CS 3516 Class Project Report

Adam Camilli (aocamilli@wpi.edu)

June 22, 2017

Abstract

This report outlines the design and testing of a chat program, following a single server-multiple client model and written in C using Linux socket commands. It consists of two programs server.c, which creates a socket to bind to all available addresses for the computer, and client.c, which creates a socket to connect to said server. Each client is handled with a thread by the server. The program is simple and procedural, making maximum use of already available libraries and techniques for multithreaded programming. It was tested with various concrete cases meant to test the larger conceptual capabilities of a functioning server, and is capable of facilitating concurrent communication between up to 20 clients.

Contents

1 Project Description **2**

2 Design **2**

3 Testing and Validation **3**

4 Future Development **3**

5 Conclusion - Solution Summary **3**

6 Appendix **3**

6.1 References 3

6.2 Test Cases and Results 4

6.3 Code 8

1 Project Description

The intention of this project was to write a chat simulation in C, using the Linux socket commands to emulate the one-to-many server-to-clients model. This was accomplished with the creation of two executable programs, a single server, which creates a socket and binds it to all available interfaces, and up to 20 clients, which create their own sockets and then connect to the newly established server. The server additionally handles the display of the chat, and makes use of the `Pthread` library to handle each client and provide a concurrent communication environment for all of the clients simultaneously.

2 Design

The basic workflow of the two programs `server.c` and `client.c` can be summarized as follows:

Connection:

1. Server program first creates a socket and binds it to all available interfaces using the constant `INADDR_ANY` from `<arpa/inet.h>`.
2. Server then begins listening for up to 20 clients with `listen()`.
3. Client creates a socket and attempts to connect to whatever address is provided by the user, usually 127.0.0.1.
4. Server, upon connecting with a client, begins a loop in which it provides the client with a thread and then loops continuously to avoid terminating the server.
5. As `main()` loops, an instance of the thread handler function pointer `*connection_handler()` deals with each client.
6. Communication may be established this way for up to 20 clients.

Communication:

1. Having established a connection to a server, each instance of `client.c` will prompt user first for their name and thereafter for whatever input they choose.
2. Both the server thread and client instance wait on each other using `recv()` from `<sys/socket.h>`, and write to each other using `write()`. After client input is sent, depending on what it is, the thread handler will:
   (a) Add a pair to the global array `nametable` consisting of the client's chosen name and their unique socket descriptor. This keeps track of each client's nickname to be displayed next to their messages.
   (b) Display their message to all clients in the form: `<Nickname>: <Message>`
   (c) Send the user a list of currently connected users.
   (d) Indicate client has disconnected when `recv()` returns 0 (client will close their own socket).

3 Testing and Validation

For the purposes of testing the program, I sought to thoroughly test all aspects of my code in three categories:

1. The basic capabilities of the chat server.
2. The basic capabilities of the server applied with multiple clients (concurrency).
3. Edge cases, as well as tests for erroneous input.

Most of my test cases were created before I started coding, to serve as goals as I progressed. As a rule, I tried to create single concrete cases, with an input and expected output, that together represented a more abstract and complete capability on the part of the program. For example, an abstract goal to have two clients connect to the server, name give themselves nicknames, and chat with each other. This powerful capability is testable concretely by creating three terminals, running `./server` on one and running the sequence

```
./client 127.0.0.1
<Some_Name>
<Phrase1>
<Phrase2>
.
.
.
```

on each of the other two. My testing strategy was essentially to assign one concrete cases to each abstract ability my server was supposed to have.

4 Future Development

In the future, I would first and foremost attempt to find a more complete plan before I began coding, and find it earlier than I did. My first instinct was to use a multithreaded server to deal with each client, but this strategy proved arduous and prone to error, and I can't imagine it would be very scaleable past the simple single-machine server I created. In particular, after I'd completed most of the server and client and had little time left to complete the project, I began to read about more standard methods of coding servers in C, such as the use `poll()` or `select()`. These would have like made the project much easier. For example, dealing with a `SIG_PIPE` signal sent by a forcefully disconnected server to a client would have been much easier to deal with if I'd used `poll()` [3].

5 Conclusion - Solution Summary

This report has described the semi-successful implementation of a single server-multiple client model chat program, written in C using Linux socket commands.

6 Appendix

6.1 References

1. https://stackoverflow.com/questions/23889062/c-how-to-handle-sigpipe-in-a-multithreaded-environment
2. http://www.tenouk.com/download/pdf/Module41.pdf
3. http://www.linuxprogrammingblog.com/all-about-linux-signals?page=show

6.2 Test Cases and Results

For the purposes of testing the program, I designed a number of test cases, organized here under three groups:

1. Tests of the basic capabilities of the chat server.

   (a) Starting the server:

```
 $ make
gcc -g -pthread server.c -o server
gcc -g -pthread client.c -o client
 $ ./server
Socket created
bind done
Waiting for incoming connections...
```

   (b) Running a single client named Adam on 127.0.0.1:

```
Client Terminal:
 $ ./client 127.0.0.1
Socket created
Connected to 127.0.0.1
Type '#' to end the session.
Type '$' for a list of currently connected users.

Enter your name (letters, numbers, and spaces only): Adam
Now type your message:

Server Terminal:

 $ ./server
Socket created
bind done
Waiting for incoming connections...
Adam has joined chat!
```

   (c) Run a single client named Adam on another viable address (for example, my computer's address on my home network):

```
Client Terminal:
 $ ./client 192.168.**.***
Socket created
Connected to 192.168.**.***
Type '#' to end the session.
Type '$' for a list of currently connected users.

Enter your name (letters, numbers, and spaces only): Adam
Now type your message:

Server Terminal:

 $ ./server
Socket created
bind done
Waiting for incoming connections...
Adam has joined chat!
```

   (d) Performing each client capability: Chat, List of Users ('$'), Disconnect ('#'):

```
Client Terminal:

Enter your name (letters, numbers, and spaces only): Adam
Now type your message: hello world
Now type your message: $
Adam

Now type your message: #
Goodbye!
 $

Server Terminal:

 $ ./server
Socket created
bind done
Waiting for incoming connections...
Adam has joined chat!
Adam: hello world
Adam disconnected
```

2. Tests of the basic capabilities of the server applied with multiple clients (concurrency)

   (a) After disconnecting, reconnect in the same terminal as a new client, this time named John:

```
Client Terminal:

 $ ./client 127.0.0.1
Socket created
Connected to 127.0.0.1
Type '#' to end the session.
Type '$' for a list of currently connected users.

Enter your name (letters, numbers, and spaces only): John
Now type your message: hello
Now type your message:

Server Terminal:

Adam disconnected
John has joined chat!
John: hello
```

   (b) Run two clients and have them chat simultaneously with each other:

```
Client 1 Terminal:

 $ ./client 127.0.0.1
Socket created
Connected to 127.0.0.1
Type '#' to end the session.
Type '$' for a list of currently connected users.

Enter your name (letters, numbers, and spaces only): John
Now type your message: Mom?
```

```
Now type your message: How are you
Now type your message:

Client 2 Terminal:

 $ ./client 127.0.0.1
Socket created
Connected to 127.0.0.1
Type '#' to end the session.
Type '$' for a list of currently connected users.

Enter your name (letters, numbers, and spaces only): John's Mom
Now type your message: Hello Son
Now type your message: good u?
Now type your message:

Server Terminal:

$ ./server
Socket created
bind done
Waiting for incoming connections...
John has joined chat!
John: Mom?
John's Mom has joined chat!
John's Mom: Hello Son
John: How are you
John's Mom: good u?
```

(c) Run one client, have them request a list of users once when they are alone, and again when one or more other clients have connected. Then have them request it after one or more have disconnected.

```
Client 1 Terminal:

 $ ./client 127.0.0.1
Socket created
Connected to 127.0.0.1
Type '#' to end the session.
Type '$' for a list of currently connected users.

Enter your name (letters, numbers, and spaces only): Alice
Now type your message: $
Alice

Now type your message: $
Alice
Bob
Now type your message: $
Alice

Now type your message:

Server Terminal:
```

```
 $ ./server
Socket created
bind done
Waiting for incoming connections...
Alice has joined chat!
Bob has joined chat!
Bob disconnected
```

3. Edge cases, as well as tests for erroneous input.

   (a) Run > 20 clients.

```
Client 21 Terminal:

 $ ./client 127.0.0.1
Socket created
Connected to 127.0.0.1
Type '#' to end the session.
Type '$' for a list of currently connected users.

Enter your name (letters, numbers, and spaces only): John
Sorry, server is full!
 $
```

   (b) Attempt to run client when no server is running.

```
 $ ./client 127.0.0.1
Socket created
Connect failed. Server is not listening for clients.
: Connection refused
 $
```

   (c) Attempt to run client with no address, an incorrectly formatted one, multiple input, etc.

```
 $ ./client
Socket created
Usage: ./client <IP Address>
 $ ./client badipaddress
Socket created
Connect failed. Server is not listening for clients.
: Network is unreachable
aocamilli@Cordelia $ ./client badipaddress andsomeotherstuff
Socket created
Usage: ./client <IP Address>
 $
```

   (d) Disconnect a server while a client is running.

```
Client Terminal

 $ ./client 127.0.0.1
Socket created
Connected to 127.0.0.1
Type '#' to end the session.
Type '$' for a list of currently connected users.

Enter your name (letters, numbers, and spaces only): Adam
```

```
Now type your message: one
Now type your message: two
Now type your message: three
Now type your message: four
 $
```

```
Server Terminal

 $ ./server
Socket created
bind done
Waiting for incoming connections...
Adam has joined chat!
Adam: one
Adam: two
^C
 $
```

Here we can unfortunately see an unhandled error in my coding that I did not have time to fix. The server is ended with Ĉ, as it also unfortunately has no other way to exit via the terminal). This sends the signal `SIG_PIPE` to the client program [1], which my program is not currently equipped to handle. As such, the client is not able to remain stable.

In addition to these, various sanity checks I performed while working are still visible in the code as commented out print statements.

6.3 Code

Server.c

```
/* A server which functions as a chat program and coordinator for up to 20 users on a Linux computer.
 *
 * This program creates a socket and binds it to 127.0.0.1:8888 . Clients can then create their own soc
 * connect from separate terminals on this computer to 127.0.0.1.
 * They may then chat with each other, being able to print a list of currently connected users or disco
 *
 * Author: Adam Camilli (aocamilli@wpi.edu), 6/20/17
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <pthread.h>

#define MAX_MSG_LEN 1000
#define MAX_CLIENTS 20

/*********************************************************************************************
 **** Lookup table struct + accompanying functions and global vars for client nicknames ****
 *********************************************************************************************/
typedef struct pair {
```

```
  char* name;
  int socket;
} pair_t;

pair_t *nametable[MAX_CLIENTS];

pair_t* search_by_socket(pair_t **table, int query) {
  for (int i = 0; i < MAX_CLIENTS; i++) {
    if (table[i] != NULL) {
      if (table[i]->socket == query) {
return table[i];
      }
    }

  } return NULL;
}

pair_t* search_by_name(pair_t **table, char *query) {
  for (int i = 0; i < MAX_CLIENTS; i++) {
    if (table[i] != NULL) {
      if (table[i]->name == query) {
return table[i];
      }
    }
  } return NULL;
}

int add_pair(pair_t **table, char *name, int socket) {

  if (table[MAX_CLIENTS - 1] != NULL) return -1;

  pair_t *pair = (pair_t *) malloc(sizeof(pair_t));
  // Essential to use duplicate pointer since name pointer will not be static
  pair->name = strdup(name);
  pair->socket = socket;

  for (int i = 0; i < MAX_CLIENTS; i++) {
    if (table[i] == NULL) {
      table[i] = pair;
      return 0;
    }
  } return -1;

}

int delete_pair(pair_t **table, char *name, int socket) {
  // Don't try to delete non-existent pair
  if (search_by_name(table, name) == NULL || search_by_socket(table,socket) == NULL)
    return -1;

  for (int i = 0; i < MAX_CLIENTS; i++) {
    if (table[i] != NULL) {
      if (table[i]->socket == socket && table[i]->name == name) {
table[i] = NULL;
```

```
return 0;
      }
    }
  } return -1;

}

/*************************************************
 **** Thread Handler function and main program ****
 *************************************************/

void *connection_handler(void *);

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

  // Initialize nametable with null elements to mark it empty
  for (int i = 0; i < MAX_CLIENTS; i++)
    nametable[i] = NULL;

  int socket_desc , client_sock , c , *new_sock;
  struct sockaddr_in server , client;

  // Create the socket
  socket_desc = socket(AF_INET , SOCK_STREAM , 0);
  if (socket_desc == -1)
    {
      printf("Could not create socket\n");
    }
  puts("Socket created");

  // Populate the sockaddr_in struct
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = INADDR_ANY; // All available interfaces
  server.sin_port = htons( 8888 );

  // Bind it to all available interfaces
  if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0) {
    //print the error message
    perror("bind failed. Error");
    return 1;
  }
  puts("bind done");

  // Now begin listening for up to clients
  listen(socket_desc, 20);
  puts("Waiting for incoming connections...");
  c = sizeof(struct sockaddr_in);

  // Upon connecting with client provide them with a thread for concurrent communication with server
  while( (client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&c)) ) {

    // printf("Connection accepted from %d\n", client_sock);

    pthread_t sniffer_thread;
```

```
    new_sock = malloc(sizeof *new_sock);
    *new_sock = client_sock;

    if( pthread_create( &sniffer_thread, NULL, connection_handler, (void*) new_sock) < 0) {
      perror("could not create thread");
      return 1;
    }

    // Loop so that we don't terminate program
    // printf("Handler assigned to %d\n", client_sock);
  }

  if (client_sock < 0) {
    perror("accept failed");
    return 1;
  }

  return 0;
}

void *connection_handler(void *socket_desc) {

  int sock = *(int*)socket_desc; // Get the socket descriptor
  int read_size;
  char *message;
  char client_message[MAX_MSG_LEN];
  char *name;

  int named = 0; // Whether or not client has chosen a nickname
  int full_violation = 0; // Whether or not client tried to connect when server was full

  // Receive name and then messages from client
  while (((read_size = recv(sock, client_message, MAX_MSG_LEN, 0)) > 0)) {
    if (!named) {
      if (add_pair(nametable, client_message, sock) < 0) {
char *full = "#full";
write(sock, full, sizeof full);
full_violation++;
break;
      }

      write(sock, client_message, sizeof client_message);

      printf("%s has joined chat!\n", client_message);
      named = 1;
    } else {
      // Send client list of users if requested
      if (client_message[0] == '$') {
char *userlist = (char *) malloc(MAX_MSG_LEN + 1);
memset(userlist, 0, sizeof userlist);
        for (int j = 0; j < MAX_CLIENTS; j++) {
  if (nametable[j] != NULL) {
    char *temp = strcat(strdup(nametable[j]->name),"\n");
    userlist = strcat(userlist,temp);
```

```
  }
}
write(sock, userlist, sizeof userlist);
free(userlist);
      } else { // Always respond so client knows its message has been received
write(sock, client_message, sizeof client_message);
printf("%s: %s", search_by_socket(nametable,sock)->name, client_message);

      }
    }
  }

  // If client disconnects
  if (read_size == 0) {
      printf("%s disconnected\n", search_by_socket(nametable,sock)->name);
      fflush(stdout);
  } else if(read_size == -1) {
      perror("recv failed");
  }

  // Delete client name/socket entry upon exit
  if (!full_violation)
    delete_pair(nametable, search_by_socket(nametable,sock)->name, sock);
  // Free thread's socket pointer upon exit
  free(socket_desc);
  close(sock);
  pthread_exit(NULL);
}
```

Client.c

```
/* A client program which creates a socket with a server and communicates with it.
 *
 * Clients upon connecting with server
 * Author: Adam Camilli (aocamilli@wpi.edu), 6/20/17
 */

#include <stdio.h> //printf
#include <string.h>    //strlen
#include <sys/socket.h>    //socket
#include <arpa/inet.h> //inet_addr
#include <unistd.h>

#define MAX_MSG_LEN 1000

/* Helper function to help remove non alphanumeric/space characters from a C-Style string */
int is_alphanumeric(char ch) {
  int not_uppercase = (ch < 'A' && ch > 'Z');
  int not_lowercase = (ch < 'a' && ch > 'z');
  int not_numeric = (ch < '1' && ch > '9');

  if (not_uppercase && not_lowercase && not_numeric)
    if (ch != ' ')
      return 0;
```

```
  return 1;
}

int main(int argc , char *argv[]) {
  // Whether we disconnected from server of our own accord
  int clean_break = 0;

  int sock;
  struct sockaddr_in server;
  char message[MAX_MSG_LEN] , server_reply[MAX_MSG_LEN];
  int named = 0;

  // Create socket our socket
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock == -1) {
    printf("Could not create socket\n");
  } printf("Socket created\n");

  // If improper arguments given
  if (argc < 2 || argc > 2) {
    printf("Usage: ./client <IP Address>\n");
    return 1;
  }

  // Populate the sockaddr_in struct
  server.sin_addr.s_addr = inet_addr(argv[1]);
  server.sin_family = AF_INET;
  server.sin_port = htons( 8888 );

  // Connect to server
  if (connect(sock , (struct sockaddr *)&server , sizeof(server)) < 0) {
    perror("Connect failed. Server is not listening for clients.\n");
    return 1;
  }

  printf("Connected to %s\n", argv[1]);
  printf("Type '#' to end the session.\n");
  printf("Type '$' for a list of currently connected users.\n\n");

  // Continuously communicate with server
  while (1) {
    if (!named) { // Name not chosen
      char name[MAX_MSG_LEN];
printf("Enter your name (letters, numbers, and spaces only): ");
fgets(name, sizeof name, stdin);

if (name[0] == '$' || name[0] == '#' || name[0] == '@') {
  printf("Must choose a name first.\n");
  continue;
}

// Clear any non A-Z, a-z, 0-9 characters from name
int i;
for (i = 0; (name[i] != '\n') && i < MAX_MSG_LEN; i++) {
```

```
  if (is_alphanumeric(name[i]))
    message[i] = name[i];
}
named = 1;
      } else if (named) {
printf("Now type your message: ");
fgets(message, sizeof message - 1, stdin);
      }

      // Disconnect if requested to
      if (message[0] == '#') {
printf("Goodbye!\n");
clean_break = 1;
break;
      }

      // Otherwise send message to server
      if (send(sock, message, sizeof message, 0) < 0) {
puts("Send failed.");
break;
      }

      // Receive a reply from the server
      if (recv(sock, server_reply, MAX_MSG_LEN, 0) < 0) {
puts("recv failed");
break;
      } if (message[0] == '$') {
printf("%s\n",server_reply);
      } else if (server_reply[0] == '#') {
printf("Sorry, server is full!\n");
clean_break = 1;
break;
      }
  }

  close(sock);
  if (!clean_break)
    printf("Server lost: Relaunch client to connect to another. Goodbye!\n");
  return 0;
}
```