Multi-Client Chat Service

Charles Lovering

July 18, 2016

CS 513: Introduction to Local and Wide Area Networks

Contents

1 Abstract

This report details the design, development and testing of a chat service program. It uses the client-server model, supporting multiple clients. The program was written in C to run under the Unix operating system, and developed on Ubuntu. It is currently a command-line tool. This program was not designed to work on the CCC. The design and program are modular in nature and make maximum use of abstract data types. One of the key aspects of this was the structure employed for keeping track of the users: a clean tree structure was developed in order to quickly sort, change, and interact with the users. Particular attention is paid to the underlying foundation of the server and the user tree structure, allowing the program to scale. Careful consideration went into interrupts and users exiting the program in various states. The report outlines the test cases used to verify the correct operation of the program, as well as the source in the appendix.

2 Project Description

Accomplishing the goal of multiple clients interacting in chat room relied heavily on careful use of threads. The server has a commissioner thread, a thread that handles the connection, and a thread that handles the server's GUI. The clients have readers and writers. These threads worked closely together to provide seamless service for the users of this program.

3 Detailed Design

3.1 Server

The initial portion of the server's source code sets the structure necessary for the socket programming of the project. It creates a socket, and binds it to a port - a number specified by the host of the server executable. Generally, any number between 1025 and 65535 will work. Then the server proceeds to initialize a semaphore to control the number of users allowed into the chat room. This constant is determined at compile time. The typical signal for a keyboard interrupt is overridden, allowing the server to gracefully handle the interrupt and carefully clean up an extra threads and memory. The rest of the functionality of the server is bundled into threads specified below.

3.1.1 Interactive

This thread allows the host of the server to administer the program. It provides most of the same commands as the command line interface the chat room clients have access too, but it also provides additional error messages and notifications. Entering "help" on either CLI (command line interface) provides the user with helpful information on how to user the program.

3.1.2 Commissioner

The original process of the server acts as the commissioner. It handles requests to join the server. Once it has validated a new client, it creates a new thread - connection - to handle that specific user. The threads are kept track in an array the reallocates additional memory as more users join the chat room.

3.1.3 Connection

The connection thread caters to a specific user. It continually waits for a prompt from the user, and then parses the users request. If the user uses one of the available commands, the connection thread will handle that specific case. Otherwise, the user wishes to message the chat room, and the connection thread sends the message to all the other users, and then returns the message to the sender - with a prefixed "Me: ", both as a confirmation to the sender and to keep the chat readable to the user.

3.2 Client

Similar to the server, the client begins by setting up the socket programming necessary to form the connection with the server. It attempts to connect to the server as determined by the command-line arguments given to the client executable, a host-name and a port number. When running the program locally, set the host-name to be 'localhost' and the port number to be whatever number one ran the server executable with. The client validates the name, and if the name is already taken, the program will exit. The rest of the functionality of the client is bundled into the reader and writer threads.

3.2.1 Reader

The reader thread continually handles input from the server. Generally speaking this will be messages from other users, confirmations and notifications, and responds to specific commands that the user inputted. For example, the list command is sent to the server via the writer thread, and the reader thread will display the list that the server put together. The complications in the reader thread arise in its handling of the server exiting, and for the changing of its user's nickname. The parsing for notification of username change is not a part of the project that I am proud of. While it is technically impossible to construct a string that would change the name of another user, there is not a lot of protection against it. This weakness, among others, is the focus of future endeavors to increase the robustness and security of the chat.

3.2.2 Writer

The writer thread continually handles input from the client and runs a minimal parsing function on it before sending the request to the server to handle it. If the request is a specific command, the writer thread in some cases specifically handles the execution of these requests. This includes commands like exit and change which respectively allow the user to leave the chat or change their nickname. The writer parses the user's input and then formats it, adding additional information, the user's name, and sends it to the server. Particular attention was put on freeing any allocated buffers in order to maintain a high-speed and efficient service.

3.3 Tree

A tree structure was developed to cleanly organize all users in the chat room. It supports fast searching for users, removing users, and adding users. Its basic implementation is below. It is a binary tree, that is not self-balancing.

3.4 Specific User Requests

3.4.1 Help

A CLI command that informs the user on how to use the chat service. Its output:

```
|> help
|-----------------------------------------
| Hi Eric! Welcome to the Chat.
| info / help / h         = printUsage
| quit / exit / leave / q = ends service
| list / l / users        = shows all clients
| "message"              = sends to everyone in room
| change NEW_NAME         = attempts to change username
| whisper USER "message" = to send secret messages
|-----------------------------------------
```

3.4.2 Change

In addition to allowing a user to choose a nickname when they first join the server, a user may change their name at any time. If the name is already taken, they will keep their original name.

```
|> change terror
| Name Change Successful. Your name is now: terror
|>
```

3.4.3 Whisper

Using the whisper command users can directly message each other without other people seeing.

```
|> whisper Rob Look at this kid try to code
| Whisper to Rob successful...
|>
```

It looks like this when you receive a whisper:

```
|>
| Whisper from Hero: What a joke!
|>
```

Or you can give yourself a whisper.

```
|> whisper Jim yo, nice work back there.
```

```
| Note to self: yo, nice work back there.
|>
```

3.4.4 Exit

The exit command will cleanly remove you from the chat service. It informs the server, which drops the connection, and lets the other users know you left.

```
|> exit
| exiting now!
|_______________________________________
```

It looks like this when a user leaves:

```
|>
| terror has left Chat.
|>
```

3.4.5 List

The command list will inform you of all active users, including yourself.

```
|> list
| Active Users: Jim Hero
|>
```

3.5 Closing sockets and Canceling Threads

In order to be stable over an extended period of time, the client and server programs deallocate any memory requested. Furthermore, on the client-side, the socket is closed when the client exits. This will happen regardless if the user used a keyboard exception to end the program, the CLI command, or if the server exits. On the server-side, when a client informs the server that is leaving the service, it closes the connection, and deletes the user from its list of users. It frees the memory used to store the user structure. The thread then exits. If the server ends, then it will close all the sockets and terminate all the threads, and then free all the memory. A current hole in this is that

the additional threads in the client aren't specifically closed when the client exits. This is because there was some issues canceling a thread that was currently executing before the other threads, and then hanging. This can be worked around by creating a clean-up thread that will close all other threads, and then exit. Its not as trivial as closing the threads via the original process because the reader and writer block waiting for input. However, of the two possible sides of the client server structure for the managing of threads and memory not to be perfect, its far better for the client to have this small hiccup of not manually closing the threads when it exits, because the program exits and so ultimately there is no difference. Clearly it would be a bit better and cleaner to have implemented to this last bit, but the important part of handling the streams on the server side is working.

4 Testing and Validation

4.1 Foundation - Testdriver

The initial major testing efforts went into a program dubbed testdriver to test the foundations of the server and the supporting code. A major component of this was testing the tree structure, and ensuring it was correctly implemented. Other parts included testing certain methods of testing auxiliary utility functions, such as trim - which removed all white space from an input. The testdriver section has the source for these tests.

Another aspect of the testdriver program was the testing of actual functions from the server and client, but largely the testing of the server and client components was tested manually. This is one of the sore spots of the project: lots of time went into testing the code, but not a lot went into automating the testing of connections, etc. Given more time, this is something that I would have invested into as it would make the project more robust, and allow for more rapid development.

That being said, the manual testing focused on ensuring all the clients worked together well, ensuring the server could handle the respective requests, and that all functionality was solid with no exceptions. Examples of these tests are below.

4.2 Interrupts

One of the original major issues was users, or the server administrator, using keyboard escape sequences: they would crash the other clients and or server. To handle this I firstly switched to a send and recv over read and write, which allowed for the option to not send signals.

```
int i = send(sock, listOfUsers, strlen(listOfUsers), MSG_NOSIGNAL);
```

The MSG_NOSIGNAL option prevents this from happening. This stops the interrupts from being passed on from a client to other clients.
To handle the interrupt gracefully locally, within the client or server from which it originated, the interrupt had to be intercepted, and handled specifically.
The interception:

```
  /* handling signals */
  struct sigaction sa;
  sa.sa_handler = sigint_handler;
  sa.sa_flags = 0; // or SA_RESTART
  sigemptyset(&sa.sa_mask);

  if (sigaction(SIGINT, &sa, NULL) == -1)
  {
    perror("sigaction");
    exit(1);
  }
```

The handler (for the client):

```
void sigint_handler(int sig)
{
  printf("| Leaving Chat...\n");
  leaveServer();
  closeClientConnection();
}
```

A lot of testing went into this, ensuring that interrupts were successfully handled in different states. All of this was manually done, as I didn't spend the time figuring out how to script it. More methodical tests would definitely be a goal for future development: setting up a test suite, perhaps a bash

script, which sent input and somehow expected specific results would provide faster turn-around. (I assume there is indeed a way to write tests in C that generate the interrupts, so I could pursue this instead.)

5 Future Development

5.1 Testing

Further testing would be a good first step. A lot of details in where I would go is in the testing sections.

5.2 Security

5.2.1 Validation

Validating the usernames and messages to a greater extent would help make the program more robust. Adding a password for clients would further help with validating the users requests.

5.2.2 Encryption

It would be interesting to add encryption to the messages, and investigate different strategies available.

5.3 Scalability

5.3.1 Standardization

In order to help be able to scale this into a larger project, I would standardize the formation of messages and requests. Currently there is a pattern, but its not perfect. Making this more robust would help prevent interference. This will also help with security, less chance for holes in the project. An idea for this is building some sort of simple JSON-like structure that holds the information necessary for a request. This would lead to tighter code and a bit less repetition. An idea for what this could look like is:

```
message :
{
   sender: Rob,
```

```
    sock: 4,
    type: CHAT_M_,
    message: "Do you have time on Friday?",
    args: []
}
```

5.3.2 red-black trees

Upgrade from a binary tree to a red-black tree structure in order to better handle operations on the user structure. This would have very little affect unless this project were to handle large numbers of users that continually joined or late. A better data structure would also be useful if the design of the chat changed and users could have permanent accounts. However, at this point it would be better to transition into using a more formal database structure.

5.4 Features

Creating separate rooms that clients could join and create would be useful. Supporting this with a list command for rooms, passwords, and hidden rooms would be useful.

5.5 Visual Features

This definitely needs improvement and is something to focus on. There are a lot of different directions that this can be taken.

5.5.1 UI

Using a library to build a more sophisticated UI, with a specific line for entering commands, and a scroll-able window with the chat in it would be very nice.

5.5.2 Modularity of Parsing

Allowing an even wider range of possibilities to work as commands would be user-friendly.

5.5.3 Auto-completion

This is always a nice thing to have. It would be fairly straight forward to implement once I have new method of taking in input from t he user. I could keep an updated list of the users and auto-complete their names on tabs when their name is uniquely spelled. I could add to this list for this specific purpose the terms and special commands.

5.5.4 Colors

Color highlighting for specific regions, messages, user names, announcements, etc.

5.6 Other Methods

To some degree using multiple processes could be more difficult but could be better: research into this could help performance. Perhaps using multiple sockets for the same user to more securely define actions would increase security and the exactness of the service. Redesigning the thread structure to test having less reader threads may overall increase performance, as the number of users increase, the overhead of the threads will outweigh their user.

6 Conclusion – Solution Summary

Overall, this was a successful venture into socket programming. From having zero experience, I now have some idea on how to put together a network in a low level setting, and how to approach the challenges that arise in such an environment. Although there is always room for improvement, I think I successfully focused on making the project robust. It handles users joining, users leaving and all such scenarios cleanly. Some of the issues that remain are largely UI related. Focusing upon building a more sophisticated UI and a deeper set of tests would be a well-advised approach to improving this work.

7 Appendices

7.1 References

I referred to a guide on how to do socket programming, an outline on handling interrupts and a couple stack overflow posts on how to trim a string and how to define a compare that disregards case. Largely, key parts of the guide of the basic outline of the client server socket programming got directly incorporated into my code-base; firstly because I was using it to learn, and secondly because nearly everything was necessary to make it work at all - and I saw the exact same code in a textbook. The guide on interrupts was more of an API then example code. The trim and strcmpc are largely unchanged from how I found them and I make no claims on their authorship. The links are below. The beej guide is really good, I plan spending more time reading through it.

- http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html
- https://www.cs.cf.ac.uk/Dave/C/node24.html
- http://stackoverflow.com/questions/122616/how-do-i-trim-leading-trailing-whitespace-in-a-standard-way
- http://cboard.cprogramming.com/c-programming/63091-strcmp-without-case-sensitivity.html

7.2 Test Cases and Results

Here is a streamlined, but comprehensive, set of tests to fully test all functionality and error handling. Some specifics of user commands are above.

7.2.1 Test I: Basic functionality of the server

Basic functionality of the server. Ensuring that the server can run and the CLI functions. Notice that entering any white space simply causes the CLI to prompt the user again. This holds for both the client and the server.

```
> ./server 123123
-->
```

```
--> help
---------------------------------------------
This is the Server for the Chat.
info / help / h        = printUsage
quit / exit / leave / q = ends service
list / l / userse      = shows all users
---------------------------------------------
-->
```

7.2.2 Test II: Basic connection of a client to the server

Basic connection of a client to the server. The server validates the name and then adds the new user to its tree structure of other users. The client attempts to join, and when it succeeds, it is informed that it has joined, and that his arrival has been announced to the other users.

```
> ./client localhost 123123
 ----------------------------------------
| Enter user name: Josh
| Hi Josh! Welcome to that chat!
| Others in the chat see you joined!
|>
```

The CLI side of the user joining:

```
--> New user Josh attempting to join...
User Josh valid and added!
-->
```

7.2.3 Test III: Basic commands for a client

Basic commands for a client. Although the results of these are detailed above in the specific user request section.

```
|> help
|---------------------------------------------
| Hi Josh! Welcome to the Chat.
| info / help / h         = printUsage
| quit / exit / leave / q = ends service
```

```
| list / l / users        = shows all clients
| "message"              = sends to everyone in room
| change NEW_NAME         = attempts to change username
| whisper/w USER "message" = to send secret messages
|-----------------------------------------
|> l
| Active Users: Josh
|> whisper Josh I am the best
Note to self: I am the best
|>
```

7.2.4 Test IV: Concurrency - Testing multiple clients

Testing multiple clients interacting on the same server.
User 'BLUE' joining, followed by 'RED' and then 'GREEN':

```
| Others in the chat see you joined!
|>
| New User: RED is now in the room!
| New User: GREEN is now in the room!
```

Different users communicating:

```
|> hi
| Me: hi
|>
| RED: yo
| GREEN: shhh stop, im trying to think
```

Whispering from RED to GREEN, and BLUE not being aware:
The original whisper:

```
| GREEN: shhh stop, im trying to think
whisper GREEN blue is so bad
Whisper to GREEN successful...
|>
| GREEN: haha yeah
```

The reception and response:

```
Whisper from RED: blue is so bad
haha yeah
| Me: haha yeah
|>
```

The oblivious bystander:

```
|>
| GREEN: haha yeah
```

The server CLI notifications of users joining:

```
--> New user BLUE attempting to join...
User BLUE valid and added!
New user RED attempting to join...
User RED valid and added!
New user GREEN attempting to join...
User GREEN valid and added!
```

7.2.5 Test V: Testing a client leaving via the CLI command

User 'BLUE' leaving via CLI:

```
|> ok im out
| Me: ok im out
|> exit
| exiting now!
|_______________________________________
```

User 'RED' is informed:

```
| BLUE: ok im out
| BLUE has left Chat.
```

Server CLI information on client activities:

```
User is BLUE leaving.
```

7.2.6 Test VI: Testing a client leaving via the keyboard interrupt

User 'GREEN' leaving via a keyboard interrupt:

```
|> X^
| Leaving Chat...
>
```

User 'RED' is informed:

```
| GREEN has left Chat.
```

Server CLI information on client activities:

```
User is GREEN leaving.
```

7.2.7 Test VII: Testing Server Termination

Testing the server ending, and ensuring clients handle this gracefully.
The server exiting:

```
--> exit
Ending service...
Freeing users...
Canceling threads!
Canceling thread 0...
Canceling thread 1...
Canceling thread 2...
Threads canceled.
Done
>
```

User 'RED' is informed and leaves gracefully:

```
| Server went down, exiting...
|_______________________________________
```

7.2.8 Test VIII: Updating user list after user connection / disconnection

```
 ----------------------------------------
| Enter user name: Rick
| Hi Rick! Welcome to that chat!
| Others in the chat see you joined!
|>
| New User: Bob is now in the room!
| l
| Active Users: Rick Bob
|>
| New User: James is now in the room!
|> l
| Active Users: Rick Bob James
|>
| Bob has left Chat.
| l
| Active Users: Rick James
|> exit
| exiting now!
|----------------------------------------
```

7.2.9 Test IX: Whispering to a Non-Existing Client fails gracefully

```
|> whisper God Please
| User not found
|>
```

7.2.10 Test X: Connecting to a non-existent server exits properly.

```
> ./client localhost 42
ERROR connecting: Connection refused
>
```

7.3 Code

All working code is view-able in full on github. Below is a static reference.

7.3.1 Client.c

```
#include "client.h"

void* writer(void* user);
void* reader(void* user);
int verify(char* username);
void printUsage(void);
void closeClientConnection(void);
char* formMessage(char* msg);
void leaveServer();

int sockfd;
char* username;
static volatile int running = 1;
void sigint_handler(int sig);
pthread_t threads[2];


int main(int argc, char *argv[])
{
  int portno, n;
  struct sockaddr_in serv_addr;
  struct hostent *server;

  char buffer[256];
  if (argc < 3)
  {
    fprintf(stderr,"usage %s hostname port\n", argv[0]);
    exit(0);
  }
  portno = atoi(argv[2]);
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
    error("ERROR opening socket");
  server = gethostbyname(argv[1]);
```

```
if (server == NULL)
{
  fprintf(stderr,"ERROR, no such host\n");
  exit(0);
}

bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
bcopy((char *)server->h_addr,
 (char *)&serv_addr.sin_addr.s_addr,
 server->h_length);
serv_addr.sin_port = htons(portno);
if (connect(sockfd, (struct sockaddr *) &serv_addr,
    sizeof(serv_addr)) < 0)
  error("ERROR connecting");

/* handling interrupts */
struct sigaction sa;

sa.sa_handler = sigint_handler;
sa.sa_flags = 0; // or SA_RESTART
sigemptyset(&sa.sa_mask);

if ( sigaction(SIGINT, &sa, NULL) == -1 )
{
  perror("sigaction");
  exit(1);
}

/*  verification */
username = calloc(sizeof(char), 20);
printf(" ________________________________________\n");
printf("| Enter user name: ");
scanf("%s", username);

if ( !verify(username) )
{
  printf("INVALID USER\n| exiting now...");
  printf("|________________________________________\n");
```

```
    running = 0;
    exit(0);
  }
  else
  {
    printf("| Hi %s! Welcome to that chat!\n", username);
    fflush(stdout);
  }


  if( pthread_create( &threads[0], NULL, writer, (void *)0 ) != 0 )
  {
    perror("pthread_create");
    exit(1);
  }

  if( pthread_create( &threads[1], NULL, reader, (void*)0 ) != 0 )
  {
    perror("pthread_create");
    exit(1);
  }

  while(running){}//without this, everythings dies!
  close(sockfd);
  return 0;
}

void* writer(void* null)
{
  int n;
  int first = 0;

  while(running)
  {
    char* buffer = calloc(sizeof(char), 256);
    char* toBeFreed = buffer;

    if (first) printf("|> ");
    else first = 1;
    bzero(buffer,256);
```

```
  fgets(buffer,255,stdin);

  buffer = trim(buffer);

  if(strlen(buffer) == 0)
    continue;

  else if (strcmpc(trim(buffer), "exit") == 0 ||
          strcmpc(trim(buffer), "quit") == 0 ||
          strcmpc(trim(buffer), "leave") == 0 ||
          strcmpc(trim(buffer), "q")  == 0)
  {
    printf("| exiting now!\n");
    printf("|_____\n");
    running = 0;

    continue;
  }
  else if (strcmpc(trim(buffer), "help") == 0 ||
          strcmpc(trim(buffer), "info") == 0 ||
          strcmpc(trim(buffer), "h") == 0)
  {
    printUsage();
    continue;
  }
  else if (strcmpc(trim(buffer), "list") == 0 ||
      strcmpc(trim(buffer), "users") == 0 ||
          strcmpc(trim(buffer), "l") == 0)
  {
    n = send(sockfd,"list",strlen("list"), MSG_NOSIGNAL);
    if (n < 0)
 error("ERROR writing to socket");

    continue;
  }

  char* newName = calloc(sizeof(char), 21);
  char* garb = calloc(sizeof(char), 21);
  int c = sscanf(buffer, "%s %s\n", garb, newName);
```

```
    if(c == 2 && strcmp(garb, "change") == 0 )
    {
      changeName(newName);
      free(newName);
      free(garb);
      continue;
    }
    else
    {
      free(newName);
      free(garb);
    }

    //normal chat message
    char* bufferMessage = calloc(sizeof(char), (20 + 255));
    strcat(bufferMessage, username);
    strcat(bufferMessage, " ");
    strcat(bufferMessage, buffer);

    n = send(sockfd,bufferMessage,strlen(bufferMessage),
        MSG_NOSIGNAL);

    if (n < 0)
      error("ERROR writing to socket");

    free(bufferMessage);
    free(toBeFreed);
  }

  //handle exit
  leaveServer();
  closeClientConnection();
}

/**
 * handles leaving the server
 */
void leaveServer()
{
  char* leave = calloc(sizeof(char), (20 + 10));
```

```
  strncat(leave, "_EXIT_ ", sizeof("_EXIT_ "));
  strcat(leave, username);
  int n = send(sockfd, leave, strlen(leave), MSG_NOSIGNAL);
  if (n < 0)
    error("ERROR writing to socket");
  free(leave);
}

/**
 * the reader thread
 */
void* reader(void* null)
{
  char buffer[256];
  int n;

  while(running)
  {
    bzero(buffer, 256);
    n = recv(sockfd,buffer,255,0);
    if (n < 0)
      error("ERROR reading from socket");

    char* leave = "_SERVER_EXIT_";
    if ( strncmp(buffer, leave, strlen(leave)) == 0 )
    {
      printf("\n| Server went down, exiting...\n");
      printf("|_____________________________________\n");
      closeClientConnection();
    }

    char* newName = calloc(sizeof(char), 20);
    int c = sscanf(buffer, "| Name Change Successful. Your name is
       now: %s\n", newName);
    if(c > 0)
    {
      free(username);
      username = calloc(sizeof(char), 20);
      strcat(username, newName);
    }
```

```
    free(newName);
    printf("%s\n",buffer);
  }
}

/**
 * requests the server to change your name
 */
void changeName(char* newName)
{

  char* leave = calloc(sizeof(char), 255);
  strcat(leave, username);
  strncat(leave, " change ", sizeof(" change "));
  strcat(leave, newName);

  int n = send(sockfd, leave, strlen(leave), MSG_NOSIGNAL);
  if (n < 0)
    error("ERROR writing to socket");

  //wait for confirmation from server that its a valid name
  free(leave);
}

/**
 * verifies that the name is ok
 */
int verify( char* username )
{
  //write name to server
  int n = send(sockfd,username,strlen(username), MSG_NOSIGNAL);

  if (n < 0)
    error("ERROR writing to socket");

  //wait for confirmation from server that its a valid name
  int valid;
  n = recv(sockfd , &valid, sizeof(int), 0);
  if (n < 0)
    error("ERROR reading from socket");
```

```
  return valid;
}

/**
 * interrupt handler
 */
void sigint_handler(int sig)
{
  printf("| Leaving Chat...\n");
  leaveServer();
  closeClientConnection();
}

void closeClientConnection(void)
{
  close(sockfd);
  exit(0);
}

void printUsage(void)
{
  printf("|------------------------------------------\n");
  printf("| Hi %s! Welcome to the Chat.\n", username);
  printf("| info / help / h        = printUsage\n");
  printf("| quit / exit / leave / q = ends service\n");
  printf("| list / l / users       = shows all clients\n");
  printf("| \"message\"             = sends to everyone in
     room\n");
  printf("| change NEW_NAME        = attempts to change
     username\n");
  printf("| whisper/w USER \"message\" = to send secret
     messages\n");
  printf("|------------------------------------------\n");
}
```

7.3.2 Server.c

```
#include "server.h"
```

```
int main(int argc, char *argv[])
{

  /* building socket */
  int sockfd, newsockfd, portno, pid;
  socklen_t clilen;
  struct sockaddr_in serv_addr, cli_addr;

  if (argc < 2)
  {
    fprintf(stderr,"ERROR, no port provided\n");
    exit(1);
  }

  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
    error("ERROR opening socket");
  bzero((char *) &serv_addr, sizeof(serv_addr));
  portno = atoi(argv[1]);
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(portno);
  if ( bind(sockfd, (struct sockaddr *) &serv_addr,
      sizeof(serv_addr)) < 0 )
    error("ERROR on binding");
  listen(sockfd,5);
  clilen = sizeof(cli_addr);

  /* init the semaphore for active connections. */
  if ( sem_init(&active_connections, 0, MAX_CLIENTS) < 0 )
  {
    perror("sem_init");
    exit(1);
  }

  /* handling signals */
  struct sigaction sa;
  sa.sa_handler = sigint_handler;
  sa.sa_flags = 0; // or SA_RESTART
```

```
sigemptyset(&sa.sa_mask);

if (sigaction(SIGINT, &sa, NULL) == -1)
{
  perror("sigaction");
  exit(1);
}

pthread_t interactive_thread;
/* create an interactive cli for server */
if ( pthread_create( &interactive_thread, NULL, interactive,
    (void*)(intptr_t)0 ) != 0 )
{
  perror("pthread_create");
  exit(1);
}


/* create structure for keeping track of threads */
threads = malloc(sizeof(pthread_t) * thread_size);

/* create structure for keeping track of users */
users = newTree();

/* core loop handling connecting clients */
/* thus, the original thread is the commissioner */
while (running)
{
  sem_wait( &active_connections );
  //block on there being room to connect, used to limit number of
      clients.
  //this isn't needed, but its a clean way of limiting the users,
      and keeping the
  //performance high. (this is an exclusive chat room.)
  /* init the semaphore for active connections. */
  if ( sem_init(&active_connections, 0, MAX_CLIENTS) < 0 )
  {
    perror("sem_init");
    exit(1);
  }
```

```
/* block on new client */
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
    &clilen);

if (newsockfd < 0)
  error("ERROR on accept");

pthread_t current_thread;

// get user name and validate!
char* userbuffer = malloc(sizeof(char) * 256);//[256];
bzero(userbuffer,256);
int n = read(newsockfd, userbuffer, 255);
if (n < 0) error("ERROR reading from socket");

printf("| New user %s attempting to join...", userbuffer);

//determine if user is unique
int valid;
if ( valid = validate(users, userbuffer) )
{
  //valid = 1
  int z = write( newsockfd, &valid, sizeof(int));
  if (z < 0) error("ERROR writing to socket");

  //create and add user to tree stucture
  addUser( users, userbuffer, newsockfd );

  //user valid and added
  printf("User %s valid and added!\n", userbuffer);

  //inform the people
  addUser(users, userbuffer, newsockfd);

  char* newUser = calloc(sizeof(char), 40);
  char* others = calloc(sizeof(char), 155);

  strcat(newUser, "| Others in the chat see you joined!");
  strcat(others, "\n| New User: ");
  strcat(others, userbuffer);
```

```
    strcat(others, " is now in the room!");

    sendAllBut(others, newUser, userbuffer);

    free(others);
    free(newUser);
  }
  else
  {
    //valid = 0;
    free(userbuffer);
    sem_post(&active_connections); //release spot in the queue
    close(newsockfd);
    printf("User %s is invalid and connection was dropped.\n",
        userbuffer);
    continue;
    //continue;
  }

  if( pthread_create( &current_thread, NULL, connection,
      (void*)(intptr_t)newsockfd ) != 0 )
  {
    perror("pthread_create");
    exit(1);
  }

    //store thread in resizing array - gonna keep it simple.
    //threads can close themselves properly, but if the server is
        the one that exits
    //going to manually close all threads

    if (thread_counter < thread_size)
    {
threads[thread_counter++] = current_thread;
    }
    else
    {
thread_size *= 2;
threads = realloc(threads, thread_size * sizeof(pthread_t));
threads[thread_counter++] = current_thread;
```

```
    }

  } /* end of while */

  return 0;
}


/**
 * This function will become a connection thread!
 * It will service the client, and handle the connection.
 * It will gracefully handle the end of the connection!
 */
void* connection (void *sock_void)
{
  int sock = (intptr_t)sock_void;
  while ( prompt ( sock ) ) {}
  close(sock); //maybe don't want to do this.
  sem_post(&active_connections); //release spot in the queue
}

/**
 * prompt :: int -> int
 * handles the prompting of clients, and their requests
 */
int prompt(int sock)
{
  int n;
  char* buffer = malloc(sizeof(char) * 255);
  bzero(buffer,256);
  n = read(sock, buffer, 255);

  if (n < 0)
    error("ERROR reading from socket");
  buffer = trim( buffer );
  int value = parse( buffer, sock ); //0, stop. 1, continue.
  free(buffer);

  return value;
}
```

```
/**
 * prompt :: int -> int
 * parses messages from clients and appropriately handles them
 */
int parse (char* message, int sock)
{
  int n;
  char* leave = "_EXIT_";
  char* saveptr1;
  char* parsed[100] = {0};
  char* inputMessages = strtok_r(message, " ", &saveptr1);
  int i = 0;

  while (inputMessages != NULL)
  {
    parsed[i] = inputMessages;
    i++;
    inputMessages = strtok_r(NULL, " \r\t\n", &saveptr1);
  }

  if (parsed[0] == NULL) return 1;

  if ( strncmp(parsed[0], leave, strlen(leave)) == 0)
  {
    char* name = parsed[1];

    printf("User is %s leaving.\n", name);

    //remove him from users
    User* u = findUser(users, name);
    if (u == NULL)
    {
      printf("user not found");
      return 0;
    }

   //closing the connection!
    close(u->socket);
```

```
  //deleting user from tree
  deleteUser(users, name);


  //close the connection
  //let all the users know he's leaving
  char* userLeaving = calloc(sizeof(char), 150);
  if (userLeaving == NULL) error("malloc failed");

  strcat(userLeaving, "\n| ");
  strcat(userLeaving, name);
  strcat(userLeaving, " has left Chat.");

  sendAll(userLeaving);
  free(userLeaving);

  return 0;
}
else if ( strncmp(parsed[0], "list", strlen("list")) == 0 ||
     strncmp(parsed[0], "l", strlen("l")) == 0 )
{
  char* listOfUsers = calloc(sizeof(char*), 10 + 22 *
      MAX_CLIENTS);
  populateList(listOfUsers);
  int i = send(sock, listOfUsers, strlen(listOfUsers),
      MSG_NOSIGNAL);
  if (i < 0)
    error("ERROR reading from socket");
}
else if ( strncmp(parsed[1], "change", strlen("change")) == 0)
{
  char* newName = parsed[2];
  int valid;

  if (findUser(users, newName) != NULL)
  {
    int i = send(sock, "This name is taken.", strlen("This name
        is taken."), MSG_NOSIGNAL);
    if (i < 0)
 error("ERROR reading from socket");
```

```
    return 1;
  }

  deleteUser(users, parsed[0]);
  addUser(users, newName, sock);

  char* msgMe = calloc(sizeof(char), 124);
  char* informOthers = calloc(sizeof(char), 124);

  strcat(msgMe, "| Name Change Successful. Your name is now: ");
  strcat(msgMe, newName);
  strcat(informOthers, "| Name Change: ");
  strcat(informOthers, parsed[0]);
  strcat(informOthers, " is now ");
  strcat(informOthers, newName);

  sendAllBut(informOthers, msgMe, newName);

  free(msgMe);
  free(informOthers);
  //inform other users of name change

  //inform success

}
else if ( strncmp(parsed[1], "whisper", strlen("whisper")) == 0)
{
  //create message
  char* secret = calloc(sizeof(char), 255+25);
  char* from  = parsed[0];
  char* to    = parsed[2];

  //error checking
  if (secret == NULL || to == NULL)
  {
    char* errorMsg = "| malformed whisper request.";
    send(sock, errorMsg, strlen(errorMsg), MSG_NOSIGNAL);
    return 1;
  }
```

```
 //whisper to yourself?
 if (strcmp(to, from) == 0)
 {
   char* confirmation = calloc(sizeof(char), 100);
   strcat(confirmation, "| Note to self: ");

   int index = 3;
   while(parsed[index]!=NULL&&strlen(trim(parsed[index]))!=0)
   {
strcat(confirmation, parsed[index++]);
strcat(confirmation, " ");
   }

   send(sock, confirmation, strlen(confirmation), MSG_NOSIGNAL);

   free(confirmation);
   return 1;
 }

 User* u = findUser(users, to);

 if (u != NULL)
 {
   //build message for TO user
   strcat(secret, "\nWhisper from ");
   strcat(secret, from);
   strcat(secret, ": ");

   int index = 3;
   while(parsed[index]!=NULL&&strlen(trim(parsed[index]))!=0)
   {
strcat(secret, parsed[index++]);
strcat(secret, " ");
   }

   //send the whisper
   int j = send(u->socket, secret, strlen(secret), MSG_NOSIGNAL);
   if (j < 0)
error("ERROR reading from socket");
```

```
    //build confirmation message for FROM user
    char* confirmation = calloc(sizeof(char), 40);
    strcat(confirmation, "Whisper to ");
    strcat(confirmation, to);
    strcat(confirmation, " successful...");

    //send confirmation
    send(sock, confirmation, strlen(confirmation), MSG_NOSIGNAL);
    free(confirmation);
  }
  else
  {
    //notify the FROM user that the TO user could not be found
    int j = send(sock, "User not found", strlen("User not
        found"), MSG_NOSIGNAL);
    if (j < 0)
 error("ERROR reading from socket");
  }
  free(secret);
}
else
{
  //build chat room messages
  char* reconstruct = calloc(sizeof(char), (20 + 255));
  char* reconstructBut = calloc(sizeof(char), (4 + 255));
  strcat(reconstruct, "\n");
  strcat(reconstructBut, "| Me: ");

  strcat(reconstruct, "| ");
  strcat(reconstruct, parsed[0]);
  strcat(reconstruct, ": ");

  int i = 1;
  while(parsed[i]!=NULL&&strlen(trim(parsed[i]))!=0)
  {
    strcat(reconstruct, parsed[i]);
    strcat(reconstruct, " ");
    strcat(reconstructBut, parsed[i++]);
    strcat(reconstructBut, " ");
```

```
    }

    sendAllBut(reconstruct, reconstructBut, parsed[0]);

    free(reconstruct);
    free(reconstructBut);
  }

  return 1;
}

/**
 * interactive :: void* -> void*
 * interactive thread for server admin to inspect chat
 */
void* interactive(void *null)
{
  char buffer[256];

  while(running)
  {
    printf("--> ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    if((buffer[0] == '\n')|(strlen(buffer) == 0))
      continue;
    else if (strcmpc(trim(buffer), "exit") == 0 ||
        strcmpc(trim(buffer), "quit") == 0 ||
        strcmpc(trim(buffer), "leave") == 0 ||
        strcmpc(trim(buffer), "q")  == 0)
    {
      destroyServer();
      //continue;
    }
    else if (strcmpc(trim(buffer), "help") == 0 ||
        strcmpc(trim(buffer), "info") == 0 ||
        strcmpc(trim(buffer), "h") == 0)
    {
      printUsage();
    }
```

```
    else if (strcmpc(trim(buffer), "list") == 0 ||
        strcmpc(trim(buffer), "l") == 0 ||
        strcmpc(trim(buffer), "users") == 0)
    {
      printf("Active Users: ");
      displayTree(users);
      printf("\n");
    }
  }
}

/**
 * sigint_handler :: int -> void
 * replaces normal keyboard interrupt to gracefully end server
 */
void sigint_handler(int sig)
{
  destroyServer();
}

/**
 * destroyServer :: -> void
 * dismantles chat service, canceling all threads and freeing all
    data
 */
void destroyServer(void)
{
  printf("Ending service...\n");

  //tell users to leave
  sendAll("_SERVER_EXIT_");

  //free users
  printf("Freeing users...\n");
  deleteTree(users);

  //end threads
  printf("Canceling threads!\n");
  int i = 0;
  while( i < thread_counter )
```

```
  {
    printf("Canceling thread %d...\n", i);
    pthread_cancel(threads[i++]);
  }

  printf("Threads canceled.\n");

  printf("Done\n");
  exit(0);
}

void printUsage(void)
{
  printf("----------------------------------------------\n");
  printf("This is the Server for the Chat.\n");
  printf("info / help / h       = printUsage\n");
  printf("quit / exit / leave / q = ends service\n");
  printf("list / l / userse     = shows all users\n");
  printf("----------------------------------------------\n");
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
// tree functionality functions that seemed to belong more here
    than in tree.c //

void sendAll(char* msg)
{
  if(users != NULL) sendAllHelper(msg, users->root);
}

void sendAllHelper(char* msg, Node* n)
{
  if (n == NULL) return;

  int socket = n->data->socket;

  int i = send(socket, msg, strlen(msg), MSG_NOSIGNAL);
  if (i < 0) error("ERROR reading from socket");

  sendAllHelper(msg, n->left);
```

```
  sendAllHelper(msg, n->right);
}

void sendAllBut(char* msg1, char* msg2, char* name)
{
  if(users != NULL) sendAllHelperBut(msg1, msg2, name, users->root);
}

void sendAllHelperBut(char* msg1, char* msg2, char* name, Node* n)
{
  if (n == NULL) return;

  int socket = n->data->socket;
  int i;

  if (strcmp(n->data->id, name)==0)
    i = send(socket, msg2, strlen(msg2), MSG_NOSIGNAL);
  else
    i = send(socket, msg1, strlen(msg1), MSG_NOSIGNAL);

  if (i < 0) error("ERROR writing to socket");

  sendAllHelperBut(msg1, msg2, name, n->left);
  sendAllHelperBut(msg1, msg2, name, n->right);
}

void populateList(char* list)
{
  if (users == NULL) return;
  strcat(list, "| Active Users: ");
  populateListHelper(list, users->root);
}

void populateListHelper(char* list, Node* n)
{
  if (n == NULL) return;

  strcat(list, n->data->id);
  strcat(list, " ");
```

```
  populateListHelper(list, n->left);
  populateListHelper(list, n->right);
}
```

7.3.3 Tree.c

```
#include "tree.h"
#define DEBUG (0)

int validate(Tree* t, char* username)
{
#if DEBUG
  displayTree(t);
#endif
  return validateHelper(t->root, username);
}

int validateHelper(Node* current, char* data)
{
#if DEBUG
  printf("%s %s\n", "Validating user: ", data);
#endif
  if ( current == NULL )
  {
    return 1;//valid
  }

  int c = strcmp( current->data->id, data );
  if      ( c > 0 ) return validateHelper( current->left, data );
     //current is greater, go left
  else if ( c < 0 ) return validateHelper( current->right, data );
     //current is smaller, go right


  return 0; //invalid
}

void displayTree(Tree* t)
{
```

```
  if (t->root == NULL)
  {
    printf("<none>");
    return;
  }

  displayTreeHelper(t->root);
}

void displayTreeHelper(Node* current)
{
  if (current == NULL) return;
  displayTreeHelper(current->left);
  printf("%s ", current->data->id );
  displayTreeHelper(current->right);
}

int compare(User* a, User* b)
{
  return strcmp(a->id, b->id);
}

Tree* newTree()
{
  Tree* newTree = (Tree*)calloc(sizeof(Tree), 1);

  if ( newTree == NULL )
  {
    printf("%s\n", "Failed to malloc");
    exit(1);
  }

#if DEBUG
  printf("%s\n", "New user tree created.");
#endif

  return newTree;
}

Node* newNode(User* data)
```

```
{
  Node* new = (Node*)malloc(sizeof(Node));
  if ( new == NULL )
  {
    printf("%s\n", "Failed to malloc");
    exit(1);
  }

  new->data = data;
  new->right = NULL;
  new->left = NULL;

#if DEBUG
  printf("%s\n", "New node created.");
#endif

  return new;
}

User* newUser(char* name, int socket)
{
  User* newUser = (User*)malloc(sizeof(User));
  if ( newUser == NULL )
  {
    printf("%s\n", "Failed to malloc");
    exit(1);
  }

  newUser->id = calloc(sizeof(char), 20); strncpy(newUser->id,
     name, 20);
  newUser->socket = socket;

#if DEBUG
  printf("%s\n", "New user created.");
#endif

  return newUser;
}

Tree* addUser(Tree* t, char* id, int socket)
```

```
{
#if DEBUG
  printf("%s %s\n", "Adding user: ", id);
#endif

  User* u = newUser(id, socket);
  t->root = addUserHelper(t->root, u);
  return t;
}

Node* addUserHelper(Node* current, User* data)
{
  if ( current == NULL )
  {
    return newNode(data);
  }

  int c = compare( current->data, data );
  if      ( c > 0 ) current->left = addUserHelper( current->left,
      data ); //current is greater, go left
  else if ( c < 0 ) current->right = addUserHelper( current->right,
      data ); //current is smaller, go right
  //else //user already exists --> itll fail quietly, which is bad

  return current;
}

Node* removeUser(Tree* t, User* data)
{
  t->root = removeUserHelper(t->root, data);
  return t->root;
}

Node* removeUserHelper(Node* current, User* data)
{
  if ( current == NULL )
    return current; //not found

  int c = compare( current->data, data );
```

```
  if     ( c > 0 ) current->left = removeUserHelper(
      current->left, data ); //current is greater, go left
  else if ( c < 0 ) current->right = removeUserHelper(
      current->right, data ); //current is smaller, go right
  else //the target is found
  {
    if ( current->left == NULL )
    {
      Node *temp = current->right;
      deleteNode( current );
      return temp;
    }
    else if ( current->right == NULL)
    {
      Node *temp = current->left;
      deleteNode( current );
      return temp;
    }

    Node* temp    = min( current->right );
    current->data = temp->data;
    current->right = removeUserHelper( current->right, temp->data );
  }
  return current;
}

Node* deleteUser(Tree* t, char* data)
{
  t->root = deleteUserHelper(t->root, data);
  return t->root;
}

Node* deleteUserHelper(Node* current, char* data)
{
  if ( current == NULL )
    return current; //not found

  int c = strcmp( current->data->id, data );
  if     ( c > 0 ) current->left = deleteUserHelper(
      current->left, data ); //current is greater, go left
```

```
  else if ( c < 0 ) current->right = deleteUserHelper(
      current->right, data ); //current is smaller, go right
  else //the target is found
  {
    if ( current->left == NULL )
    {
      Node *temp = current->right;
      deleteNodeData( current );
      return temp;
    }
    else if ( current->right == NULL)
    {
      Node *temp = current->left;
      deleteNodeData( current );
      return temp;
    }

    Node* temp = min( current->right );

#if DEBUG
    if (temp != NULL) printf("min: %s", temp->data->id);
#endif

    current->data = temp->data;
    //we can't delete the string, because its needed!!
    current->right = removeUserHelper( current->right, temp->data );
  }
  return current;
}

Node* min(Node* current)
{
  if (current->left == NULL) return current;
  else min(current->left);
}

User* findUser(Tree* t, char* name)
{
  return findUserHelper(t->root, name);
}
```

```
User* findUserHelper(Node* current, char* data)
{
  if ( current == NULL )
    return NULL;

  int c = strcmp( current->data->id, data );
  if      ( c > 0 ) return findUserHelper( current->left, data );
     //current is greater, go left
  else if ( c < 0 ) return findUserHelper( current->right, data );
     //current is smaller, go right
  else return current->data; //user already exists
}

void deleteNode(Node* n)
{
  free( n );      //free node
}

void deleteNodeData(Node* n)
{
  free( n->data ); //free user
  free( n );      //free node
}

void deleteTree(Tree* t)
{
  deleteAll( t->root );
}

void deleteAll(Node* n)
{
  if ( n == NULL ) return;
  deleteAll( n->left );
  deleteAll( n->right );
  deleteNodeData( n );
}
```

7.3.4 TestDriver.c

```
#include "tree.h"
#include "utility.h"

int main()
{
  Tree* t = newTree();
  addUser(t, "james", 12);
  addUser(t, "ann", 12);
  addUser(t, "bob", 12);
  addUser(t, "robert", 12);
  addUser(t, "pop", 12);
  addUser(t, "teemo", 12);

  displayTree(t);
  User* r1 = findUser(t, "robert");
  User* r2 = findUser(t, "bob");
  User* r3 = findUser(t, "teemo");
  User* r4 = findUser(t, "sdfgdsfg");
  User* r5 = findUser(t, "pop");
  User* r6 = findUser(t, "james");

  removeUser(t, r6);
  displayTree(t);

  if (validate(t, "james")) printf("Correctly validated.\n");
  else printf("ERROR\n");

  if (!validate(t, "pop")) printf("Correctly did not validate.\n");
  else printf("ERROR\n");

  // testing
  Tree* t2 = newTree();
  addUser(t2, "Xeno", 12);
  addUser(t2, "Rock", 12);
  if (validate(t2, "james")) printf("Correctly validated.\n");
  else printf("ERROR\n");
```

```
  if (!validate(t2, "Xeno")) printf("Correctly did not
     validate.\n");
  else printf("ERROR\n");

  if (!validate(t2, "Rock")) printf("Correctly did not
     validate.\n");
  else printf("ERROR\n");

  //trim

  char* s1 = malloc(sizeof(char) * 20);//"hello";
  char* s2 = malloc(sizeof(char) * 20);//"hello";" hello\n ";
  strcpy(s1, "hello");
  strcpy(s2, "  hello\n ");

  if (strcmp(s1, trim( s2 )) == 0) printf("trim worked\n");
  else printf("trim did not work\n");

  //strcmpc
  char* s3 = "_EXIT_ user";
  char* s4 = "_EXIT";

  char* t1;

//more tests on delete user
  printf("testing removeUser\n");
  Tree* t4 = newTree();
  addUser(t4, "james", 12);
  addUser(t4, "ann", 12);
  addUser(t4, "bob", 12);
  addUser(t4, "robert", 12);
  addUser(t4, "pop", 12);
  addUser(t4, "teemo", 12);

  displayTree(t4);

  removeUser(t4, findUser(t4, "james"));
  removeUser(t4, findUser(t4, "ann"));
  removeUser(t4, findUser(t4, "bob"));
  removeUser(t4, findUser(t4, "robert"));
```

```
  removeUser(t4, findUser(t4, "pop"));
// removeUser(t4, findUser(t4, "teemo"));
  displayTree(t4);
//  removeUser(t4, findUser(t4, "teemo"));

  char* ss1 = calloc(sizeof(char),10); strcat(ss1, "james");
  char* ss2 = calloc(sizeof(char),10); strcat(ss2, "ann");
  char* ss3 = calloc(sizeof(char),10); strcat(ss3, "bob");
  char* ss4 = calloc(sizeof(char),10); strcat(ss4, "robert");
  char* ss5 = calloc(sizeof(char),10); strcat(ss5, "pop");
  char* ss6 = calloc(sizeof(char),10); strcat(ss6, "teemo");

  //more tests on delete user
  printf("testing deleteUser\n");
  Tree* t3 = newTree();
  addUser(t3, ss1, 12);
  addUser(t3, ss2, 12);
  addUser(t3, ss3, 12);
addUser(t3, ss4, 12);
  addUser(t3, ss5, 12);
  addUser(t3, ss6, 12);

  displayTree(t3);

  deleteUser(t3, ss1);
  deleteUser(t3, ss2);
  deleteUser(t3, ss3);
  deleteUser(t3, ss4);
  deleteUser(t3, ss5);
  deleteUser(t3, ss6);

  displayTree(t3);
#endif
  printf("testing deleteUser again\n");

  Tree* t5 = newTree();
  displayTree(t5);
  addUser(t5, "bob", 12);
  displayTree(t5);
  addUser(t5, "robert", 12);
```

```
  displayTree(t5);
  addUser(t5, "aaa", 12);
  displayTree(t5);
  addUser(t5, "bbb", 12);
  displayTree(t5);
  addUser(t5, "ccc", 12);
  displayTree(t5);

  deleteUser(t5, "bob");
  displayTree(t5);
  deleteUser(t5, "robert");
  displayTree(t5);
  deleteUser(t5, "aaa");
  displayTree(t5);
  deleteUser(t5, "bbb");
  displayTree(t5);
  deleteUser(t5, "ccc");
  displayTree(t5);

  return 0;
}
```

7.3.5 Utility.c

```
//This file has functions used in both server.c and client.c

#include "utility.h"
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>


char* trim(char* str)
{
  char *end;

  // Trim leading space
```

```
  while(isspace(*str)) str++;

  if(*str == 0) // All spaces?
    return str;

  // Trim trailing space
  end = str + strlen(str) - 1;
  while(end > str && isspace(*end)) end--;

  // Write new null terminator
  *(end+1) = '\0';

  return str;
}

/* case blind strcmp */

/**
 * caseless compare
 * used code found on a forum for this:
    http://cboard.cprogramming.com/c-programming/63091-strcmp-without-case-sensitivity.
 */
int strcmpc (const char *p1, const char *p2)
{
  register unsigned char *s1 = (unsigned char *) p1;
  register unsigned char *s2 = (unsigned char *) p2;
  unsigned char c1, c2;

  do
    {
      c1 = (unsigned char) toupper((int)*s1++);
      c2 = (unsigned char) toupper((int)*s2++);
      if (c1 == '\0')
   {
     return c1 - c2;
   }
    }
  while (c1 == c2);

  return c1 - c2;
```

```
}

void error(const char *msg)
{
  perror(msg);
  exit(1);
}
```

7.3.6 Client.h

```
#ifndef _CLIENT_H
#define _CLIENT_H

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#include <errno.h>
#include <signal.h>

#include <string.h>
#include <signal.h>
#include "utility.h"

void changeName();

static volatile int keepRunning = 1;
void  inthandler(int sig);

#endif
```

7.3.7 Server.h

```
#ifndef SERVER_H_
#define SERVER_H_

/* headers */
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <errno.h>
#include <signal.h>

#include "server.h"
#include "tree.h"
#include "utility.h"

/* function prototypes */
int prompt(int sock);
int parse (char* message, int sock);
void* connection (void *sock_void);
void* interactive(void *zero);
void sigint_handler(int sig);
void printUsage(void);
void destroyServer(void);
void sendAll(char* msg);
void sendAllHelper(char* msg, Node* n);
void sendAllBut(char* msg1, char* msg2, char* name);
void sendAllHelperBut(char* msg1, char* msg2, char* name, Node* n);
void populateList(char* list);
void populateListHelper(char* list, Node* n);

/* variables & constants */
#define MAX_CLIENTS (100)
```

```
pthread_mutex_t mutex;    /* mutex used to protect regions */
sem_t active_connections;        /* the semaphores to limit the
    number of clients */
Tree* users;
pthread_t* threads;
static int thread_counter = 0;
static int thread_size = 10;
static volatile int running = 1;

#endif
```

7.3.8 Tree.h

```
#ifndef _TREE_H
#define _TREE_H

#include <string.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct binaryTree
{
  struct node* root;
  int counter;
} Tree;

typedef struct node
{
  struct user *data;
  struct node *right;
  struct node *left;
} Node;

typedef struct user
{
  char* id;
  int socket;
} User;
```

```
int validate(Tree* t, char* username);
int validateHelper(Node* t, char* username);
Node* newNode(User* data);
void displayTree(Tree* t);
void displayTreeHelper(Node* current);
int compare(User* a, User* b);
Tree* newTree();
User* newUser(char* name, int socket);
Tree* addUser(Tree* current, char* name, int socket);
Node* addUserHelper(Node* t, User* data);
Node* removeUser(Tree* t, User* data);
Node* removeUserHelper(Node* current, User* data);
Node* min(Node* current);
User* findUser(Tree* t, char* name);
User* findUserHelper(Node* current, char* name);
void deleteNode(Node* n);
void deleteTree(Tree* t);
void deleteAll(Node* n);
void deleteNodeData(Node* n);
Node* deleteUser(Tree* t, char* data);
Node* deleteUserHelper(Node* t, char* data);

#endif
```

7.3.9 Utility.h

```
#ifndef UTILITY_H_
#define UTILITY_H_

#define DEBUG (1)

char* trim(char* string);
int strcmpc (const char *p1, const char *p2);
void error(const char *msg);

#endif
```

7.3.10 makefile

```
all: server client testDriver

server: server.o tree.o utility.o
   gcc server.o tree.o utility.o -pthread -g -o server

server.o: server.h server.c
   gcc -pthread -g -c server.c

client: client.o utility.o
   gcc client.o utility.o -pthread -g -o client

client.o: client.h client.c
   gcc -pthread -g -c client.c

tree.o: tree.h tree.c
   gcc -pthread -g -c tree.c

testDriver: testDriver.o tree.o utility.o
   gcc testDriver.o tree.o utility.o -pthread -g -o testDriver

testDriver.o: testDriver.c
   gcc -pthread -g -c testDriver.c

utility.o: utility.h utility.c
   gcc -pthread -g -c utility.c

clean:
   rm *.o
   rm server
   rm client
   rm testDriver
tar:
   tar README makefile server.c client.c tree.c tree.h chat.h
```