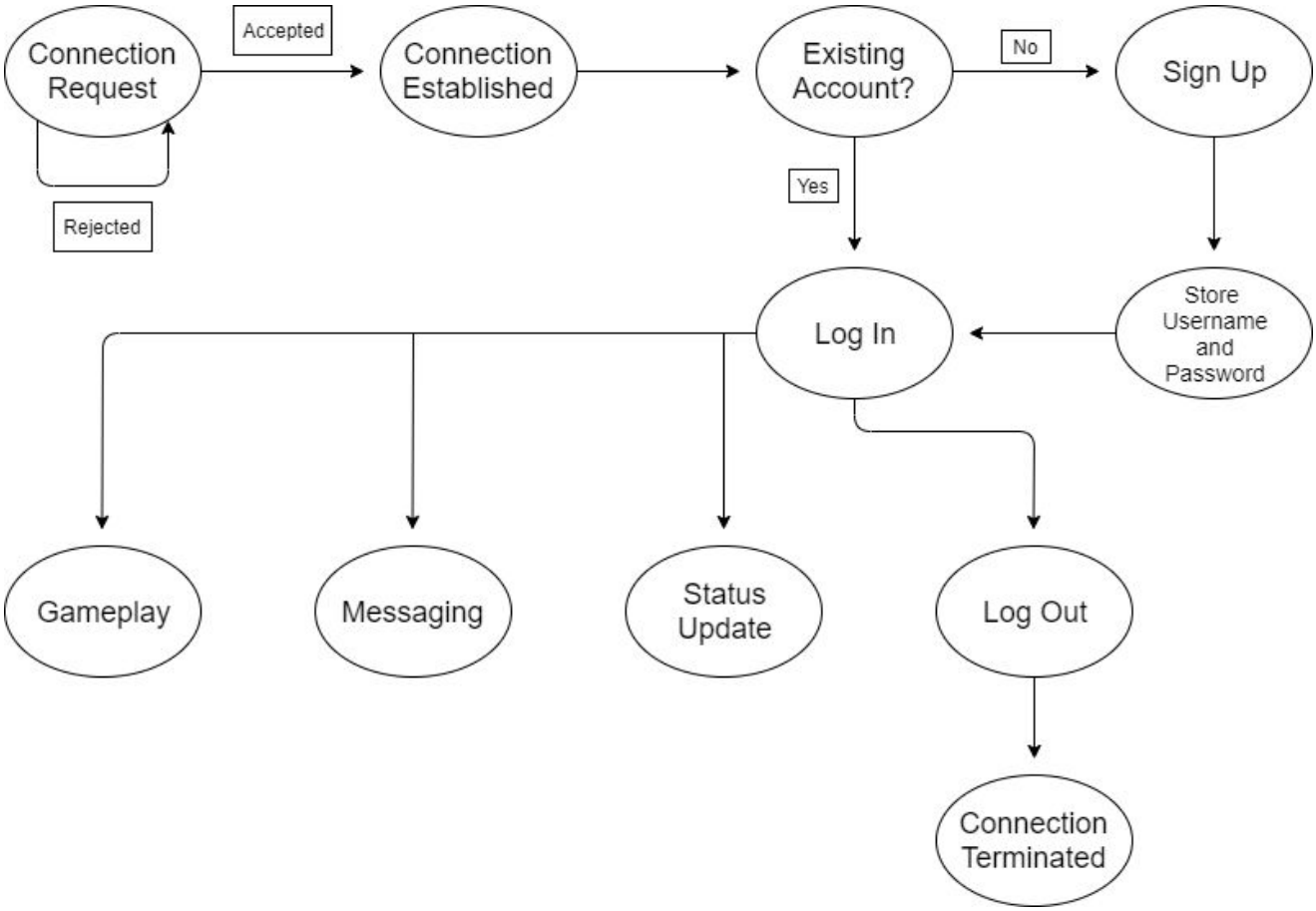
2.3 Module interfaces

- Server.c
    - int server(int argc, char *argv[ ]);

- void serverMessaging(char *Name)
- void Login(char *txt)
- void registration(char *txt)
- void Status(char *Name, char *txt)
- void Gameplay(char *txt)
- int recv_message(int socket, Message *msg)
- int verify_login(Message M, LIST *AList)
- int signup(Message M, LIST *AList)
- int send_msg(Message M, LIST *AList, char *TempBuf)
- void FatalError(const char *ErrorMsg)
- int MakeServerSocket(uint16_t PortNo)
- void ProcessRequest(int DataSocketFD, Message M, char *IP_Address, int port, LIST *AList)
- void ServerMainLoop(int ServSocketFD, ClientHandler HandleClient, TimeoutHandler HandleTimeout, int Timeout, Message M, LIST *AList)

- Account.c
  - ACCOUNT *CreateAccount(char *username, char *password, char *fname, char *lname)
  - void DeleteAccount(ACCOUNT *account)
  - ENTRY *CreateAccountEntry(ACCOUNT *account)
  - void DeleteAccountEntry(ENTRY *entry)
  - LIST *CreateAccountList(void)
  - void DeleteAccountList(LIST *list)
  - void AppendAccount(LIST *list, ACCOUNT *account)

2.4 Overall program control flow



3. Installation

3.1 System Requirements and Compatibility

OS:	Linux with Display Managers Compatible with X11 and SDK/GTK
Processor:	2 GHz Intel Dual-Core Processor
Memory:	900 MB Available
Graphics:	Video Card with at least 1GB VRAM
Storage:	1 GB Available

3.2 Setup and Configuration

The chess software will be compressed into a package with the extension ".tar.gz". To untar please type:

gtar -xvzf P2_Alpha_src.tar.gz

Please read the provided installation documentation before continuing with the installation process by typing:

vim INSTALL

3.3 Building, compilation, installation

Please type the following to compile the package:

make

Then type the following to run the client or server (must have two separate accounts/computers to run the programs in the same network):

make test

Please follow the prompts after running the previously stated command line

4. Documentation of packages, modules, interfaces

4.1 Detailed description of data structures

- Server Data Types:
  - A double link list to store the user information such as their username, password, first name, and last name. This way we can check to see if that user has an account already or if they are trying to make an account with a username that exists on the server already. In the data structure below will determine what kind of option the user wants to do such as sign up, login, send a message, or receive a message. (5 max users for now)
    - Code snippets:

```
typedef struct {
        int type;
        char message[256];
        char sender[21];
        char receiver[21];
        char password[9];
        char fname[15];
        char lname[15];
} Message;
```

  - File descriptor variables for sockets and port number in the server side. Server is waiting for the client to send a message to the correct socket number.
    - Code snippets:

```
/*receive the struct from the client*/
int recv_message(int socket, Message *msg) {
        size_t length = sizeof(Message);
        char *ptr = (char*) msg;
        while(length > 0) {
                int num_bytes = recv(socket, ptr, length, 0);
                if(num_bytes < 0)
```

```
                return -1;
            ptr += num_bytes;
            length -= num_bytes;
        }
            msg->type = ntohl(msg->type);
            return 0;
        }

int verify_login(Message M, LIST *AList){
        ENTRY *current;
        int verified;
        int Port_number;
        current = AList->First;

        if(AList->length == 0){
                printf("No registered user!\n");
                    }
        for(int i = 0; i < AList->length; i++)
        {
                printf("%d\nID: %s\nPass: %s\n", i+1,
                current->account->username, current->account->password);
                if (!strcmp(M.sender, current->account->username))
                {
                        printf("Username in database\n");
                        if (!strcmp(M.password,current->account->password))
                        {
                                verified = 1; //set flag to 1 since it is verified
                                strcpy(M.sender,
                                current->account->username);
                                Port_number = current-> account->
                                port_number;
                        }
                        //otherwise we have incorrect password
                        else
                        {
                                verified = 2; //incorrect password sets flag to 2
                                break;
                        }
         }
```

```
                Else
                {
                        current = current->Next;
                         if(i == AList->length - 1 && current == NULL)
                         {
                                //printf("NO Username in Database\n");
                                verified = 3;
                          }
                }
         }//end for loop
        return verified;
}

int signup(Message M, LIST *AList){
        ENTRY *e;
        ACCOUNT *a;
        a = CreateAccount(M.sender, M.password, M.fname, M.lname);
        e = CreateAccountEntry(a);
        AppendAccount(AList, a);
        printf("ID: %s\nPass: %s\n", e->account->username,
        e->account->password);
        return 0;
}
int send_msg(Message M, LIST *AList, char *TempBuf)
{
        ENTRY *e;
        e = AList -> First;
        assert(AList);
        while(e)
        {
                if(strcmp(e -> account -> username, M.receiver) == 0)
                 {
                        strcpy(e -> account -> chat, TempBuf);
                        return 0;
                 }
                e = e -> Next;
        }
        return -1;
}
```

```
void FatalError( const char *ErrorMsg)
{
        fputs(Program, stderr);
        fputs(": ", stderr);
        perror(ErrorMsg);
        fputs(Program, stderr);
        fputs(": Exiting!\n", stderr);
        exit(20);
} /* end of FatalError */

int MakeServerSocket(uint16_t PortNo)          /* create a socket on this server */
{
        int ServSocketFD; /*socket file descriptor*/
        struct sockaddr_in ServSocketName;
        ServSocketFD = socket(PF_INET, SOCK_STREAM, 0);
        if (ServSocketFD < 0){
                FatalError("service socket creation failed");
        }
        /* bind the socket to this server */
        ServSocketName.sin_family = AF_INET;
        ServSocketName.sin_port = htons(PortNo);
        ServSocketName.sin_addr.s_addr = htonl(INADDR_ANY);
        if (bind(ServSocketFD, (struct sockaddr*)&ServSocketName,
        sizeof(ServSocketName)) < 0){
                FatalError("binding the server to a socket failed");
        }
        /* start listening to this socket */
        if (listen(ServSocketFD, 5) < 0)    /* max 5 clients in backlog */
        {
                FatalError("listening on socket failed");
        }
        return ServSocketFD;
} /* end of MakeServerSocket */

void ProcessRequest(int DataSocketFD, Message M, char *IP_Address, int port,
LIST *AList)              /* process a time request by a client */
{
        int  I, n;
        int sign_up;
```

```
char SendBuf[256];  /* message buffer for sending a response */
char TempBuf[256];
int bye = 0;
int verified;
int send_status;
n = recv_message(DataSocketFD, &M);
if (n < 0){
        FatalError("reading from data socket failed");
}
if (M.type == BYE) {
        bye = 1;
        strncpy(SendBuf, "server bye", sizeof(SendBuf)-1);
        SendBuf[sizeof(SendBuf)-1] = 0;
}
/*client writing shutdown*/
else if (M.type == SHUTDOWN)
{
        Shutdown = 1;
        strncpy(SendBuf, "server shutdown", sizeof(SendBuf)-1);
        SendBuf[sizeof(SendBuf)-1] = 0;
}
/*sign up*/
else if(M.type == SIGNUP)
{
        sign_up = signup(M, AList);
        if(sign_up == 0){
                printf("Signup Successful!\n\n");
                strncpy(SendBuf, "signup successful", sizeof(SendBuf)-1);
                SendBuf[sizeof(SendBuf)-1] = 0;
        }
        else{
                printf("signup failed\n\n");
                strncpy(SendBuf, "signup failed", sizeof(SendBuf)-1);
                SendBuf[sizeof(SendBuf)-1] = 0;
        }
}
/*login*/
else if(M.type == LOGIN)
{
```

```
                verified = verify_login(M, AList);
                if(verified == 1){
                        printf("Successful login!\n\n");
                        strncpy(SendBuf, "Successful login!", sizeof(SendBuf)-1);
                        SendBuf[sizeof(SendBuf)-1] = 0;
                }
                else if(verified == 2){
                        printf("Incorrect password!\n\n");
                        strncpy(SendBuf, "Incorrect password!", sizeof(SendBuf)-1);
                        SendBuf[sizeof(SendBuf)-1] = 0;
                }
                else if(verified == 3){
                         printf("NO username in database!\n\n");
                         strncpy(SendBuf, "NO username in database!",
                        sizeof(SendBuf)-1);
                         SendBuf[sizeof(SendBuf)-1] = 0;
                }
}
 /*client sending message*/
else if (M.type == SEND)
{
        /*concatenate the sender and the message*/
        strcat(M.sender, ": ");
        strncpy(TempBuf, M.sender, sizeof(TempBuf)-1);
        TempBuf[sizeof(TempBuf)-1] = 0;
        strncat(TempBuf, M.message, sizeof(TempBuf)-1-strlen(TempBuf));
        printf("message is %s\n\n", TempBuf);
        send_status = send_msg(M, AList, TempBuf);
        if(send_status < 0)
        {
                printf("sending failed\n\n");
                strncpy(SendBuf, "sending failed", sizeof(SendBuf)-1);
                SendBuf[sizeof(SendBuf)-1] = 0;
        }
        Else
        {
                printf("message sent\n\n");
                strncpy(SendBuf, "message sent", sizeof(SendBuf)-1);
                SendBuf[sizeof(SendBuf)-1] = 0;
```

```
                }
        }
        /*request message*/
        else if (M.type == REQUEST)
        {
                /*loop thru the list and find the message*/
                ENTRY *e;
                e = AList -> First;
                assert(AList);
                while(e)
                {
                        if(strcmp(e -> account -> username, M.sender) == 0)
                        {
                                if(!strcmp(e -> account -> chat, ""))
                                {
                                        printf("There is no message\n\n");
                                        strcpy(SendBuf, e -> account -> chat);
                                }
                                else
                                {
                                        strcpy(SendBuf, e -> account -> chat);
                                        printf("Message found\n\n");
                                }
                        }
                         e = e -> Next;
                }
        }

        l = strlen(SendBuf);
        n = write(DataSocketFD, SendBuf, l);
        if (n < 0){
                FatalError("writing to data socket failed");
        }
} /* end of ProcessRequest */

void ServerMainLoop(int ServSocketFD,           /* server socket to wait on */
                                ClientHandler HandleClient,     /* client handler to call */
                                TimeoutHandler HandleTimeout,
```

```
        int Timeout, Message M, LIST *AList)                /*
timeout in micro seconds */
{
   int DataSocketFD;   /* socket for a new client */
   socklen_t ClientLen;
   struct sockaddr_in
      ClientAddress;  /* client address we connect with */
   fd_set ActiveFDs;   /* socket file descriptors to select from
*/
   fd_set ReadFDs;     /* socket file descriptors ready to read
from */
      //fd_set clientFDs;
   struct timeval TimeVal;
   int res, i;
      char IP_Address[16];
      int port;

   FD_ZERO(&ActiveFDs);              /* set of active sockets
*/
   FD_SET(ServSocketFD, &ActiveFDs);   /* server socket is
active */
   while(!Shutdown)
   {
            ReadFDs = ActiveFDs; /*because select is
destructive so we need a temp*/

            TimeVal.tv_sec  = Timeout / 1000000;    /*
seconds */
            TimeVal.tv_usec = Timeout % 1000000;    /*
microseconds */

            /* block until input arrives on active sockets or until
timeout */
            /*first is size, second is reading, third is writing,
fourth is errors,
            last is timeout*/
            res = select(FD_SETSIZE, &ReadFDs, NULL,
NULL, &TimeVal);
```

```
            if (res < 0)
            {
                  FatalError("wait for input or timeout (select)
failed");
            }
            if (res == 0)   /* timeout occurred */
            {
                  #ifdef DEBUG
            printf("%s: Handling timeout...\n", Program);
                  #endif
            HandleTimeout();
            }
            else           /* some FDs have data ready to read */
            {
                  for(i=0; i<FD_SETSIZE; i++) /*go through all
FD values*/
            {
                        if (FD_ISSET(i, &ReadFDs)) /*check if it
is set*/
                        {
                              if (i == ServSocketFD) /*new
connection we can get*/
                        {     /* connection request on server
socket */
                                    #ifdef DEBUG
                                    printf("%s: Accepting new
client %d...\n", Program, i);
                                    #endif
                                    ClientLen =
sizeof(ClientAddress);
                                    DataSocketFD =
accept(ServSocketFD, (struct sockaddr*)&ClientAddress,
&ClientLen);
                                    if (DataSocketFD < 0)
                                    {
                                          FatalError("data socket
creation (accept) failed");
                                    }
                                    #ifdef DEBUG
```

```
                                    printf("%s: Client %d
connected from %s:%hu.\n",
                                    Program, i,

inet_ntoa(ClientAddress.sin_addr),
ntohs(ClientAddress.sin_port));
                                    #endif
                                    /*add socket to set of sockets
we're watching which is ActiveFDs*/
                                    FD_SET(DataSocketFD,
&ActiveFDs);
                        }
                        else
                        {   /* active communication with a client
*/
                                    #ifdef DEBUG
                                    printf("%s: Dealing with client
%d...\n", Program, i);
                                    #endif

                                    strcpy(IP_Address,
inet_ntoa(ClientAddress.sin_addr));
                                    port =
ntohl(ClientAddress.sin_port);

                                    //handle the connection
                                    HandleClient(i, M,
IP_Address, port, AList);

                                  //#ifdef DEBUG
                                    printf("%s: Closing client %d
connection.\n", Program, i);
                                    //#endif
                                    close(i);
                                    //remove from set of
connections we are watching
                                    FD_CLR(i, &ActiveFDs);
                        }
                        }
```

```
            }
            }
    }
} /* end of ServerMainLoop */

/*** main function
*********************************************************/

int main(int argc, char *argv[])
{
    int ServSocketFD;   /* socket file descriptor for service */
    int PortNo;         /* port number */

        LIST *AList = NULL;
        //ENTRY *e, *current;
        //ACCOUNT *a;
        Message M;
        //if the list is empty make it createaccount list
        if(AList == NULL){
                AList = CreateAccountList();
        }

    Program = argv[0];  /* publish program name (for
diagnostics) */
        #ifdef DEBUG
    printf("%s: Starting...\n", Program);
        #endif
    if (argc < 2)
    {
            fprintf(stderr, "Usage: %s port\n", Program);
            exit(10);
    }
    PortNo = atoi(argv[1]);     /* get the port number */
    if (PortNo <= 2000)
    {
            fprintf(stderr, "%s: invalid port number %d, should
be >2000\n",
            Program, PortNo);
        exit(10);
```

```
    }
        #ifdef DEBUG
    printf("%s: Creating the server socket...\n", Program);
        #endif
    ServSocketFD = MakeServerSocket(PortNo);
    printf("%s: Waiting for messages %d...\n", Program,
PortNo);
    ServerMainLoop(ServSocketFD, ProcessRequest,
PrintCurrentTime, 250000, M, AList);
        DeleteAccountList(AList); //delete the account list after
the server is shut down
    printf("\n%s: Shutting down.\n", Program);
    close(ServSocketFD);
    return 0;
}
```

```
int sockfd, newsockfd, portno;

void serverMessaging(char *Name)
{
        /* Reads txt file for the registered users */
        /* Makes new txt file for server to write back to each
        user as needed */
}
void Login(char *txt)
{
        /* checks for user in txt */
        /* sends error messages if user does not exist */
```

```
        /* if everything is correct, server connects user to
        account */
}
void registration(char *txt)
{
        /* opens the registered txt file */
        /* obtains the information from the user */
        /* if everything they entered is valid then the txt file
        gets written */
}
void Status(char *Name, char *txt)
{
        /* default changes user as available */
        /* alters the txt file of the users status accordingly */
}
void Gameplay(char *txt)
{
        /* opens the txt file that keeps the log of the chess
        moves between the players */
        /* updates it accordingly with the user input */
        /* Updates the gameplay movement page */
}
```

- Client Data Types:
  - File descriptor variables for socket and port number. The client needs to connect to the correct server in order to communicate.
    - Code snippets:

```
int sockfd, portno;
typedef struct ContactEntry ENTRY;
typedef struct Contact CONTACT;
typedef struct ContactList LIST;
struct Contact
{
        int messages;
        int GameRequests;
        int ChessMoves;
};
```

```
struct ContactEntry
{
        LIST *List;
        ENTRY *Next;
        ENTRY *Prev;
        CONTACT *Contact;
};

struct ContactList
{
        int Length;
        ENTRY *First;
        ENTRY *Last
};
CONTACT *addContact(char *Name)
{
        CONTACT *c;
        c = malloc(sizeof(CONTACT));
        if (!c)
        {
                perror("Out of memory! Aborting… ");
                exit(10);
        }
        c->Name = Name;
        return c;
}
void DeleteContact(CONTACT *c)
{
        assert(c);
        free(c);
        /* Remove Contact from list */
}
int ContactSort(CONTACT *c1, CONTACT *c2)
{
        assert(c1);
        assert(c2);
        return(c1->Name - c2->Name);
}
void AppendContact(LIST *l, CONTACT *c)
```

```
{
        ENTRY *e = NULL;
        assert(l);
        assert(c);
        e = AddContact(c);
        if (l->Last)
        {
                e->List = l;
                e->Next = NULL;
                e->Prev = l-Last;
                l->Last ->Next = e;
                l->Last = e;
}

int ClientInfo(char * Name)
{
        /* get the text file saved with all the users info
        From the server to display when user selects
        account information */
}
```

- GUI Data Types: // work in progress
  - GtkWidget *main_window;
  - GtkWidget *chat_box;

4.2 Detailed description of functions and parameters

- Client.c
  - This module is responsible for connecting with the host and then prompting the user to enter a message. When the user sends a message it is written to the socket and is sent out. If there is a message sent to the client then the client will read that socket back. The client will read from the terminal to get the message that is sent back. This module will also keep track of the User's contacts and information.
  - Function Parameters:
    - int client(int argc, char *argv[ ]);

        - struct Contact
        - struct ContactEntry
        - struct ContactList
        - CONTACT *addContact(char *Name)
        - void DeleteContact(CONTACT *c)
        - int ContactSort(CONTACT *c1, CONTACT *c2)
        - void AppendContact(LIST *l, CONTACT *c)
        - int recv_message(int socket, Message *msg)
        - void FatalError(const char *Program, const char *ErrorMsg)

- GUI.c
    - This module serves to be the visual interface for the user to easily navigate the software with the addition of mouse clicks on top of using keyboard input. The GUI will be the front of the software that the user will interact with the most and is integrated with all of the functions to provide input feedback to the user for every action they make in the program.
    - Function:
        - void chat_window(GtkWidget …) //work in progress

```
GtkWidget *main_window;
GtkWidget *chat_box;
gtk_init (&argc, &argv);
main_window =
gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_window_set_title (GTK_WINDOW (main_window), "The
Best Chess Game Ever V2.0");
gtk_widget_show (main_window);
chat_box = gtk_window_new(GTK_WINDOW_MIDLEVEL);
gtk_window_set_title (GTK_WINDOW (chat_box), "The Best
Chess Game Ever V2.0");
gtk_widget_show (chat_box);
...
gtk_main ();
```

return 0;

- Server.c
    - This module is responsible for opening a connection with a socket so that the client can communicate with it. It will send a message back to the client when the server receives the message sent from the client. The server also reads the terminal in order to get the information the client sent.
    - Function Parameters:
        - int server(int argc, char *argv[ ]);
        - void serverMessaging(char *Name)
        - void Login(char *txt)
        - void registration(char *txt)
        - void Status(char *Name, char *txt)
        - void Gameplay(char *txt)
        - int recv_message(int socket, Message *msg)
        - int verify_login(Message M, LIST *AList)
        - int signup(Message M, LIST *AList)
        - int send_msg(Message M, LIST *AList, char *TempBuf)
        - void FatalError(const char *ErrorMsg)
        - int MakeServerSocket(uint16_t PortNo)
        - void ProcessRequest(int DataSocketFD, Message M, char *IP_Address, int port, LIST *AList)
        - void ServerMainLoop(int ServSocketFD, ClientHandler HandleClient, TimeoutHandler HandleTimeout, int Timeout, Message M, LIST *AList)
- Account.c
    - ACCOUNT *CreateAccount(char *username, char *password, char *fname, char *lname)
    - void DeleteAccount(ACCOUNT *account)

- ENTRY *CreateAccountEntry(ACCOUNT *account)
- void DeleteAccountEntry(ENTRY *entry)
- LIST *CreateAccountList(void)
- void DeleteAccountList(LIST *list)
- void AppendAccount(LIST *list, ACCOUNT *account)

4.3 Detailed description of the communication protocol

Note that the chat is still a work in-progress and the communication protocol will become more detailed as we move forward in the development process. For now, we only have the basic structure.

In order for the client and server to communicate with each other, the server will first have to open up a connection. The server will then wait for the client to send it a message through that open connection. When the message reaches the server, the server will send a message back and the client will read it from the socket.

As of now, we have implemented multi-client connections by using the select function in the server. This function allows the client to take turns to connect to the server on a first come first serve basis.

When the client starts,

If we are to apply this basic scenario to a chess game between two people and they are playing through the chat the protocol would be similar but slightly different. The server will still first open the connection and the client will send a message through that connection asking if the other user would like to play chess. When the server receives this request it will send that message to the other user and the second user will respond. In order to receive messages from two clients at once, the connection will be closed after the clients send each message. When the user makes a move on their chess board, the string they enter will be sent to the server then to the client. That string is parsed by the client and that move is made on their board. This way the moves update automatically. This same process continues until one of the players win.

5. Development plan and timeline

5.1 Partitioning of tasks

Allison Cornell:

- Work on the server side of chat program
- Main

Adrien Guillaume:

- Building the GUI in GTK
- Main

Vivian Lam:

- Work on the client side of chat program
- Main

Devin Pham:

- Work on client side of chat program
- Main

Michael Yang:

- Work on server side of chat program
- Main

5.2 Team member responsibilities

Devin Pham: Manager

Makes sure the team is up to par with their duties, makes sure the team follows through with their deadlines, and checks if each team member is doing their part.

Allison Cornell: Recorder

Keeps a record of everything said or made amongst the team. Records each team members duties for each assignment.

Vivian Lam: Presenter

Addresses the issues the team members are facing and then addresses it with the class.

Adrien Guillaume & Michael Yang: Reflector

Reevaluates the team's efforts and discusses what options are the best.

Back Matter:

Copyright

References

U.S. Chess Federation. “Let's Play Chess .” *US Chess Federation:* www.uschess.org/archive/beginners/.

Dang, Quoc-Viet. “Project 1.” EECS 22L. University of California - Irvine, California. 19 Apr, 2020.

Index:

Q

R

S