CMPE 156: FINAL CHAT PROGRAM

A PREPRINT

Alexander Dobrota
Network and Digital Technology Major
University of California, Santa Cruz
`adobrota@ucsc.edu`

June 12, 2019

ABSTRACT

In the last weeks of my winter quarter, the professor assigned an arduous task: to build a TCP-based chat program that challenges the breadth of my foundation in Socket Programming in the C language. The chat program consists of a client for the end users and a server program which will be used to connect clients so that they can exchange messages. This report assumes audience to possess some familiarity with socket programming but for quick reference, I've added a page at the end of the report dedicated to defining key terms and concepts. The layout of this report will be dealt in several sections, each describing a stage of the development process. I'll first begin with a high level overview of what the project is by describing its specifications and then go on to discuss how I approached the problem, solved the problem, and finally, give a demo and finish with some concluding thoughts .

1 Project Specification

The client should operate in three modes: asking the server for information (INFO), waiting to be contacted by another client (WAIT), or exchanging messages with another client (CHAT).

On start up the client must connect first connect to the server. The user can then either tell the server it will be awaiting for a connection, request a list of users that are awaiting for connection, or request to connect with one of those users. If the user requests to wait, the client will wait until it receives a request for connection, then change to the conversation mode (CHAT). In conversation mode the clients can send messages to each other until one of them drops from the conversation, which results in both clients returning to the server (INFO).

1.1 Operation

The server should be concurrent in processing connections from clients. It is started with a port number. Example:

```
./server 1234
```

The client should be started with the IP address and port to the server and

an id. Example:

```
./client 128.114.104.54 1234 student1
```

All commands should be in the format `/<word>`, in which <word> is a sequence of lower case letters. A command that appears in the middle of a message can be ignored and treated as part of the conversation, e.g., the message ”Use the command `/quit` to leave chat” sent in CHAT mode should be displayed at the other client, not cause the user end the application.

The client program should ignore commands that follow the format but have not been implemented, displaying the message `’’Command /<word> not recognized’’`. For simplicity of implementation, it will be enough to check that the first character in the input is the slash (/) for purposes of identifying whether the user has sent a command or not. The client should also have a prompt

```
<id>\n>
```

available at all times, where `<id>` is the id provided when starting the client. While not at INFO mode the client should support the command */quit* to quit the application. It should also support the following commands while in INFO mode:

- */list* - get a list of identifiers of awaiting clients in alphabetical order.
- */wait* - switches to WAIT mode, while telling the server to add <id> to the list of awaiting clients to be shown by /list.

the list of awaiting clients to be shown by /list. The id should contain only letters and numbers. The client should also provide the server with a port number that it will use to receive the connection, this should be done without direct input from the user.

- */connect <id>* - start a conversation with the client identified by <id>.

Use Control-C to end the connected conversation. The server should provide the client with the address information necessary for the connection and remove it from its waiting list, or say that the other client is not waiting anymore if it is not in the waiting list anymore (e.g. because another client started a conversation with it in the mean time). The other client should be able to refuse the connection if already engaged in conversation.

- */quit* - Leave program

1.2 Example Usage

A example usage is listed below

```
> /list
1) Alice
2) bob
3) UsEr13
> /connect bob
Connected to bob
> Hi bob
>
bob: Hi how are you?
> Testing my chat program
>
bob: Oh ok
> Bye
> ^C
Left conversation with bob
> /list
1) Alice
2) UsEr13
> /wait
Waiting for connection.
>
Connection from bob.
>
bob: Is the waiting part also working?
> Yes
> ^C
Left conversation with bob
> /wait
Waiting for connection.
> ^C
Stopped waiting.
> /quit
```

Most of the logic is self explanatory. `/list` should present the current waiting clients, `/connect bob` should connect the user to a client named bob. One minor thing to note. In the conversation,the user has with bob, his name along with his message is displayed to the user which means I need to remember bob's name and combine it with his message before sending to the user. Notice that when the user left the conversation with bob, bob no longer shows up when the user inputs `/list` since bob no longer in the WAIT state; in fact he's now in the INFO state.

2 Design Strategy

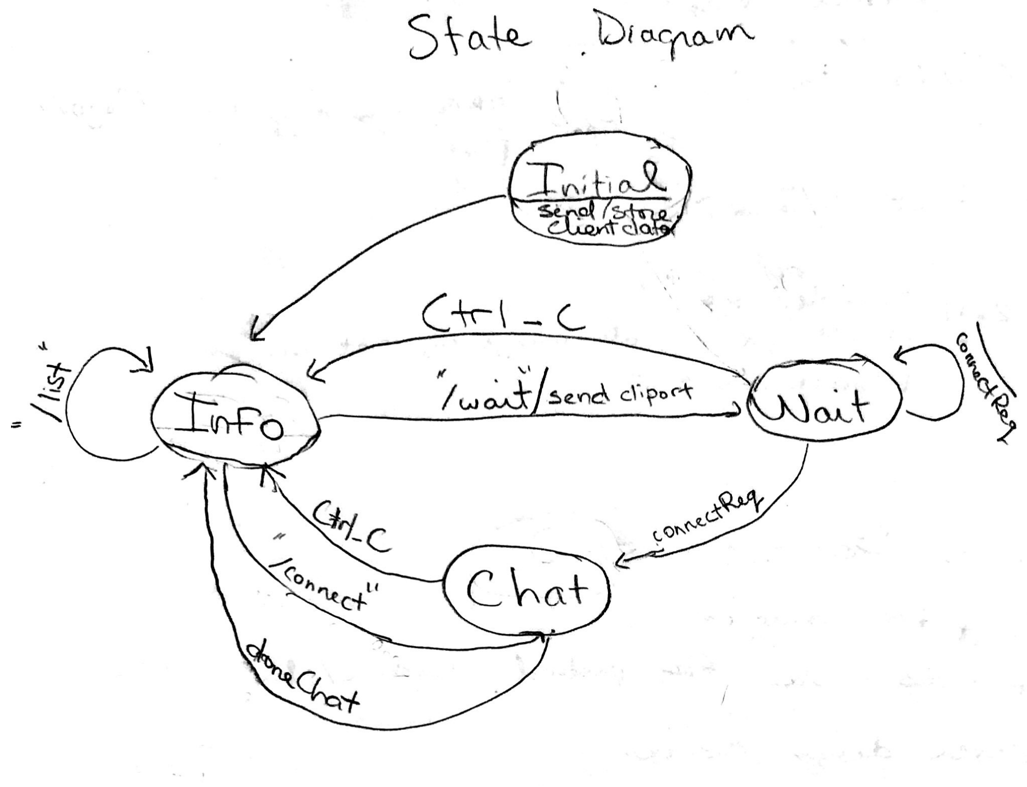


Figure 1: Client State Diagram

In figure 1, I drew up a state diagram on how I envision the different states a client can be in and the events that trigger a state change. When the program starts up, the client will be in the default state, INITIAL

INITIAL State:
In this state, several things will happen. The client will attempt to establish a connection to the server. Once the connection succeeds, the client program will send its client id, ip, and its port to the server. The server will take the following information about the client, initializes a struct client object with the data, and store the client in an array of clients.

The client will now automatically transition to the next state, INFO.

INFO State:
This is the main state. Here's where many states will transition back to rather than the INITIAL state. Client will remain in this state until an accepted user input is passed in. If the user types `/list`, the user will get see on the terminal (stdout) a list of all available clients, in alphabetical order, that they can connect to, and remain in this state.

Events #1: If user inputs `/wait`, the client listening port will be sent to the server, saved in their respective profile on the server, and transition to the WAIT state.

Events #2: If the user inputs `/connect <desired client's id>`, the server will check to see if the requested client exist in its array of clients and sends back an acknowledge of success if found or a failed one. If the user had selected a valid client it will transition to the CHAT state.

NOTE: If at any point in the state diagram, the user inputs the `/quit` command, the user should be alerted that the program is now closing and the program will exit successfully.

WAIT State:
There's not much happening in this state. Here the client is in a dormant status, waiting for someone to connect to them unless the following events occur.

Events #1: If someone does want to connect to this client, then the server should connect the two together and leave them be. As per the spec, no new clients should be able to connect to these clients; no more than two clients can talk to each other at a time.

Events #2: If the user presses the `Ctrl-C` on their keyboard, while in this state, the client will move to the INFO state

CHAT State:
Here's where the exciting stuff happens and also the toughest to implement. Now two clients can have their connections for however long they want and will remain in this state unless one event occur.

Events #1: If the user presses `Ctrl-C` on their keyboard, both clients will move from CHAT->INFO. doneChat is the same as `Ctrl-C`.

2.1 Server Design Choices

In designing the server, I made several implementation decisions. First off, I was required to design a concurrent server but it was up to me to choose they way the server will handle each clients simultaneously. In my head, I had two options: one was to create one process per client or two, create one thread per client.

Before we go any further, let's go over the advantage and disadvantage of choosing either multi threading or multiprocessing for our concurrent server.

One Child Process per Client:

Pros	Cons
the crash of a process does not affect the other processes	parents need to clean up after children using `waitpid()`
simplest server design choice -> all I need to do is `fork()` to create a child for each client	process do not share memory with each other
no need to worry about mutex management	creating new child continuously is a costly operation

One Thread per Client:

Pros	Cons
easier for sibling process to share info -> less overhead	inter thread communication causes deadlock -> requiring use of mutex and condition variables
easier to create and destroy threads -> more efficient	not as reliable as the other option

All threads within a process share the same global memory like

- Process Instructions
- Open File(File Descriptors)
- Signal handlers and Signal Dispositions (how the process behaves when it is delivered the signal)
- Current Working Directory
- User and Group IDs
- Global Variables - very useful when we want to keep a global array to hold all the clients

But each thread possess their own

- Thread ID - useful to know confirm the right thread are tasked to the right client
- Set of registers, including PC and Stack Pointer (SP)
- errno
- Signal Mask
- Priority

Not to mention threads are more efficient and faster to create than to create a new process with `fork()`. In fact, threads are considered *lightweight processes* since they can be up to 100 times faster than process creation. Considering that I need to share client information when two clients when they're trying to chat with each other and have a efficient server, the best solution would be to create a concurrent server with one thread per client, a implementation strategy I've used several times in the past assignments.

To use threads in C, I used POSIX threads, also known as Pthreads. The pseudo code to create a concurrent server using one thread per client looks something like this:

```
//Server MAIN

while(1){
        int i = 0; //keep tracks of thread #
        acceptClient();
        i++;
        //worker_args = a struct var to store client data in
        worker_args * threadargs = calloc(1, sizeof(worker_args));
        saveClientData(clientip, clientport, i, clientfd)
        if( pthread_create( &tid , NULL ,  handleConnection , (void*) threadargs) < 0){
                perror("could not create thread");
                return 1;
        }
}

printf("finished with thread #%d\n", i);
```

First thing first, we want the server to run continuously which we can achieve through `while(1)`.

Then we create a counter variable, which I will use to keep track all the threads that will be created through new clients. Although I can use this as a way ot identify the client, I'm also receiving the client id , which ,in the context of my application, is a much more useful form of client identification. The counter variable is mostly using for debugging purposes.

Next up, we have the creation of a thread. Notice how I have a variable called threadargs. As it names implies, this variable is a struct object containing the client data that we will send to the newly created thread. `pthread_create()` preforms the actual thread creation. It's signature is :

```
#include <pthread.h>

//return 0 if success, positive if error
int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*func)(void *), void *arg);
```

`func` and `arg` are the most important arguments to watch out for. The function takes one argument, a generic pointer (void *), and returns a generic pointer (void *). This lets us pass one pointer (to anything we want) to the thread, and lets the thread return one pointer (again, to anything we want). If `pthread_create()` cannot create a new thread because of exceeding some system limit on the number of threads,the function return value is EAGAIN.It's important to note that the Pthread functions do not set errno variable. In our case, when `pthread_create()` fails, I will send an FAILED message to create thread to stderr, based on the error state stored in erno and return 1 or `EXIT_FAILURE`.

`handleConnection` is the function that the thread execute. It handles inserting the client data to the global array in alphabetical order and handling all client requests the server receives through the client socket. All this functionality will be explained in the next section of this report. See Section:3

2.2 Client Design Choice

Now I had to decide how to implement the client side of this application. When I was looking at the project requirements, I noticed a future headache. I needed to share an annoyingly amount of information and simultaneous inputs to worry about: I had to share the state of clients, data that clients send to each other, synchronously check for user input for commands, and also listen for any data from the server. All events that could be happening at the same time!

Worrying about deadlocks is something I didn't want to have to fear if I had a better choice. Luckily for me, I found a better solution: synchronous I/O multiplexing in a process with `select()`. `Select()` gives me the power to monitor several sockets at the same, similar to a interrupt handler which gets activated as soon as any file descriptors(fds) sends any data. I can tell which file descriptors are ready for reading, ready for writing, and even which one have triggered exceptions.

Its function signature is as followed:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout);
```

`select()` is relatively easy to understand. The function observes three set of file descriptors: readfds, writefds, and execptfds for any activity. The file descriptors listed in readfds will be watched to see if characters become available for reading (more precisely, to see if a read will not block; in particular, a file descriptor is also ready on end-of- file). The file descriptors in writefds will be watched to see if space is available for write (though a large write may still block). The file descriptors in exceptfds will be watched for exceptional conditions. The last argument *timeout* determines the length of time that `select()` should block waiting for a file descriptor to become ready. *Taken from select(2) man page.*

The struct timeval has the follow fields:

```
struct timeval {
int tv_sec;      // seconds
int tv_usec;     // microseconds
};
```

NOTE: `select()` will block until one of the following conditions occur:

- the timeout expires
- the call is interrupted by a signal handler
- or lastly, a file descriptor becomes ready

Now let's look at the macros that we will use in our application to manipulate the sets of fds.

```
//Add fd to the set.
FD_SET(int fd, fd_set *set);

//remove fd from the set
FD_CLR(int fd, fd_set *set);

//return true is fd is in the set
FD_ISSET(int fd, fd_set *set);

//clear all entries from the set
FD_ZERO(fd_set *set);
```

Now that I know everything there is to know about `select()`, I now have a better idea of how I want to implement the client side of this application. In my mind I envision the following:

```
1  //SELECT LOGIC:
2
3  // run through the existing connections looking for data to read
4    for(int i = 0; i <= fdmax; i++) {
5          if (FD_ISSET(i, &read_fds)) {
6            if (i == STDIN){
7                  //recieved data from stdin
8                  }
9            else if (i == listener) {
10                 //recieved connection request
11             }
12              else if (i == serverfd){
13                 //recieved data from server
14             }
15           else if (i == clientfd){
16                 //recieved data from a chat client
17             }
18          }
19   }
```

Notice the variable `fdmax`. fdmax is the value of the highest file descriptor, plus one, to ensure we go through all the possible file descriptors we have. In *Lines 5* and onward, I check which file descriptors are ready to read from and preform the relevant operations, depending on what state the client is in at that moment.

3 The Program - Code Analysis

In this section, I will present my code, both the client and server side of my application. Since they are rather long, I will discuss the code in part and hopefully give the readers enough detail, so that they can recreate a chat program on their own. Most of the code I present will have lots of documentation to support it but I will chime in to note any important details that I think are worth mentioning. That said, let's move on. First on the list, is the Makefile which we will use to compile and build our two executable files: client and server.

3.1 Makefile

```
##Makefile for the Final
## Author: Alexander Dobrota
## Date: 01/24/19
CC = gcc
CFLAGS  = -g -Wall -O0 -std=gnu99 -D_DEFAULT_SOURCE

all: client server

client: client.c
${CC} ${CFLAGS} -o  client client.c -lpthread

server: server.c
${CC} ${CFLAGS} -o server server.c -lpthread

clean:
${RM} client server
```

A very simple Makefile. There's not much to say about other than the -lreadline -lpthread options. -lpthread allows us to compile our C program with pthread.h library since GCC does not include the pthread library by default.

Now let's move on to server.c!

3.2 server.c

We will first look at the main function, the global variables and then follow main's called functions around the file.

```
1  //MAIN- First Part
2  int main(int argc, char * argv[]){
3     //var to hold client data
4     struct sockaddr_in client_addr;
5     socklen_t client_size = sizeof(client_addr);
6     pthread_t tid; //holds our thread
7
8     if(argc != 2){
9        perror("expected input: ./myserver <port
             number> \n");
10       exit(1);
11    }
12
13    int port;
14    if((port = atoi(argv[1])) < 0){
15       perror("invalid port \n");
16       exit(1);
17    }
18    //creating socket for client
19    int clientlistensockfd =
          createBindedPort(port);
20
21    //LISTENING TO CLIENT SOCKET
22    if(listen(clientlistensockfd,
23       MAXCLIENTS)<0){
24       perror("error: listen() failed");
25       exit(1);
26    }
```

```
1  //GLOBAL SPACE
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <string.h>
6  #include <sys/socket.h>
7  #include <netinet/in.h>
8  #include <arpa/inet.h>
9  #include <sys/types.h>
10 #include <sys/stat.h>
11 #include <fcntl.h>
12 #include <string.h>
13 #include <pthread.h>
14 #include <readline/readline.h>
15 #define MAXCLIENTS 10 //# of clients server can
       handle at a time
16
17 typedef struct{
18 char * id;
19 char * ip;
20 int port;
21 } clientinfo;
22
23
24 typedef struct{
25 int socket;
26 int clientport;
27 char * clientip;
28 int clientnum;
29 }worker_args;
30
31
32
33 clientinfo clients[MAXCLIENTS];
34 clientinfo waitinglist[MAXCLIENTS];
35
36 static int openspotindex = 0;
37 static int waitingopenindex = 0;
38
39 static int clientssize = 0;
40 static int waitlistsize = 0;
41
42 pthread_mutex_t lock;
```

Let's talk a little bit about the first part of main(). *Lines 4-6*, I declared variables to hold client data and a thread of type pthread_t. *Lines 8-10*, I checked to see if user inputs the right number of arguments. Otherwise, the program will exit with an error. Once that error checking is done, it's now time to grab the port of the server, first making sure the port is a valid one. Then in *Line 19*, we bind the server socket to the port. By calling `listen()`, the socket is converted to a listening socket, in which incoming connections from clients will be accepted by the kernel.

One the right side of the page, we have all the global variables, struct, arrays, and a mutex lock. The works_args struct will hold all the client data: it's ip address, port, socket fd, client number. I also have two global arrays, clients and waitinglist of size MAXCLIENTS. Clients will hold all the connected clients to the server, while `waitinglist` holds just the ones that are in the WAIT state. In *Line 29-30*, I have indexes for the two arrays and variables containing their current size. They will be very useful in keeping track of which position of the arrays are currently available to store a new client. Lastly, we have a mutex *lock* that we'll use to prevent threads from accessing the global arrays at the same time, triggering deadlock condition which is problematic.

Next up, we will look at second part of main and the thread function, `handleConnetion()`

```
//MAIN Pt.2
int i = 0;
//Main LOOP
while(1){
   int clientfd =
       accept(clientlistensockfd,(struct
       sockaddr *) &client_addr, &client_size);
   char * clientip =
       inet_ntoa(client_addr.sin_addr);
   int clientport = (int)
       ntohs(client_addr.sin_port);
   i++;

   worker_args * threadargs = calloc(1,
       sizeof(worker_args));
   threadargs->clientnum = i;
   threadargs->socket = clientfd;
   threadargs->clientport = clientport;
   threadargs->clientip =
       (char*)malloc(sizeof(char) * 100);
   strcpy(threadargs->clientip, clientip);
   if( pthread_create( &tid , NULL ,
       handleConnection , (void*) threadargs)
       < 0){
     perror("could not create thread");
     return 1;
     }

   printf("finished with thread #%d\n", i);


   }
   return 0;

}
```

```
void * handleConnection(void * args){
   char clientbuff[512];
   bzero(clientbuff, sizeof(clientbuff));
   worker_args * data = (worker_args *)args;

   int clientsock = data->socket;
   //clientbuff will contain the client's
       id/username
   read(clientsock, clientbuff,
       sizeof(clientbuff));

   clientinfo * myclient =
       initClientData(data->clientip,
       clientbuff, data->clientport,
       strlen(clientbuff));
   readclientbuffer(clientsock, myclient);
   close(clientsock);

   pthread_exit(NULL);

}
```

I've already mentioned a lot of what I wanted to say in the second part of main in Section 2.1. If you want to see what I wrote, go to that section. 2.1. But what's different here is that now we can see what's beings stored in `workargs`. We pass the struct into the thread's executing function `handleConnection()` which we will take a look at next.

`handleConnection()` is essentially the main function that every thread executes in this program. It's an easy one to spot with its void * return type. In this function, I initialize a worker_args struct object `data` with a casted `args`. I then read from the client socket to get the clients id and store it into an array of chars, `clientbuff[]`. Once we have all data from the client, I initialize a variable of type clientinfo that holds the client's id, ip and port and pass it through another helper function `readclientbuffer()` along with the client socket fd. After the helper function finishes, the thread's job is done, I close the client socket fd so it can be reused and terminate the thread with `pthread_exit(null)`

`readclientbuff()` is a long function but it's relatively easy to follow. Now we see in action the protocol that exist between the client and the server.

```
//READCLIENTBUFFER()
int  readclientbuffer(int socket, clientinfo * myclient){
        puts("trying to read clientbuffer now");
        char buffer[512];
        while(1){
                bzero(buffer, sizeof(buffer));
                printf("trying to read command from client %s\n", myclient->id);
                int recvsize = recv(socket, buffer, sizeof(buffer), 0);
                if(recvsize > 0){
                        printf("buffer = %s\n", buffer);
                        puts("got msg");
                        if(strcmp(buffer, "getlist")== 0){
                                puts("handling getlistcommand");
                                bzero(buffer, sizeof(buffer));
                                puts("before handlinglist");
                                handleGetListCmd(buffer,socket);
                                puts("after handlinglist");
                                bzero(buffer, sizeof(buffer));
                        }
                        if(strcmp(buffer, "waiting")== 0){
                                puts("got waiting msg");
                                bzero(buffer, sizeof(buffer));
                                recv(socket, buffer, sizeof(buffer), 0);
                                printf("port = %s\n",buffer);
                                int listenport = atoi(buffer);
                                myclient->port = listenport;
                                printf("MYCLIENT-> port = %d\n", myclient->port);
                                inputclientdata(myclient);
                        }
                        if(strcmp(buffer, "chat")== 0){
                                puts("got chat msg");
                                bzero(buffer, sizeof(buffer));
                                recv(socket, buffer, sizeof(buffer), 0);
                                char reqid[strlen(buffer)];
                                strcpy(reqid, buffer);
                                printf("requested id = %s\n",reqid);
                                int foundidwithport = checkWaitingListforID(buffer);
                                printf("port num is %d\n", foundidwithport);
                                char datareq[512];

                                //SENDING the REQ CLIENT DATA
                                sprintf(datareq, "connectreq %d username %s", foundidwithport, reqid);
                                //sending req client port +
                                //sending req client id
                                write(socket, datareq, strlen(datareq));
                        }
                }
                else{
                 break;
                }


        }

return 0;

}
```

The server expects three types of formatted request from the client: "getlist", "waiting", and "chat". If none of these requests are sent, the server will return an error and exit. Now let's see how the server will react to these requests.

"getlist":
If the client wants the list, helper function `handeGetListCmd()` is called and will return the entire content of the alphabetized global array, `waitinglist` back to the client through the client socket fd from the function's parameter, `socket`.

"waiting":
Looking at the state diagram from Section 2, we can infer that the client is trying to transitions state INFO -> WAIT. As such we know we need to read from the client socket, since per my protocol design, the client needs to send it's listening port that a new client can reach it from. Once done, we input the client in the `waitinglist` array using `inputclientdata()`.

"chat":
A client is trying to go INFO->CONNECT. To make this possible, we need to see if the requested client id is in the `waitinglist` array by calling `checkWaitingListforID(buffer)`. The helper function will return -1 if it cannot find the requested client id. I'll go in more details with this function later on. In Line 45, we write all waiting client's info that the client will need to connect to them.

3.2.1 Where are the Mutex?

Since we have multiple threads trying to access shared variables and arrays, possibly at the same time, how do we deal with the potential race conditions problem? The answer is through the use of mutex. In my server program, there are two occasions where I need to protect multiple threads from accessing shared data and changing it at the same time: first time is when I'm trying to insert a client into a shared array in `inputclientdata()` and second, when I'm searching the shared `waitinglist` array for a requested client in `checkWaitingListforID()`. I'll show you how to protect shared data with mutex in C by displaying both helper functions side by side.

```c
/**
*  checkWaitingListforID()
* Checks to see if the waiting queue has the requested id
*
* Returns the port num of requested client if found
*        -1, Otherwise
* */
int checkWaitingListforID(char * requestedclient){
int size = waitlistsize;
int port = -1;
int foundclientindex = 0;

for(int i=0;i< size;i++){
   if(strcmp(waitinglist[i].id, requestedclient) == 0){
      puts("found requested client");
      port = waitinglist[i].port;
      foundclientindex = i;
   }
}
pthread_mutex_lock(&lock);

openspotindex = foundclientindex;
clientssize --;
waitingopenindex= foundclientindex;
waitlistsize --;

pthread_mutex_unlock(&lock);

return port;
}
```

```c
    //inputclientdata()
    void inputclientdata(clientinfo*myclient){
    pthread_mutex_lock(&lock);
    printf("waitingopenindex = %d\n",waitingopenindex );
    printf("waitlistsize = %d\n",waitlistsize);
    waitinglist[waitingopenindex].id = myclient->id;
    waitinglist[waitingopenindex].port = myclient->port;

    clients[openspotindex].id = myclient->id;
    clients[openspotindex].ip = myclient->ip;
    clients[openspotindex].port = myclient->port;

    openspotindex ++; clientssize ++;
    waitingopenindex++; waitlistsize ++;
    pthread_mutex_unlock(&lock);

}
```

In both functions we can see that I call `pthread_mutex_lock(&lock)`.

NOTE: Once we do this, any other calls to `pthread_mutex_lock(&lock)` will not return until you call `pthread_mutex_unlock` in this thread.

Once we lock the shared mutex variable `lock`, we can now access and modify shared data without any worry for race conditions or data synchronization.

4 client.c

Phew, that was a lot of code and there's still more to go over. But don't worry, we're almost at the end. Hang in there!

client.c isn't as compartmentalized and as modular in design as server.c was. In client.c we have one big main function where I preform Synchronous I/O multiplexing through `select()`. Since it's so long, (over 300 lines of code). I'll refrain from blowing this paper with code, so I'll be giving snippets and summarized code. If you want the full code, feel free to email me. My email is at top of this paper. Now, let's check client.c out!

```c
//Main PT.1
int main(int argc, char * argv[]){
   int port;
   if(argc != 4){
   perror("expected input: ./client
       <server-ip> <server port> <client
       identifier> \n");
   exit(1);
   }
   char * id = (char *)calloc(strlen(argv[3]),
       sizeof(char));
   strcpy(id, argv[3]);
   if((port = atoi(argv[2])) < 0){
      perror("invalid port \n");
      exit(1);
   }
   signal(SIGINT, intHandler);

   int state = INITIAL; // *state variable*
   int serverfd =
       createConnectedServerPort(port, argv);

   char * username = malloc(strlen(id) + 2);
   strcpy(username, id);

   char other_username[512];
   write(serverfd, id, strlen(id) );
   char * userinput;
   size_t userinputsize = 512;
   size_t characters;
   userinput = (char *)malloc(userinputsize *
       sizeof(char));

   char requestedid[100];

   socklen_t addr_size = sizeof(struct
       sockaddr_in);
   struct sockaddr_in addr;
   char buf[1024];
```

```c
#define INITIAL 0
#define WAIT 1
#define INFO 2
#define CHAT 3

#define STDIN 0 // file descriptor for
    standard input
static volatile int Ctrl_C = 0;
void intHandler(int dummy) {
   signal(SIGINT, intHandler);
   Ctrl_C = 1;
}
```

If you remember from the first section, part of the requirement states that when user's press Ctrl_C on their keyboard, they should leave their current state, which will put them in the INFO state. Because of that I need to watch when that happens through the use of a signal handler. In our case, `intHandler(int dummy)` is the signal handler that is called when a specific signal, SIGINT (signal triggered by Ctrl_C press), occurs. We catch the signal in the signal handler and set the global variable `Ctrl_C` to 1.

Other than that, in this part of the code, we connect our client to the server in and send their client id there while in the INITIAL state.

```c
//--START HERE---

fd_set master; // master file descriptor list
fd_set read_fds; // temp file descriptor list for select()
int fdmax;      // maximum file descriptor number

int listener;  //descriptor for listener socket when in WAIT state
int clientfd;  // newly accept()ed socket descriptor for client

// clear the master and temp sets
FD_ZERO(&master);
FD_ZERO(&read_fds);

// add STDIN to the master set
FD_SET(STDIN, &master);
// add the server socket to the master set
FD_SET(serverfd, &master);
fdmax = serverfd;
listener = -1; // no listener before WAIT state

state = INFO;

struct timeval tv;
//set timeout for select() to 0.5 seconds in order to check for Ctrl-C
tv.tv_sec = 0;
tv.tv_usec = 500000;

writeprompt(username);
```

```c
   for(;;) {
   read_fds = master; // copy it
   if (select(fdmax+1, &read_fds, NULL, NULL, &tv) == -1) {
      if ( EINTR == errno) {
         FD_ZERO(&read_fds);
      }
      else {
         perror("select");
         exit(4);
      }
   }
   //first handle Ctrl-C
   if (Ctrl_C) {
      if(state == CHAT) {
         close(clientfd);
         FD_CLR(clientfd, &master);// remove from master set
         printf("Left conversation with %s\n", other_username);
         //writeprompt(username); //print prompt
         state = INFO;
      }
      else if(state == WAIT) {
         //exit from WAIT
         close(listener);
         FD_CLR(listener, &master);
         printf("Stopped waiting.\n");
         writeprompt(username); //print prompt
         state = INFO;
      }
   //else don't do anything (?)
   Ctrl_C = 0;
}
```

I have `master` hold all the currently connected socket descriptors, as well as the socket descriptors that is listening for new connection and I also set the `state` variable to INFO. I also check to see if any socket descriptor sends a Cntrl_C signal in which I set `state` to the new appropriate state.

```c
//MAIN CONTINUED -> SELECT() Logic Now
// run through the existing connections looking for data to read
for(int i = 0; i <= fdmax; i++) {
        if (FD_ISSET(i, &read_fds)) { // we got data from one fd
                if(i==STDIN){
                        characters = getline(&userinput, &userinputsize, stdin);
                        userinput[characters-1] == 0;
                        if(STATE == CHAT){
                                send(clientfd, userinput, sizeof userinput, 0)
                        }
                        else{
                                //check if user put in getlist, wait, connect and send the req to the s
                                //if not puts(wrong command)
                        }
                        writeprompt(username); //print prompt
                        continue;
                }
                else if(i == listener){
                        clientfd = accept(listener, (struct sockaddr *)&addr, &addr_size);
                         // close (listener);
```

```
            FD_CLR(listener, &master);// remove from master set
            FD_SET(clientfd, &master); // add client to master set
            //read first message that contains "Connection from %s", username);
            STATE = CHAT
            writeprompt(username); //print prompt
            continue;
        }
        else if(i == serverfd){
            //check to see if server hasn't shut down
            else{
                if(strstr(buff, "connectreq")!= NULL)
                    //get "port" and "other_username" from buf coming from server
                    state = CHAT;
                    clientfd = createChatConnection(requestedport);
                    FD_SET(clientfd, &master); // add to master set
                else{
                    //server returning "getlist" request
                    //print everything from the server to the console
                }
            }
            writeprompt(username); //print prompt
            continue;
        }
        else if(i == clientfd){
            if((nbytes = recv(i, buff, sizeof buff, 0))<=0){
                close(i); // bye!
                FD_CLR(i, &master); // remove from master set
                printf("Exit conversation with %s\n", other_username);
                //writeprompt(username); //print prompt
                state = INFO;
            }
            else{
            // we got some data from a client
            printf("\n%s:%s\n",other_username,buf);
            }
            writeprompt(username); //print prompt
            continue;
        }
    }
}
close(serverfd);
```

We continue with main and we can now see the select logic in more details. I condensed the code quite a bit for brevity but the overall idea is still there. In this part of main, we are monitoring for any data coming from any of the four file descriptors: STDIN, listener, serverfd, or clientfd.

STDIN
We checking for user input here. If the user(client) is in CHAT mode, the user input should be sent to its connect client through `clientfd`, otherwise, we check which of the (*getlist, wait, connect*) requests the user wants, and send it to the server. Then we finish this block with a writepromppt, which displays the required user id to the terminal.

listerner
If we reached this part of the code, we got a client to connect to! We `accept()` the new client on clientfd, and close and remove listener from master since we don't need it any more. We add the client to the master chat. We now transition state, WAIT->CHAT, and read the first message from the other client.

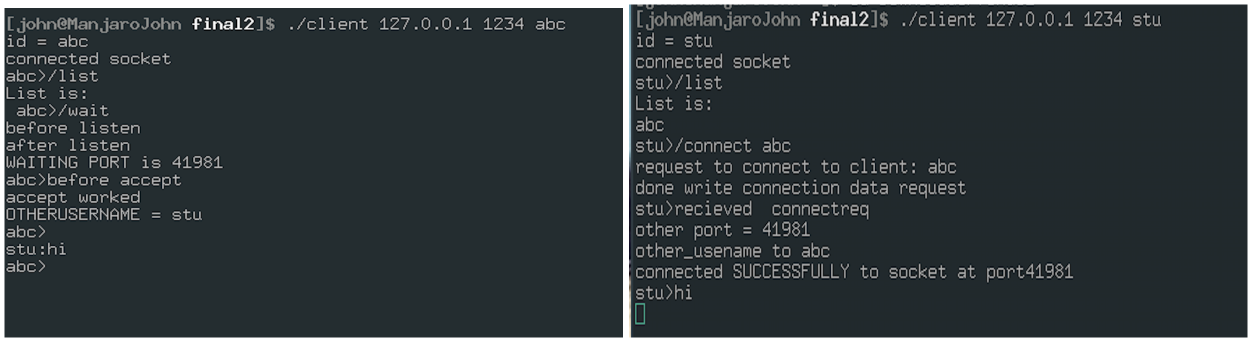
serverfd
We check if the server is still valid. Then we see if the attempted data transfer has the term connectreq in it. If so, then the user is ready to go to CHAT mode and create a connection to the other client.

If the server didn't send a connectreq message, the only other option would be the server is trying to send back the result from the "getlist" request we sent earlier. We print the server's response to the console.

clientfd
Lastly, we try to continue the CHAT state unless the other client closes the connection, after which we print *bye* to the console and go to INFO state.

5 Demo

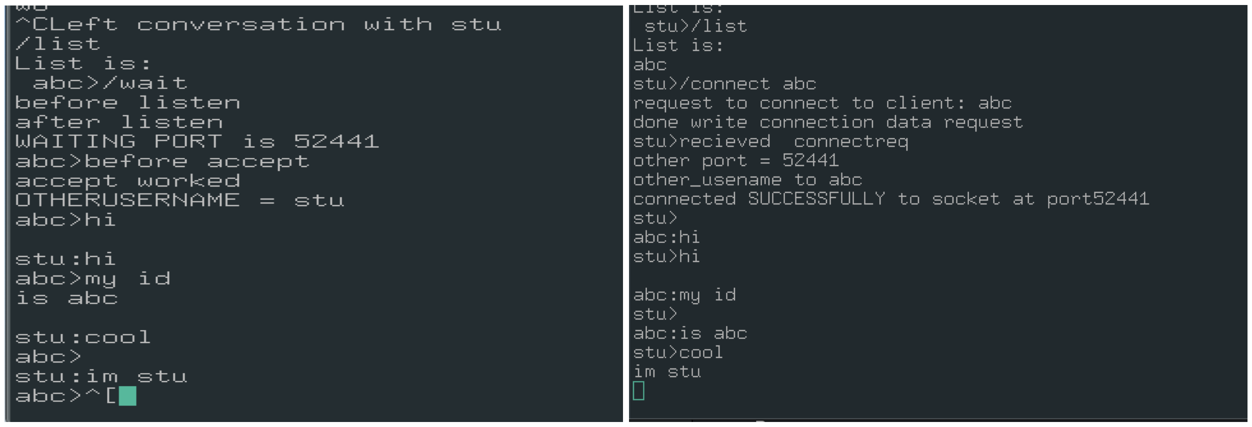
Here, I'll show an example conversation between two clients, *abc* and *stu*.



```
[john@ManjaroJohn final2]$ ./client 127.0.0.1 1234 abc
id = abc
connected socket
abc>/list
List is:
 abc>/wait
before listen
after listen
WAITING PORT is 41981
abc>before accept
accept worked
OTHERUSERNAME = stu
abc>
stu:hi
abc>
```

```
[john@ManjaroJohn final2]$ ./client 127.0.0.1 1234 stu
id = stu
connected socket
stu>/list
List is:
abc
stu>/connect abc
request to connect to client: abc
done write connection data request
stu>recieved  connectreq
other port = 41981
other_usename to abc
connected SUCCESSFULLY to socket at port41981
stu>hi
```

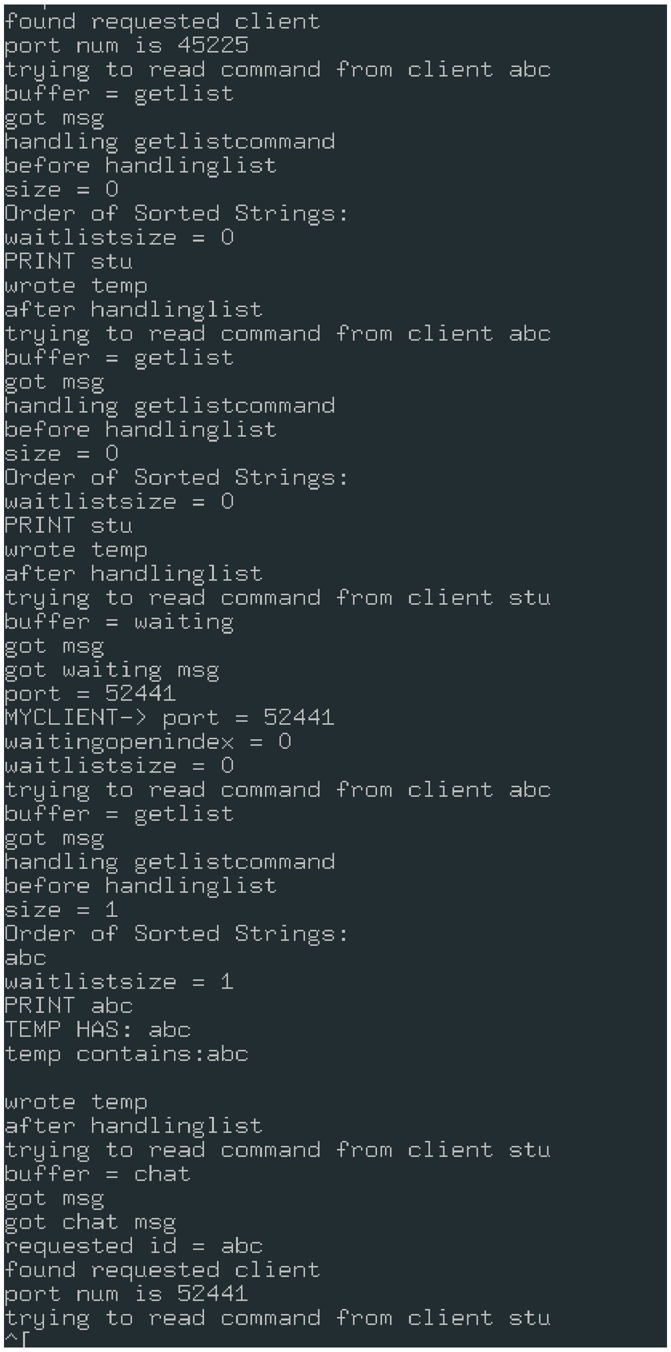
Figure 2: Beginning of a good conversation



```
^CLeft conversation with stu
/list
List is:
 abc>/wait
before listen
after listen
WAITING PORT is 52441
abc>before accept
accept worked
OTHERUSERNAME = stu
abc>hi

stu:hi
abc>my id
is abc

stu:cool
abc>
stu:im stu
abc>^[
```

```
 stu>/list
List is:
abc
stu>/connect abc
request to connect to client: abc
done write connection data request
stu>recieved  connectreq
other port = 52441
other_usename to abc
connected SUCCESSFULLY to socket at port52441
stu>
abc:hi
stu>hi

abc:my id
stu>
abc:is abc
stu>cool
im stu
```

Figure 3: Conversation's going great!



```
found requested client
port num is 45225
trying to read command from client abc
buffer = getlist
got msg
handling getlistcommand
before handlinglist
size = 0
Order of Sorted Strings:
waitlistsize = 0
PRINT stu
wrote temp
after handlinglist
trying to read command from client abc
buffer = getlist
got msg
handling getlistcommand
before handlinglist
size = 0
Order of Sorted Strings:
waitlistsize = 0
PRINT stu
wrote temp
after handlinglist
trying to read command from client stu
buffer = waiting
got msg
got waiting msg
port = 52441
MYCLIENT-> port = 52441
waitingopenindex = 0
waitlistsize = 0
trying to read command from client abc
buffer = getlist
got msg
handling getlistcommand
before handlinglist
size = 1
Order of Sorted Strings:
abc
waitlistsize = 1
PRINT abc
TEMP HAS: abc
temp contains:abc

wrote temp
after handlinglist
trying to read command from client stu
buffer = chat
got msg
got chat msg
requested id = abc
found requested client
port num is 52441
trying to read command from client stu
^[
```

Figure 4: Server logging the two clients' requests during this time

6 Concluding Thoughts

I hope my paper gave you a good idea of what my final project was. The chat program was a fun one to design and implement. However, like with any final projects, it had its up and downs. The toughest part of this project was writing client.c. It took me a long time to figure out a viable solution to deal with simultaneous inputs and one that could also deal with the whole connection process between two process without spawning more worker threads. But I knew if I took this path, my program would likely be very buggy: there's bound to be a delay between threads communication to each other, or a client could be waiting to access a shared variable for a very long time if there's multiple threads also trying to access the same resources. Luckily, when I found out about synchronous I/O multiplexing, I found a solution that was efficient and most importantly, easy to understand and implement. If I had more time to do this project, I would have redesigned the server program to also utilize `select()`.Thank you for reading. Again, if you want to see the full code of the chat program, feel free to email me.

Key Terms

Sockets: a way to speak to other programs using standard Unix file descriptors.

File Descriptors: A file descriptor is simply an integer associated with an open file. That file can be a network connection, a FIFO, a pipe, a terminal, a real on-the-disk file, or just about anything else. Everything in Unix is a file! So when you want to communicate with another program over the Internet you're gonna do it through a file descriptor. In our case we obtain the necessary file descriptors through calling the socket() system routine.

Threads: A Thread is a single sequence stream within a process. It is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

Process: A process is an instance of a program running in a computer.A process can initiate a subprocess, called a child process. Each child process is a replica of the parent process and shares some of its resources, but cannot exist if the parent is terminated.

Protocol: A network protocol defines rules and conventions for communication between network devices.

Mutex: A mutual exclusion object is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously