Danny Lopez and Ryan Lanphear

Group 8

ECEL 353 Project 4 Report

Professor Michael Lui

8/23/2017

**Chat Server Program**

For Assignment 4, a chat server-client program was made to act as an interface where a group of users can exchange messages locally. There are several requirements that had to be met. First, a server needs to be created that can host up to 10 users. Then, a separate client program was made for each of the users to use. Once connected, users can type messages directly into the command line and the messages would be displayed to all of the other users. Additionally, users can send direct messages to one specific user instead of the group. This program was accomplished by multithreading and shared memory communication.

**Usage:**

To compile the programs, a make file is used to compile all of the required programs to condense the work into a single line. In the directory with the programs, all you need to do is type *make*:



There will be several warnings when the files are compiled, but that is okay. Now the two executables that are needed are ready to be run.

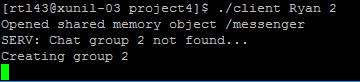
To start the server, type in the following to start the connection:



The two lines that should print out confirm that the status of the server is running.

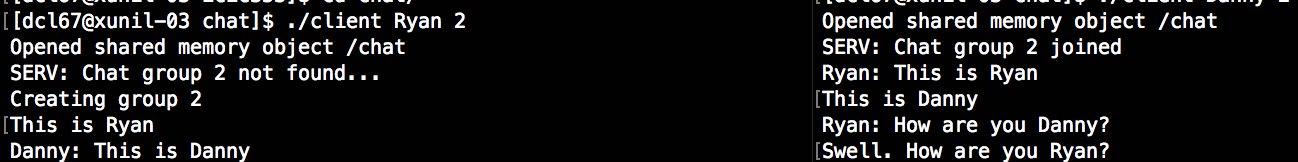
While communication is going on, the server will spit out information regarding the status and other messages that have been sent and received.

In a separate workspace the client can be started up using:

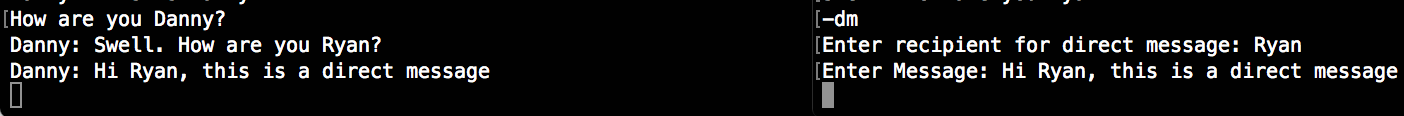


The usage is ./client [name] [roomID]. Since Ryan was the first user, the server created the group 2 and started communication. Ryan can now type to send information to the room.

A separate client is started and now the two can talk to each other. The following shows some interaction between the users in the chat room:



Also, a user can choose to send a direct message to a user instead of to the whole room, shown below:



If a client tries to connect to a server that does not exists, the following error tells the client that there is no server and exits the program.



**Coding Architecture:**

The code was split into 2 major files: server.c and client.c. There is also a makefile used to compile all of the code, a README.txt file for usage, and the header file chatserver.h.

**chatserver.h:**

This header file contained all of the definitions of global variables and constants that had to be shared between the client and the server, as well as the struct definitions that were used to pass messages and contain the groups.

**server.c:**

The server file is the basis for the whole set of communications and contains all of the server code. The first major part of this code is setting up the shared memory point in memory. The default place that shm\_open saves to is /dev/shm/ so our file that shares information is located there. Then mmap is used to get access to the shared memory. Once this is set up, threads are set up that will be used to make sure that no client is writing to the shared memory at the same time. Once the connection goes up, the server sits idle until a client connects, and once a client connects the server will set up the room by connecting the client to that room. The server also sends messages to the clients, so then a few functions were created to handle talking to the clients. It can send the group messages to all of the clients, send out error messages, inform clients of their status, and handle direct messages by checking to see who is connected. Once all of the clients disconnect, the server will shut down.

**client.c:**

The client file that is run by users contains all of the code that is needed to run the client’s messages. It connects to the shared memory location set up by the server and opens a connection to pull data from there. It looks for new messages to be shown to the user and displays both group messages and direct messages sent to that particular user on the interface while allowing them to enter input of their own. Most of the program is just opening up, maintaining, and closing a connection, as the server handles a lot of the message work. Messages are sent in packets to the shared memory before the server decides where to send the information.

**Challenges faced:**

One of problems faced was communicating between the applications in real time using shared memory. As this is a new process that neither of us had done before, we had to look up a lot of examples to figure out how to make it work. It took a lot of trial and error to mix the shm\_open commands with the threading to ensure that the clients weren’t sending information at the same time. Another problem was getting all of the features that we wanted to work. We wanted to add in a lot of different functionalities but were restricted by the requirements of the program as well as time. After playing around with different layout for the program, we finally found a configuration that made sense and that we could get to work. Another challenge was the testing phase, as it was hard to test a lot of the functionality of either file without having the other file working as well. It took a lot of pushing and shoving to get the programs to work at the same time to get real test results in.