THOT Patrol

Bianca Corine Doronila
Cesar Alvarenga
Joshua Miller
Michael Reza Honar
Michael Barbosa
Tun Myat

In affiliation with: The Henry Samueli School of Engineering at The University of California Irvine

THOT Patrol Software
Version 1.0

Released 18th March 2018

Table of Contents

Glossary

application	-	another word for the program. *Remarq* is an application
client	-	a label for the application/executable the user installs on their PC
create account	-	a button where a user can create an affiliation with our service and chat with other users
global	-	specified for chat rooms, the global room is a group where all users are connected to
GUI	-	graphical User Interface, a window that the user interacts with app
IP	-	Internet Protocol: a set of rules governing the format of data sent over the Internet or other network.
IP address	-	a numerical label for every device connected to the internet
Instant Message	-	a text message sent over an internet protocol
master	-	part of the master-slave configuration, controls slave devices
non-blocking	-	a statement that does not stop the program to wait for a value, a 'blocking' statement will hold the program until the value is known (i.e. the message is received) whereas non-blocking will continue past even if the value is not known, and will stop to store the value if a value is waiting*
Protocol	-	a set of rules governing the exchange or transmission of data between devices.
provider service	-	this will be the server, in our case one of the UCI servers.
server	-	the computer which handles communication and storing user info
slave	-	part of the master-slave configuration, listens to master device
socket	-	one side of an interprocess communication channel.
status	-	options: Online, Idle, Offline. Tells users if a user is on or offline

Structure	**-**	a data type that contains variables as its members, which in a sense acts as a list.
FriendsList	**-**	will be of type struct and contain the users friends in it acting as a double linked list.
user	**-**	an account holder with credentials stored in our server

*a helpful analogy is with packages in the mail, a package that requires a signature is like a blocking statement, in that you must wait around in order to receive it, whereas a non-blocking statement is like a package that does not require a signature, it will be left at your door for you to collect when you check, and if it’s not there when you check, you move on.

FINAL SOFTWARE SPEC RELEASE NOTES& Basic intro/Readme

Please read the README/INSTALL in the parent directory for FINAL README

This is the software specification is written for the FINAL release of Remarq. Everything in this revision of the Software Specification is TENTATIVE and will change according to the patches coming up. The structures will be different: the linked lists will be different. Most functions will be revised and the parameters defined below might change. The control flow will change as we make our code more efficient. Our Client-Server protocol may change to make communications simpler and more secure.

This Software Specification will update with the alpha, beta and final submission.

In the alpha release, we have two apps:GUI and Test. GUI has two applets, ./Remarq and ./ServerMC. Test also has two applets, ./ServerTest and ./ClientTest. They both contribute to our beta.

1. The Client/Server Test works by connecting TWO clients to the server then starting a chat by typing in the clients. If only one client is connected, nothing happens. There is no “prompt” to tell the user when another client is connected, nor is there a prompt to tell the user when to start typing.
    a. We included this to prove we can private message two clients.
    b. Features: Non-blocking server, client-2-client communication
2. The second applet is the GUI and server structure, consisting of ./Remarq and ./ServerMC.
    a. Run both as explained in part 4 - installation
    b. The GUI and Server Structure are still work under progress
    c. The Login/Register is more of a dummy button. We have a working login/register implemented in our alpha release. Full GUI integration comes with the final release.

1 Client Software Architecture Overview

1.1 Main Data Types and Structures

Alpha release does not include data structures such as linked lists and typedef structs!

Alpha release also uses an ASCII GUI - GUI saved for beta.

- Client Application
    - GUI
        - Window 1:
            - “Log In” button
            - “Register” button
            - “Exit” button
        - Window 2:
            - Registration
                - “User ID” label and text entry
                - “Password” label and text entry
                - “Confirm Password” label and text entry
                - “FIRST” name label and text entry
                - “LAST” name label and text entry
                - “Date of Birth” label and text entry
                - “Create” button
                - “Exit” button
            - Log In
                - “User ID” label and text entry
                - “Password” label and text entry
                - “Exit” button
                - “Log In” button
                    - Goes to Window 3 if id and password match
                    - Dialog Box if id and password does not match
        - Window 3:
            - Main Menu
            - Online, Idle, and Offline labels
                - Scrollview box on each to show list
                - “Edit” button
                - Status button
                    - “Available”, “Idle”, “Invisible” radio buttons
                - “+” button

            - “-” button
            - Chat window
    - Structures.h (elaborate definition in section 5)
        - Data Types and Structures
            - Struct User
            - Struct Personal
            - Struct FriendList
            - Struct FriendEntry
        - Functions
            - void CreateFriendList(*user client);
                - Generates a FriendsList structure for a user
                - Formatted input from <user>.txt
                    - <user>.txt is a input file defined below
            - void AppendEntry(*user client, *user friend);
                - Append an entry of friend to add to user’s FriendsList. Handles an empty list as well.
            - void DeleteEntry(*user, *user friend);
                - Delete an entry of friend from user’s FriendsList

1.2 Major Software Components

- There are two major components to our software, a **client** that handles user interaction, such as opening chat windows and managing a user account. A **server** handles the backend functions, including providing a plug for a secure “socket” connection, sending and receiving packets to the correct IP address according to the client, and store chat logs and user lists. A Client and Server operate on a *master-slave* system, where the Server (when running) is always on, ready for a client to connect and tell the server what to do. The Client instructs the server where to direct the packets it is sending via TCP/IP.
    - The Client
        - Also known as the “User Application”, this application will launch a GUI so the user can interact with our chat tool *Remarq*.
        - Basic functions include: User Registration with personal information encrypted over the network, Login screen that updates a user’s status, a friends list screen that displays online and offline friends, and a chat window to IM a friend.
        - The Client is regarded as the **master**

- Provided **GUI.**
    - Displays a user interface to interact with

1.2.1 Diagram of Module Hierarchy

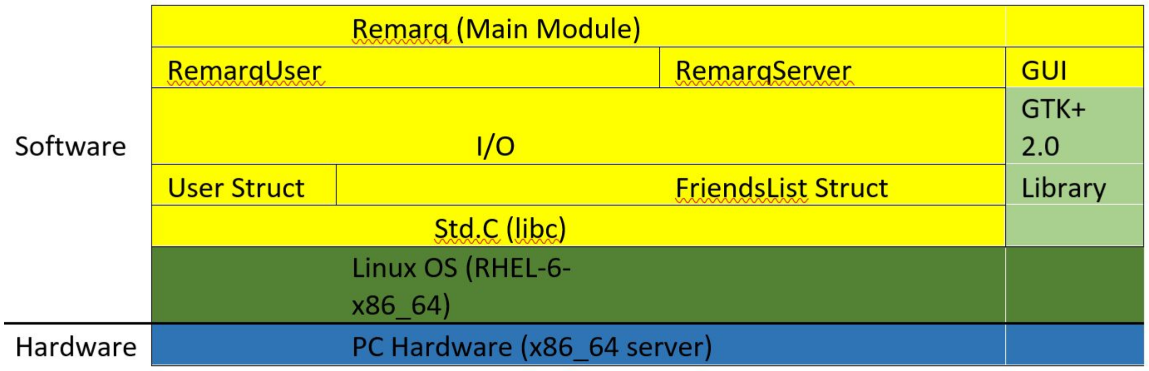
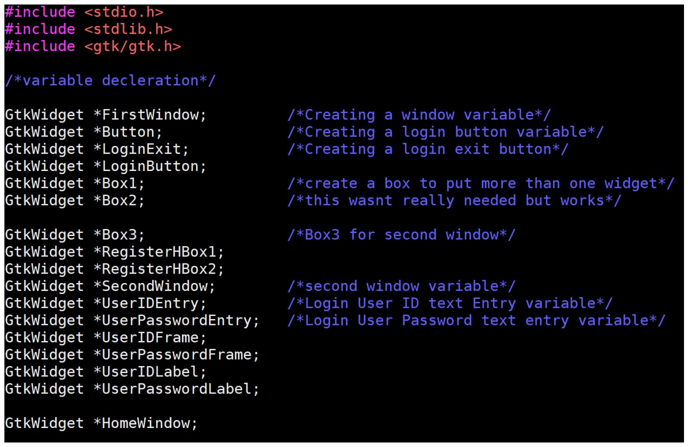


Figure 1.1 shows the diagram of module hierarchy

1.3 Module Interfaces

Our GUI will be made using the GTK+ 2.0 Library. In order to use the library, simply add the GTK library into the header file and run it using the specific commands instructed under the GTK tutorials (link to GTK tutorial provided in References).



```
#include <stdio.h>
#include <stdlib.h>
#include <gtk/gtk.h>

/*variable decleration*/

GtkWidget *FirstWindow;         /*Creating a window variable*/
GtkWidget *Button;              /*Creating a login button variable*/
GtkWidget *LoginExit;           /*Creating a login exit button*/
GtkWidget *LoginButton;
GtkWidget *Box1;                /*create a box to put more than one widget*/
GtkWidget *Box2;                /*this wasnt really needed but works*/

GtkWidget *Box3;                /*Box3 for second window*/
GtkWidget *RegisterHBox1;
GtkWidget *RegisterHBox2;
GtkWidget *SecondWindow;        /*second window variable*/
GtkWidget *UserIDEntry;         /*Login User ID text Entry variable*/
GtkWidget *UserPasswordEntry;   /*Login User Password text entry variable*/
GtkWidget *UserIDFrame;
GtkWidget *UserPasswordFrame;
GtkWidget *UserIDLabel;
GtkWidget *UserPasswordLabel;

GtkWidget *HomeWindow;
```

Figure 1.2 shows GTK header file

1. This will be the first window to show up when the user runs the program. The User will either be able to login to their existing account or create a new account if they do not have one.

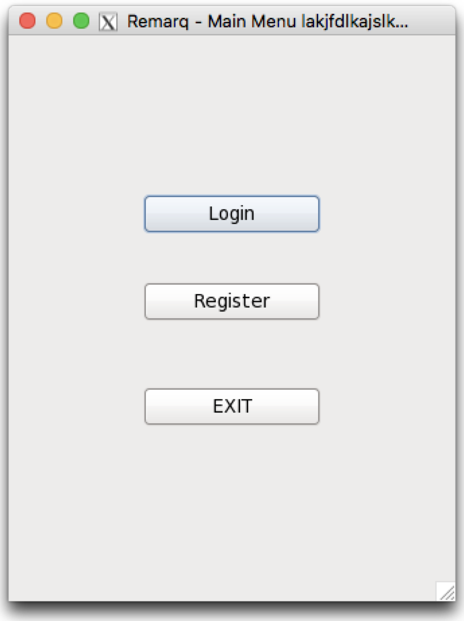


Figure 1.3 shows GTK login and register menu window

2. The Login Page will contain two text entries where the user can enter their username and password. It will also contain the Login in button that will guide the user to the next window, and an exit button for easy transition out of the application.

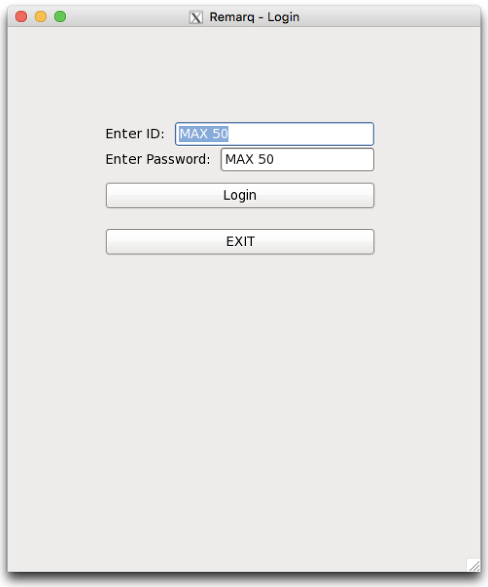


Figure 1.4 shows GTK user login window

3. The Register window will consist of a couple text entries that will provide us the necessary information to create an account.

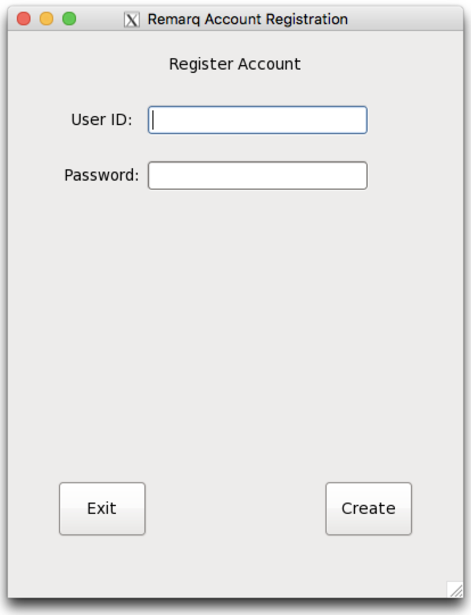
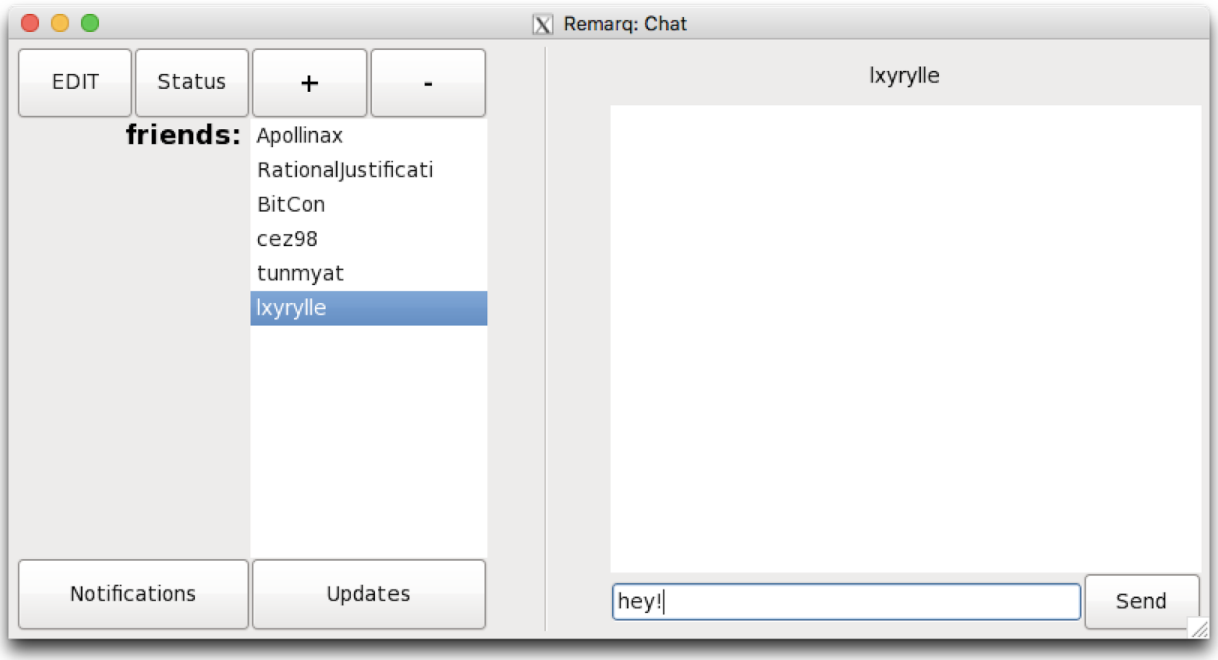
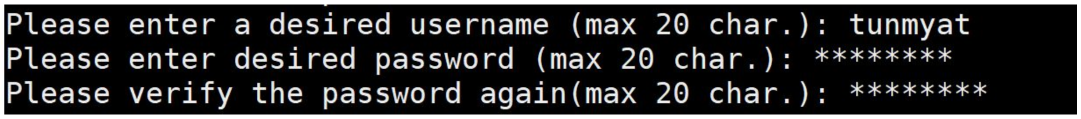


Figure 1.5 shows GTK user registration form window

4. The Chat Window has the edit button which will modify the txt file associated with the registered user, status which will edit your current status with the server, add friends that adds to the text file and shows on your friends' list view, delete to remove. Notifications and updates are **pending implementation and may be removed.** Highlighted friend is the who the user is chatting with on the right side.



5. An example of how we will replace a typed password with asterix that will be integrated into our alpha release. The applet uses a getchar( ) function and replaces the output to console as a ‘*’ (asterisk).



```
Please enter a desired username (max 20 char.): tunmyat
Please enter desired password (max 20 char.): ********
Please verify the password again(max 20 char.): ********
```

Figure 1.6 shows user input password is replaced with ‘*’ (asterisk)

1.3.1 API of Major Module Functions (client)

API Module Interface of current files. Since this is tentative we will add more files as we progress.

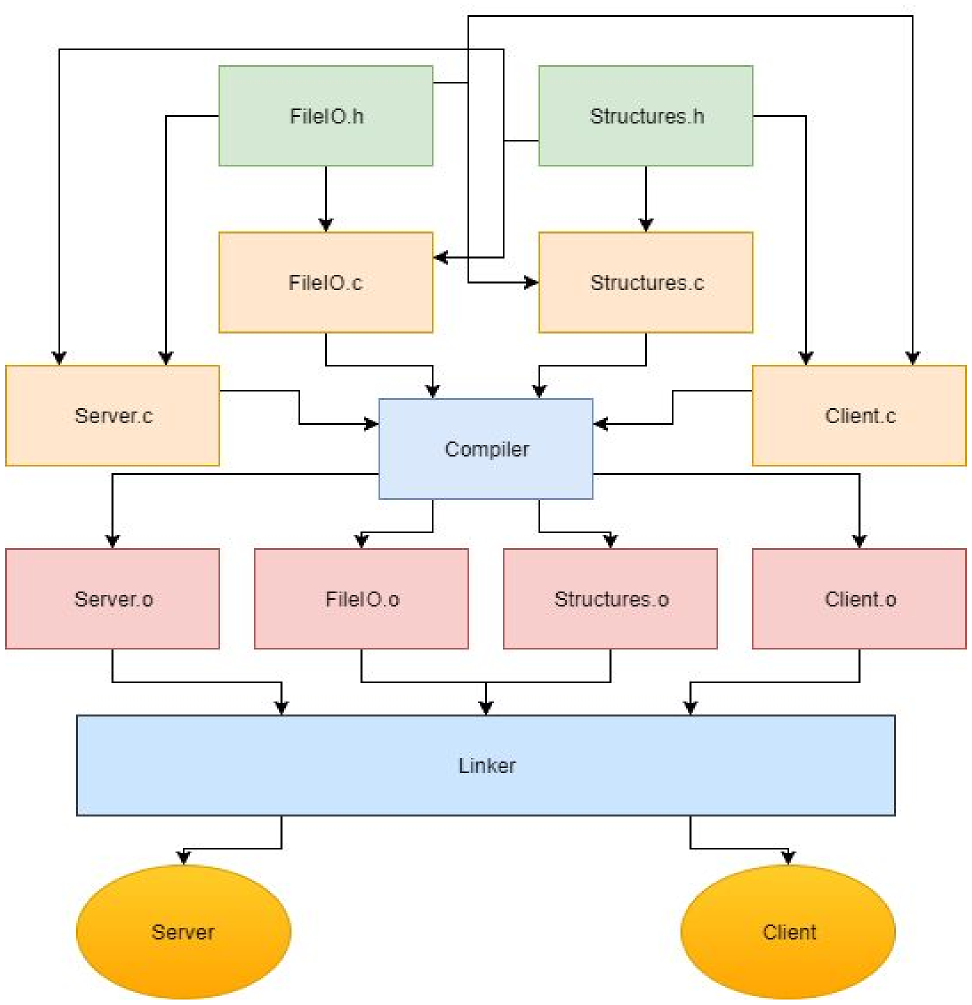
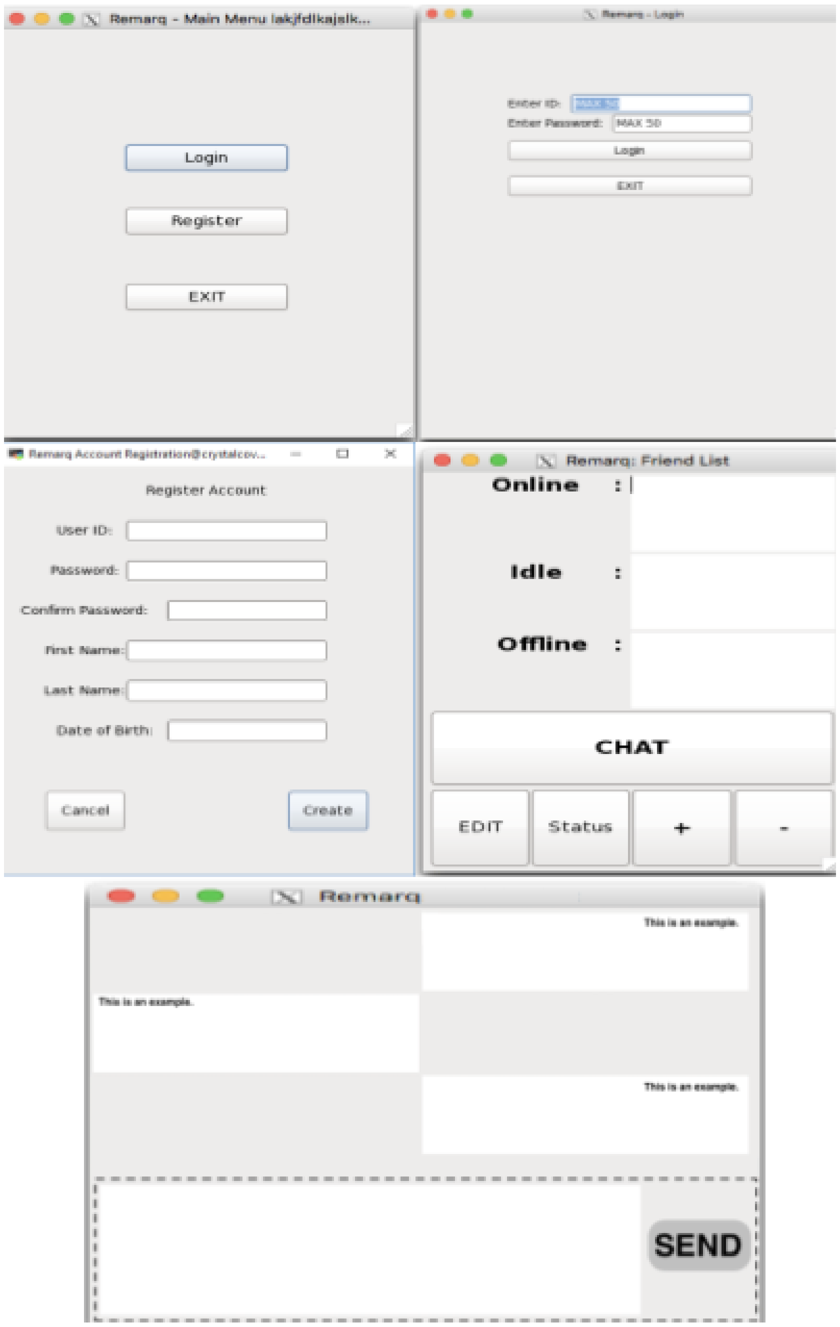


Figure 1.7 shows the API module interface diagram

1.4 Overall Control Flow

- Client Control Flow



In reference to the GUI on the left, which outlines the main control flow of the Client

1. Remarq
   a. Opt 1: Login
      i. Goto :Log In
   b. Opt 2: Register
      i. Goto :Register1
2. Remarq: Register1 (filling the user struct)
   a. Input: User ID
   b. Input: User PW
   c. Input: User PW confirmation
   d. Input: User First name
   e. Input: User Last name
   f. Input: Date of Birth in MM-DD-YYYY
      i. Button Cancel, return to Remarq
      ii. Button Create, goto :Main
3. Remarq: Login
   a. Input: User ID
   b. Input: User PW
      i. Button Cancel, return to Remarq
      ii. Button Login, goto :Main
4. Remarq: Main
   a. List: Online users, can select multiple users
      i. Double click, goto: chat
   b. List: Idle users
   c. List: Offline users
   d. Button Edit
      i. Modify User data (Name, etc)
   e. Button Status
      i. Change to Online, Idle, etc
   f. Button '+' add friend
      i. Goto: Add Friend
   g. Button '-' remove friend
      i. Remove friend
5. Remarq: Chat
   a. Chat window that displays texts.
   b. Has a input window
      i. Button send & erase

Figure 1.8 shows overall client control flow

2 Server Software Architecture Overview

2.1 Main Data Types and Structures

- The main data types and structures of the server are shared between the server and client. Both applications use a “user” struct and a FriendsList to control who can send messages to which user. Since our software is in pre-alpha release, most of the structures defined above and in section 5 will not be used until our alpha/beta release. For now, our server handles multiple connections on a global chat room. Progress will be made to fully implement all structures

2.2 Major Software Components

- The Server
  - Also known as the “Provider Service”, the server handles all backend operations that the user does not see.
  - Basic functions include: Storing a list of all registered users, matching login information with the user list, receiving packets of data (our data is the IMs) and sending the packets to the correct client over IP (using “sockets”), managing the user accounts, and maintain user’s status, IP addresses, port numbers.
  - Also saves a text document of all the chat logs
  - The Server is regarded as the **slave**, since it is always “listening” to the client.
- Server main file
  - Main functions used: read() and write()
  - Handles connections. A multi threaded process will be employed where the connection to each user has its own thread of process happening.
    - As of alpha version runs without multithreading, however may be implemented to handle more clients or multiple conversations.
      - Currently runs utilizing non-blocking read statements in an infinite while loop that only exists when one of the two users disconnects
  - Server Protocol
    - All data sent to the server per user is handled in a protocol. The Client sends at least 2 messages to the server and the server handles the messages correctly. For example, if the first message sent to the server is “LOGIN” followed by “<username>” and “<password>”, then the server identities the client is in the “login state” trying to gain access to the chat function. Similarly, “MESSAGE” followed by “<text>” is protocol for a message to be sent to the *global* room.

        - Current protocol definitions:
            - “REGISTER” <user> <password> - register a new user
            - “LOGIN” <user> <password> - login to server
            - “MESSAGE” <text> - send a message to the global room
        - Future protocol definitions
            - “MESSAGE” <writer> <reader> <text> - private message
            - “MESSAGE” <writer> <reader1> <reader2> <text> - pm in group chat
        - Return to client
            - Each protocol execution sends a confirmation message back to the client. For example, with a successful registration protocol sent to the server, a response will be sent back to the client via server with a message “Registration Successful!”. A message protocol sent to the server for a chat room will return a confirmation to the client and also display the sent message by the user. A message received by another user will notify the client and display the message in a similar fashion.

- The Client
    - Will be the front end of the application, the software utilized by each user to communicate with the server and by extension other users
    - Houses the gui with which the user interfaces on the front end of the client
    - Handles taking user input and relaying it to the server
    - Handles taking replies from the server and relaying them to the user
    - Runs multithreaded (?)
        - Spawns a child task to handle listening to the server and the parent of that tasks handles user input as well as relaying said input to the server
        - Two pipes are opened between the parent and child processes through which they can communicate
            - Currently used to pass a stop code between the two processes, as the command to stop the chat could come from the user or, in the current implementation, from the server
            - In later versions these pipes may be used for passing strings/messages between the two processes once the GUI is implemented
        - Multithreading may be removed in later version in favor of non-blocking read statements, should data exchange across threads become too cumbersome or infeasible

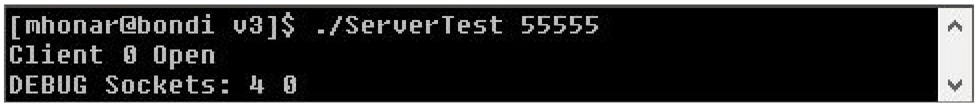
- FileIO.h
    - Handles reading and writing into local .txt files
        - Local .txt files will compose the application’s user base and text logs
        - Files
            - <user>.txt
                - Every user registered has a <user>.txt associated with them. It stores their login credentials (encrypted) and also stores a list of friends associated with the user.
                - In the future the user.txt will be used as a formatted input file to generate a user struct and userFriendsList upon login.
            - RegisteredUsers.txt
                - A file containing the login and passwords of all users on the server. For now this is a rudimentary implementation for the pre-alpha release to handle basic login/registration, and will be eliminated by beta phase. All data stored here will be encrypted.
            - <user><user>Log.txt
                - A simple text log of every message exchanged between two or more users. Messages will be time stamped with the sender’s userID.
        - Functions
            - char* ServerRegister(char* cUser, char* cPass, char* RegisterProtocol)
                - Once the server loads in the client login credentials using the read() function, ServerRegister() will pass in the user and password and create a new account by generating a new line in <RegisteredUsers.txt> and for future implementation create a user file called user.txt. Remember each user has a unique file association named after the unique userID.
                - Later implementation will use a user struct defined earlier.
                - Returns a confirmation flag
            - char* ServerLogin(char* cUser, char* cPass, char* RegisterProtocol)
                - Compares a sent username and password to the RegisteredUsers.txt. Later implementation will use a user struct. If a user and password matches the file, then the client has access to chat functions

                - Later implementations in beta release will implement the user struct to create an instance of a user for the client and load the friendsList as well.

- Structure.c & Structure.h
    - Defines two new types: A struct user, and a struct for a userFriendsList of types user.
        - The entries of the structures are defined above in section 1 and again in section 5
        - The structures are used to keep track of the current user who is registered to the client, and allow the server to send the appropriate message to the correct user/client.
        - Structures will also be used to correctly log messages and update user.txt when a new friend gets added/removed to a list.
            - At the login of a client, the server will send the client the friends list of the user with the use of write() via the file user.txt containing dada about the friends of a user. Now the client handles the friends list.
            - Upon exit of the client, if the friendsList gets modified, then the user.txt file will be updated so the new list can be stored back on the server. Exchange happens using read() and write() functions.
- These are the main Data Types and structures that will be used for the server.

2.3 Module Interfaces

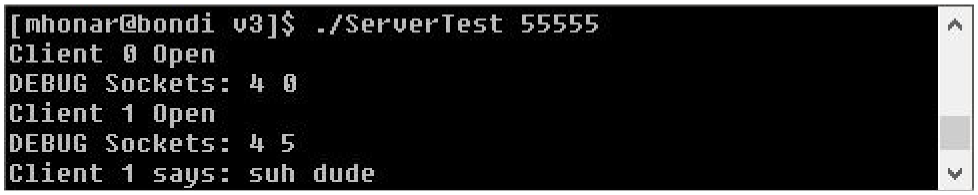
- The server will not have a GUI, however be a console based and report what is going on. Since we are in pre-alpha release, we do not have the server do much. It our current implementation of the server allows up to 10 users to connect to a global chat room, and show what users are connected to what socket. Here is a sample output of a server console. No input is supported at this moment.



```
[mhonar@bondi v3]$ ./ServerTest 55555
Client 0 Open
DEBUG Sockets: 4 0
```

Figure 2.1 shows 1 user connect to a global chat room

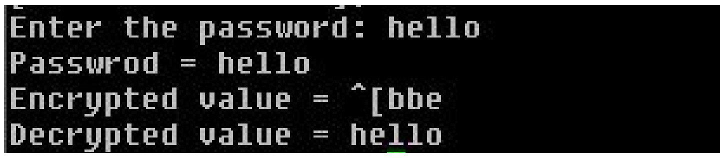
- Here in an example with another user logged in



```
[mhonar@bondi v3]$ ./ServerTest 55555
Client 0 Open
DEBUG Sockets: 4 0
Client 1 Open
DEBUG Sockets: 4 5
Client 1 says: suh dude
```

Figure 2.2 shows another user login to chat room

- Password Encryption/Decryption



```
Enter the password: hello
Passwrod = hello
Encrypted value = ^[bbe
Decrypted value = hello
```

Figure 2.3 shows password encryption and decryption

2.3.1 API of Major Module Functions (server)

API Module Interface of server:

Since our Client and Server application is still in development, it is small in file number and therefore we use the same chart as the Client, since it includes the current “full” interface

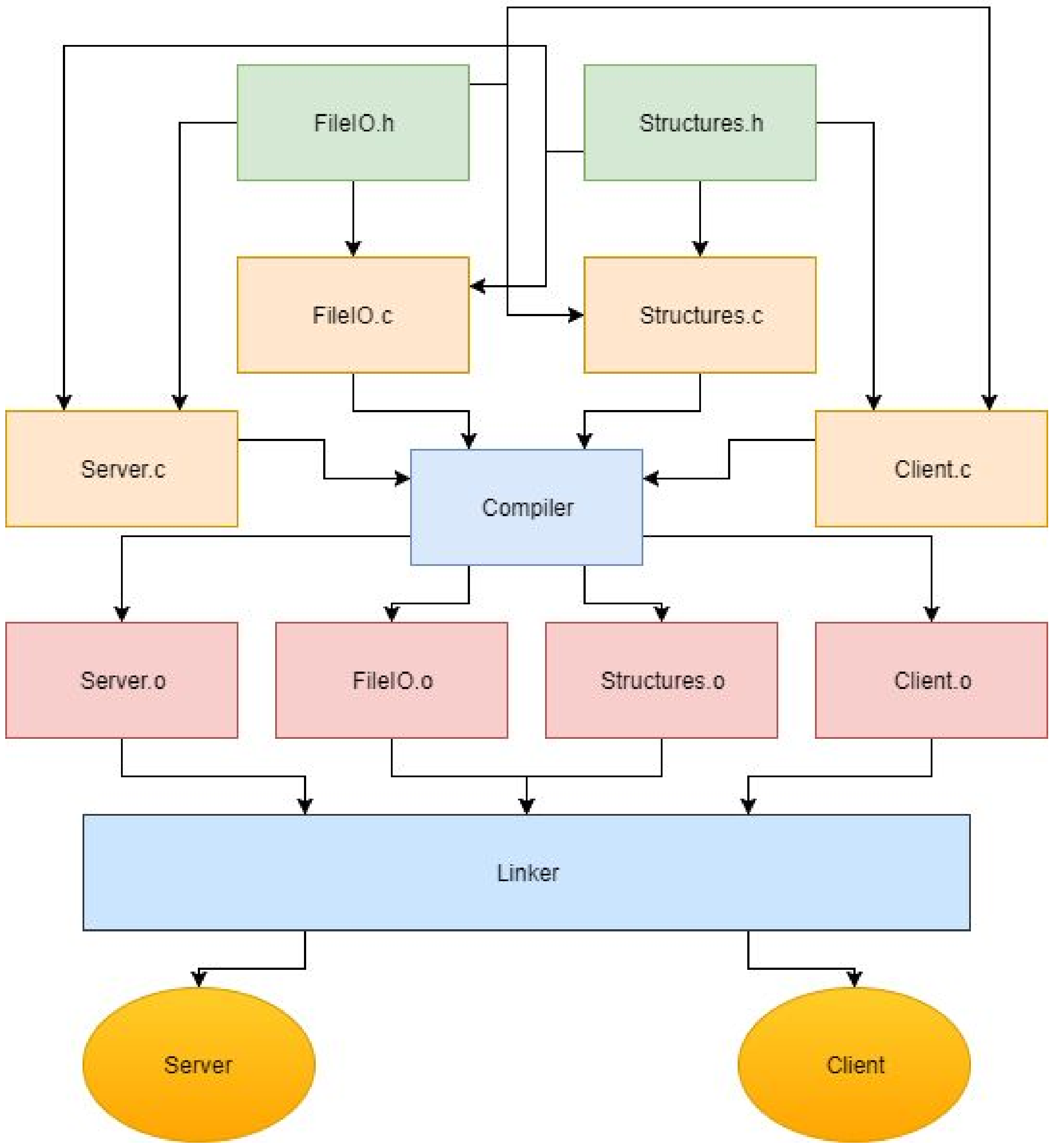


Figure 2.4 shows API of Major Module Functions (server)

2.4 Overall Control Flow

Master Control Flow

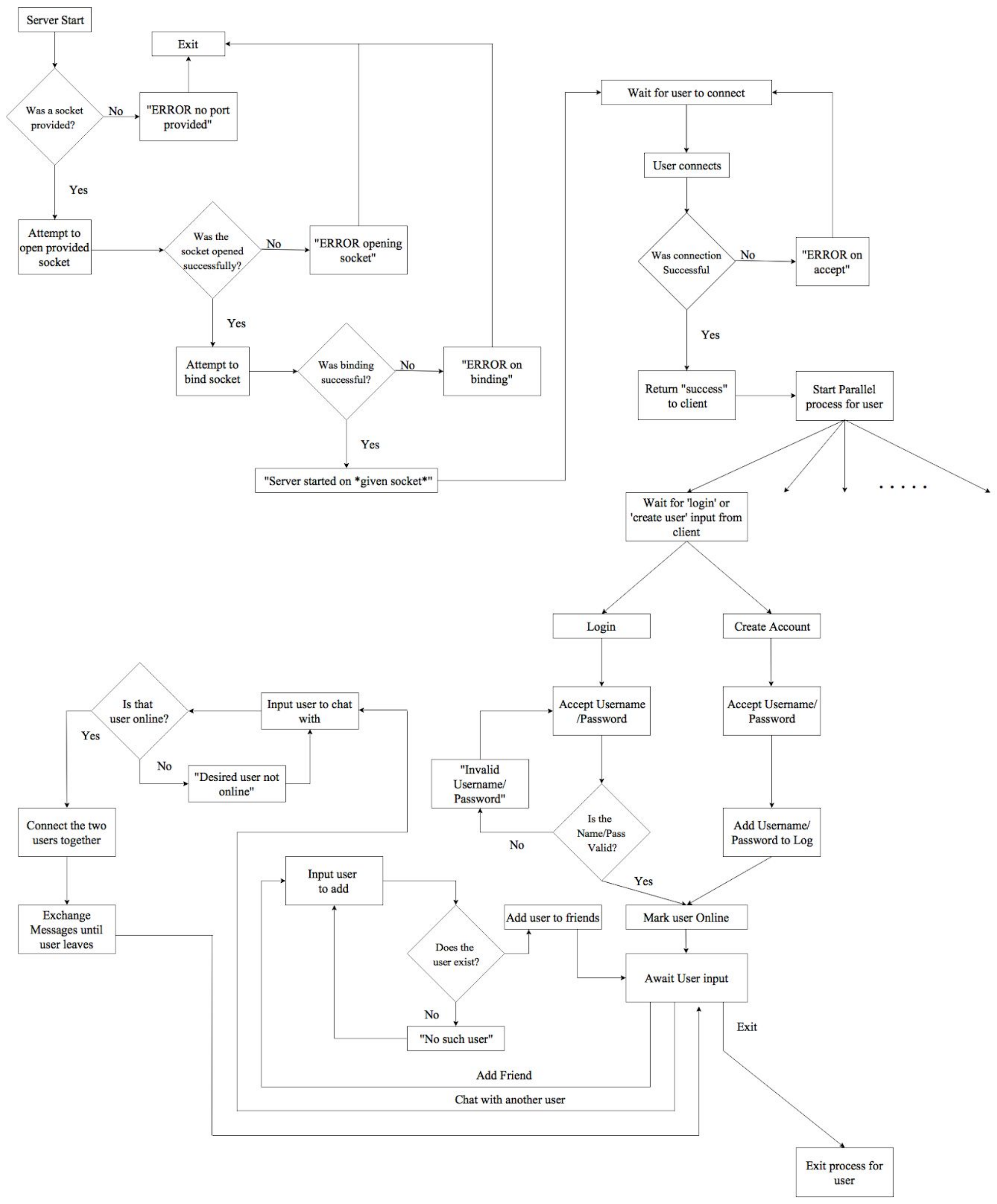


Figure 2.5 shows overall master control flow

3 Client-Server Protocol and Logic Flow

This is the control flow for handling the protocol for Login, Register, and traversing the sending messages. As noted above and again in section 3 under “Client-Server Protocol”, we will be adding more Protocol keywords such as Protocol for group messages or adding/removing friends. The Flowchart below accounts for LOGIN/REGISTRATION. More will be added as we progress our program.

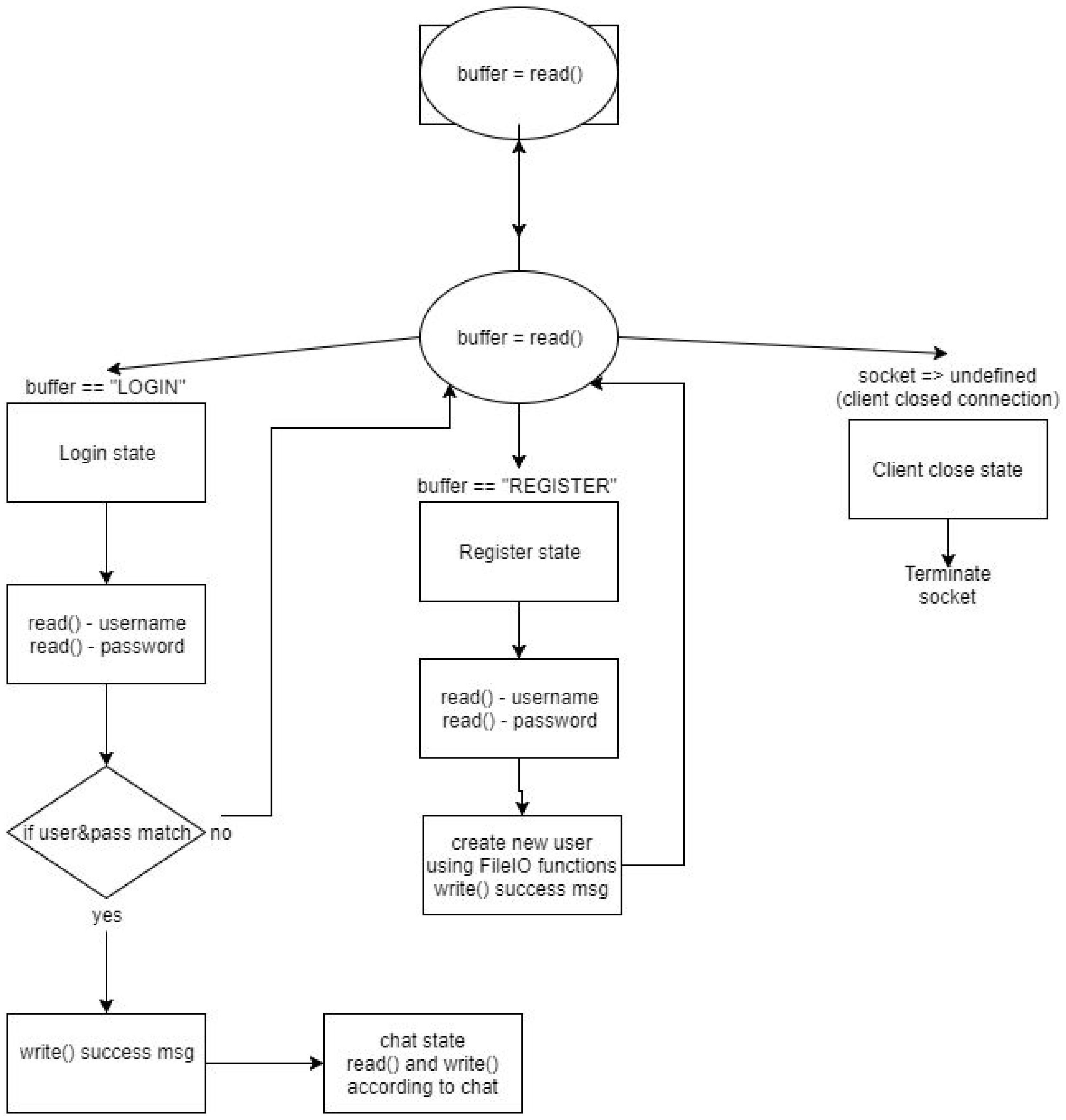


Figure 3.1 shows Client-Server Protocol and Logic Flow

As the Client and Server communicate on a read() and write() basis using C-strings, we will use these functions to follow a protocol. The First Message sent from the client to the server is hardcoded to send a command to the server. Depending on the "command" sent, there will be following parameters sent by the client using write() and received by the server using read(). For example, when a REGISTER protocol is sent to the server, then the two following parameters contain the user and encrypted password. Protocol definitions are defined below.

Please note this list is tentative since we are in pre-alpha release.

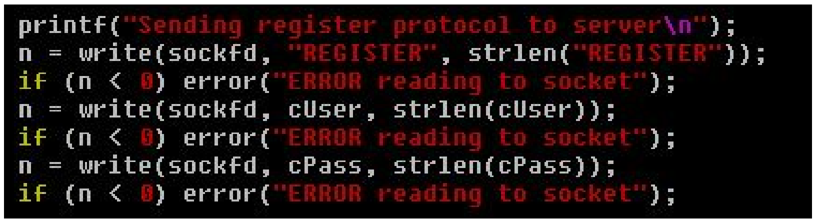
The protocol is as follows

- "REGISTER" <user> <pass> - Register a new user with a unique user ID and password
- "LOGIN" <user> <pass> - Compare the user ID and password to the masterlist of RegisteredUsers.txt
- "MESSAGE" <writer> <reader> - send an ordinary message to <reader>
- "MESSAGE <writer> <reader1> <reader2> - send a group message between 3 or more people

More to be added in the future, for example

- "DELETE" <user> - delete associated user.txt
- "ADDFRIEND" <user> <newFriend> - add a friend to a user's list
- "REMFRIEND" <user> <oldFriend> - remove a friend from a user's list

Example Code from the Client to write to the server a Register Protocol



```
printf("Sending register protocol to server\n");
n = write(sockfd, "REGISTER", strlen("REGISTER"));
if (n < 0) error("ERROR reading to socket");
n = write(sockfd, cUser, strlen(cUser));
if (n < 0) error("ERROR reading to socket");
n = write(sockfd, cPass, strlen(cPass));
if (n < 0) error("ERROR reading to socket");
```

Figure 3.1 shows the code from the Client to write to the server a Register Protocol

4 Installation

4.1 System Requirements

- OS: Linux Access on either: Windows 7 or higher, or MacOS
- Processor: 2.0 GHz Dual Core
- Memory: 2GB RAM
- Network: Broadband Internet Connection
- Storage: 500 MB
- Terminal shell

4.2 Setup and Configuration

- Open the command prompt (Terminal, MobaXterm, PuTTY)
- Login to SSH server
- Type: tar -xvzf Remarq_v1.0_src.tar.gz
- Find the directory called “bin/”
- Execute the program “./Remarq”

4.3 Uninstalling

- Open the command prompt
- Type “make clean” in the parent directory of the extracted folder

5 Documentation of Packages, Modules, and Interfaces

5.1 Data Structures

The User Structure will contain all significant information such as the username and password associated with the account. A pointer to its' personal profile is included, defined below

User
Char[20] FirstName
Char[20] LastName
Char[20] UserID
Char[20] Password
*Personal Profile

The Personal structure contains a pointer to the "parent" or the original user. It also contains a string "Handle" which is a changeable display name and a pointer to the Friends List. Notice that the latter two data types are dynamic, meaning they change. For this reason we separate User and Personal as this may have significance later in beta and final release versions where we dont want to pass user's credentials thus posing a security risk.

Personal
*User Master
Char[20] Handle
*FriendsList Friends

The FriendsList will be implemented as a double linked list seen before. The List will be of type *User. The FriendsList structure contains the vital data types for starting a List.

FriendsList
Int Length
***User Client**
***FriendListEntry First**
***FriendListEntry Last**

The entry of the FriendsList also follows a standard definition on an entry: It contains a pointer to the List, Next, Previous and the Friend of type *User.

FriendsListEntry
***FriendsList List**
***Entry Next**
***Entry Prev**
***User Friend**

Structures are subject to change for each release version. The Software Spec will reflect those changes

5.2 Functions and Parameters

- read( ) and write( )
    - These two functions serve as the basis of communication between the client and the server and allow the Protocol to exist. The write( ) function allows an application to send a string over the socket to another application. The read( ) function allows the application to receive a string over the socket. A socket is what connects two applications together. The parameters of read( ) and write( ) are the file descriptor, the string, and the string length in int.
- Under structure.h
    - Datatypes/Structures
        - Struct User
        - Struct Personal
        - Struct FriendList
        - Struct FriendEntry
    - Functions
        - User* CreateUser(char* First, char* Last, char* User, char* Password);
            - Creates an instance of a user in memory
        - DeleteUser(*User User)
            - Deletes an instance of a user in memory
        - Personal* CreatePersonal(char* Handle, User* parent);
            - Creates an instance of personal in memory
        - DeletePersonal(Personal* Personal);
            - Deletes an instance of Personal in memory
        - Void CreateFriendList(*user client);
            - Generates a FriendsList structure for a user
            - Formatted input from <user>.txt
                - <user>.txt is a input file defined below
        - Void AppendEntry(*user client, *user friend);
            - Append an entry of friend to add to user’s friendsList. Handles an empty List as well.
        - Void DeleteEntry(*user, *user friend);
            - Delete an entry of friend from user’s friendsList
        - More functions will be added as we expand our program
- Under Server.c
    - ServerLogin(char* user, char* password);
        - Compares the username and password to the database. If login is sucessful, we allow the client to operate under the user

  - ServerRegister(char* user, char* password);
    - Creates a new account with username and password
- Under FileIO.h
  - PrintUser(*user Parent);
    - Creates or overwrites a text document named after the username
    - Example: *Apollinax.txt (see 5.3.2)*
  - PrintMessageLog(*user Friend1, *user Friend2);
    - Creates or writes into the next open line of a text document named after the two users chatting.
    - Example: *ApollinaxBitConLog.txt (see 5.3.2)*
- Under Makefile:
  - Responsible for generating .o files and executable binaries
  - GNU compiler will be ran through here
  - “Make all” generates all executables
  - “Make clean” removes all .o files and binaries

5.3 Input and Output Formats

5.3.1 User Input Syntax and Format

- Pending the implementation of the GUI, the program will focus on its communication with the server via console
    - For our alpha release, the menu and input will be formatted through the console application. Our output (of registered users, for example) will be formatted into a .txt file using FileIO.o functions.
    - From the beta and onwards, our input will be sent through the GUI and no more console. The GUI will be set up to be integrated into the client application to handle all input. Output will be handled by FileIO.o as before.
- All input with GUI will use a familiar format similar to MSN Messenger, Steam Chat or Yahoo! Messenger. Windows for every chat window and a master window with the friends list and settings will be provided.

5.3.2 Log File Recordings

- The log files were defined above under section 2.2 (Software component of server). The server “database” will be composed of multiple .txt files, both encrypted and decrypted depending on the data contained in each .txt file.
- <user>.txt
    - Every user registered has a <user>.txt associated with them. It stores their login credentials (encrypted) and also stores a list of friends associated with the user.
    - In the future the user.txt will be used as a formatted input file to generate a user struct and <user>->FriendsList upon login.
    - Example: Apollinax.txt

Apollinax.txt

```
Apollinax  //username
123admin //password
John         //Firstname
Doe         // Lastname
Apolli      //Handle or TempID
--             //start friendslist of username
cez98
BitCon
RationalJustification
Tunmyat
lxyrylle
```

Figure 5.1 shows an example of Apollinax.txt file

- RegisteredUsers.txt
    - A file containing the login and passwords of all users on the server. For now this is a rudimentary implementation for the pre-alpha release to handle basic login/registration, and will be eliminated by beta phase. All data stored here will be encrypted.
    - Example: *RegisteredUsers.txt*

RegisteredUsers.txt

```
Apollinax 123admin
cez98 321nimda
BitCon admin123
RationalJustification administrator
Tunmyat rotartsinimda
lxyrylle zzbbygirl
```

Figure 5.2 shows an example of *RegisteredUsers.txt* file

- <user><user>Log.txt
    - A simple text log of every message exchanged between two or more users. Messages will be time stamped with the sender's userID.
    - Example: *ApollinaxBitConLog.txt*

ApollinaxBitConLog.txt

```
Apollinax [2/26/2018] 11:40:05 AM: Hey I finished the campaign to the
game
BitCon    [2/26/2018] 11:40:45 AM: We have 20 minutes to submit the PDF!
Apollinax [2/26/2018] 11:59:50 AM: I just submitted it ;)
BitCon disconnected [2/26/2018] 12:00:00 PM
```

Figure 5.3 shows an example of *ApollinaxBitConLog.txt* file

6 Development Plan and Execution Timeline

6.1 Task Partitioning

TASK	Bianca	Cesar	Joshua	Michael	Reza	Tun
GUI	x	x		x		
Server			x		x	x
Client		x	x	x		x
Protocol					x	
Data Struct		x	x	x	x	
Logs & Database	x					
I/O	x					x

Figure 6.1 shows the task partitioning for all the team members

6.2 Team Member Responsibilities

We partitioned the tasks as shown in 6.1 whereas each team member is collaborating with at least one other team member to accomplish each assigned task. In this way, we get the tasks done as efficient as possible. Timeline started as early as the first week of the project and will continue to be adjusted accordingly until week 10. Furthermore, each team member is expected to document their update on the servers via CVS, as well as work on the deliverables and changing them appropriately with respect to the time permitted and accomplished tasks.

7 Copyright

8 References

https://eee.uci.edu/18w/18020/schedule.html - lecture slides.

http://www.linuxhowtos.org/C_C++/socket.htm - intro to sockets, client-server architecture.

https://developer.gnome.org/gtk-tutorial/stable/ - GTK tutorials

8 Index