

# ECE470:Network Client-Sever Programming

## Project 1: Part 1

Aaron Jesus Valdes

March 2, 2021

# Contents

<b>1</b>	<b>Project Description</b>	<b>3</b>
<b>2</b>	<b>Specifications</b>	<b>3</b>
<b>3</b>	<b>Assumptions</b>	<b>3</b>
<b>4</b>	<b>Initial Design</b>	<b>4</b>
4.1	Business Logic . . . . .	5
4.1.1	Basic Operations . . . . .	5
4.1.2	Storyboard . . . . .	5
4.2	Sever state chart . . . . .	6
4.3	Client State Chart . . . . .	7
4.4	Application Protocol Design . . . . .	7
<b>5</b>	<b>Example</b>	<b>8</b>
<b>6</b>	<b>Data model</b>	<b>8</b>
<b>7</b>	<b>Message Protocol</b>	<b>9</b>
7.1	Client Marshal . . . . .	9
7.1.1	Header file . . . . .	9
7.1.2	Test File . . . . .	12
7.1.3	Makefile . . . . .	12
7.2	Results . . . . .	13
7.3	Server Marshal . . . . .	13
7.3.1	Header File . . . . .	13
7.3.2	Test program . . . . .	15
7.3.3	Makefile . . . . .	15
7.3.4	Result . . . . .	16
<b>8</b>	<b>TCP Protocols</b>	<b>16</b>
8.1	Client . . . . .	16
8.1.1	Header File . . . . .	16
8.1.2	Test Program . . . . .	20
8.1.3	Makefile . . . . .	21
8.1.4	Results . . . . .	21
8.2	Client . . . . .	21
8.2.1	Header File . . . . .	21
8.2.2	Test Program . . . . .	21
8.2.3	Makefile . . . . .	22
8.2.4	Results . . . . .	23

# 1 Project Description

Designing a smart home remote access system which follows the Network/Client architecture. This system allows the user to control certain devices from their home such lights, alarm, and locks from a remote location.

## 2 Specifications

- Supports smart lights
  - Can turn on and off lights
  - Can change the color and brightness of lights
  - Groups lights into rooms
- Supports alarm system
  - Can arm and disarm the alarm
  - Can have more than one alarm system
  - Can store 4 digit pin per alarm
- Supports Electronic Locks
  - Can open and lock the locks
  - Can have more than 4 locks
  - Can store a 5 digit pin per lock
- Supports more devices
  - Can control the thermostat
    - Turn on or off heat/cold
    - Set temperature
  - Can check the security cameras
    - Can store the video from a security camera
  - Can control blinds and drapes
    - Opens and closes the blinds

## 3 Assumptions

- We are only working with a single house
- There are multiple rooms in the house
- Each room has multiple lights

- There is only one alarm
- There are at least 4 locks
- One user at a time can connect
- The server is in the home
- The server can be accessed both inside and outside of the house
- The user needs to check and alter status of the devices
- The method of communication is
- The server must be protected by having a login page

## 4 Initial Design

### Data Model

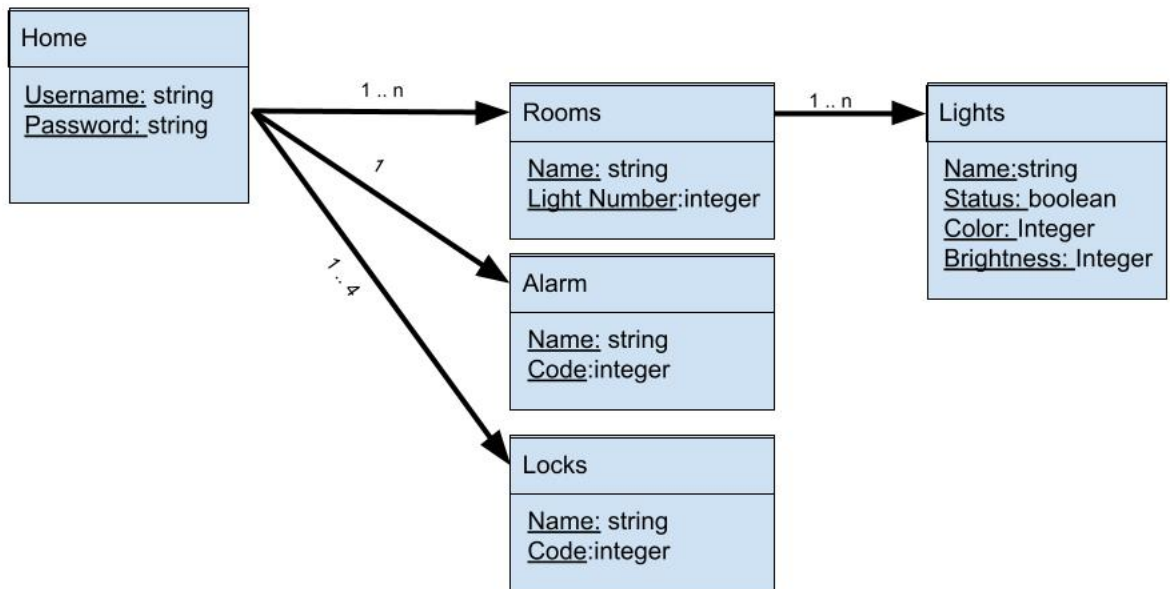


Figure 1: Data models follow the specifications

This is the simplest data model which we can use as a template given the fact that as we move one with this project we are going to be adding more features such as setting RGB colors, change lock combinations, setting the brightness, and later on be able to add more devices

## 4.1 Business Logic

### 4.1.1 Basic Operations

- Login/Logout
- Check Status of each device
- Change Status of each devices

### 4.1.2 Storyboard

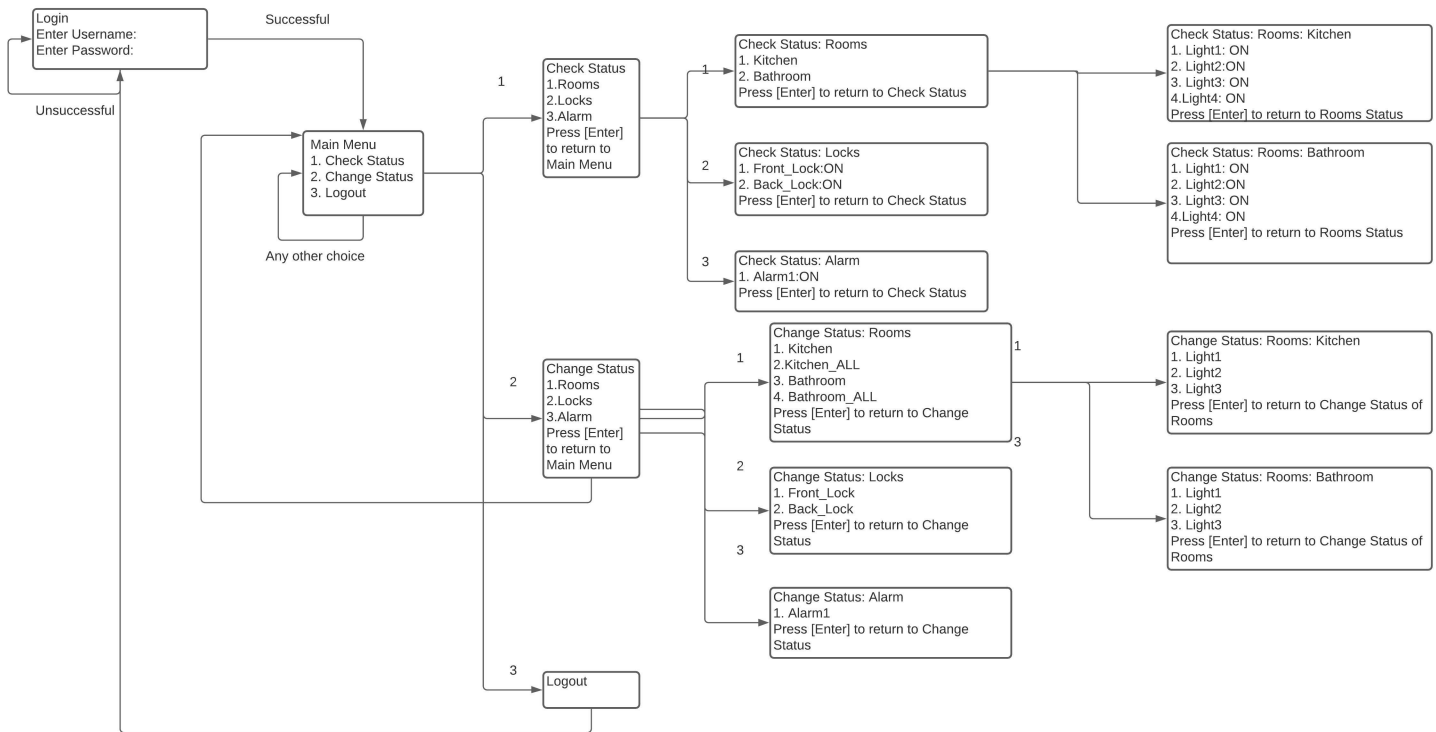


Figure 2:

The reason I chose to divide my check status and change status based on devices is because that is the method that my google home uses show the status of my devices and change the status of my devices. In my opinion the most effective method is by simply having a single main menu with a all the options since it leads to less amount of packets being transferred back and forth between the client and the server and can therefore reduce bandwidth. As we move on, I am going to implement more menus for the alarm, locks, and lights as way to provide more features for each device

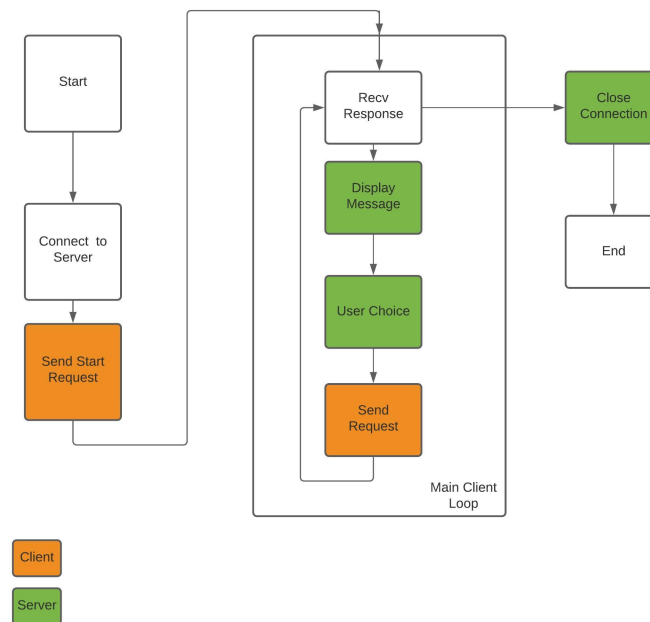
## 4.2 Sever state chart



The server state diagram starts by simply waiting for the client to establish a connection. After establishing a connection, the user is going to be asked to send their username and password which is going to be check by the server based on the data model. If the username and password is correct then we move on to the

main menu else we return back to the server login menu. After entering the main menu, the user needs to send the server a request for either checking status of a device, changing the status of a device, or logging out . After the user request the either check or change the status, they are going to receive the list of devices available to choose from. After requesting a device, the user is going to either the rooms, locks, or alarms available to them so they can either change or check the status. For changing the status there is an extra step in which the server updates the data model for each change and returns back to displaying the status of that device. For each menu there is always a return option which is simply by requesting to press enter.

### 4.3 Client State Chart



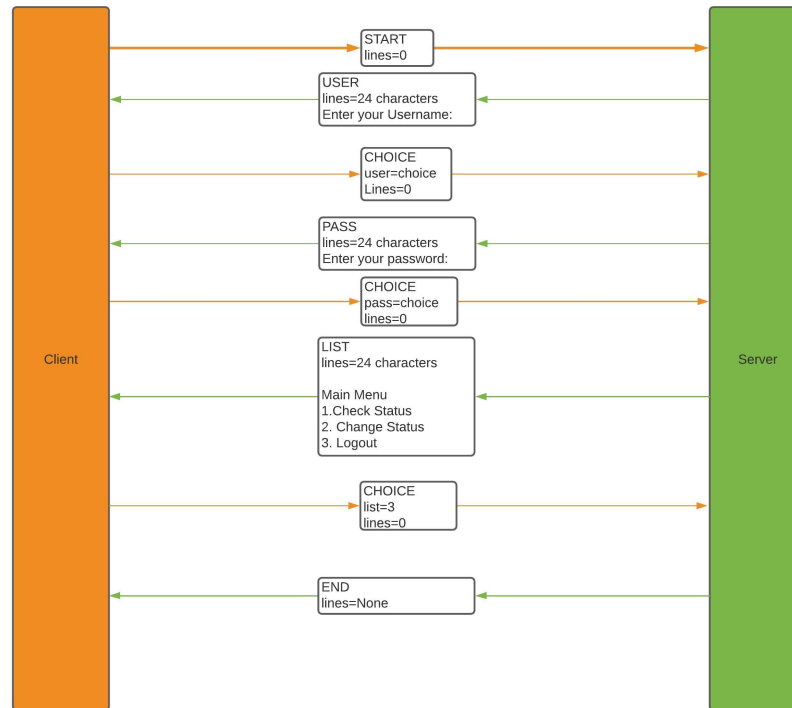
The client state diagram starts by simply establishing a connection between the server and the client. Then the server sends the client a message, which the user uses to make a request. After the request is sent to the server, the server makes a decision based on that request and executes a command. The connection between the server and the client ends when the user decides to log out.

### 4.4 Application Protocol Design

Client Application Protocols	Parameters	Body	Description
START	none	none	Establishes connection with the server
CHOICE	Lines=24 character, lines=0	None	Based on the options of the server, the user can send up to 24 character to the server
END	None	none	Ends the connection between the client and the server

Server Application Protocols	Parameters	Body	Description
USER	Lines=24 character	Line	Ask the user for the username
PASS	Line=24 character	Line	Ask the user for the password
LIST	Line=24 character	Line(s)	Gives the user options which they can chose from
ERROR	Line=24 character	Line(s)	Send the user information about an error in the system and waits for his choice
END	None	None	The server ends the connection between the client and the server

## 5 Example



## 6 Data model

```
#ifndef _DATAMODEL_H_
#define _DATAMODEL_H_
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class light
{
public:
    string name;
    bool status;
};

class room
{
public:
    vector<light> l;
};

class rooms
{
}
```



```

public:
vector<room> r;
string name;
int light_number;
};

class alarm
{
public:
string name;
int code;
};

class lock
{
public:
string name;
int code;
};

class locks
{
public:
vector<lock> llock;
};

class home
{
private:
string password;
string username;
public:
rooms n;
alarm a;
locks lo;
void setUsername(string newUsername);
void setPassword(string password);
};
#endif

```

## 7 Message Protocol

### 7.1 Client Marshal

#### 7.1.1 Header file

```

#ifndef _CLIENT_MARSHAL_H_
#define _CLIENT_MARSHAL_H_
#include <iostream>
#include <string>
#include <stdio.h>
#include <stdlib.h>

```

```

#include <sstream>
#include <vector>
using namespace std;
class server_message
{
public:
string options;
string message;
vector<string> lines;
int num_lines;
void printing()
{
cout<<options<<"      "<<num_lines<<"      "<<message<<endl;
for(unsigned int i=0;i<lines.size();i++)
{
cout<<lines[i]<<endl;
}

}
};

class client_message
{
public:
string options;
int decision;
void printing()
{
cout<<options<<"      "<<decision;
}
};

server_message unmarshal(string message)
{
string command;
server_message res;
stringstream ss(message);
ss>>command;
if(command=="USER")
{
res.options="USER";
getline(ss,res.message);
}
else if(command=="PASS")
{
res.options="PASS";
getline(ss,res.message);
}
else if(command=="LIST")
{
res.options="LIST";
getline(ss,res.message,'\\');
ss>>res.num_lines;
vector<stringstream> tt(res.num_lines);
string temp;
int i=0;
while(ss>>temp)
{
if(res.message=="\\\\\\")
{

```

```

break;
}
else if(temp=="\\")
{
i++;
}
else
{
tt[i]<<temp<<" ";
}
}
for(unsigned int j=0;j<tt.size();j++)
{
string temp2;
getline(tt[j],temp2);
res.lines.push_back(temp2);
}
}
else if(command=="ERROR")
{
res.options="ERROR";
getline(ss,res.message);
ss>>res.message;
}
else if(command=="END")
{
res.options="END";
getline(ss,res.message);
ss>>res.message;
}
else
{
}
}
return res;
}

```

```

string marshal(client_message cm)
{
stringstream ss;
string result;
if(cm.options=="START")
{
ss<<cm.options<<" "<<cm.decision;
}
else if(cm.options=="CHOICE")
{
ss<<cm.options<<" "<<cm.decision;
}
else if(cm.options=="END")
{
ss<<cm.options<<" "<<cm.decision;
}
else if(cm.options=="ERROR")
{
ss<<cm.options<<" "<<cm.decision;
}
else
{

```

```

}
getline(ss,result);
return result;
}
#endif

```

### 7.1.2 Test File

```

#include <iostream>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "client_marshall.h"
using namespace std;
int main(int argc,char*argv[])
{
if(argc==2)
{
cout<<"Testing unmarshal"<<endl;
server_message test1;
test1 =unmarshal(argv[1]);
test1.printing();
}

if(argc>2)
{
cout<<"Testing Marshall"<<endl;
client_message test;
test.decision=atoi(argv[2]);
test.options=argv[1];
cout<<marshal(test)<<endl;
}
return 0;
}

```

### 7.1.3 Makefile

```

all:client server_marshall.h client_marshall.h
echo Testing the client
client:client.cpp client_marshall.h
g++ client.cpp -o client
test_client: client client_marshall.h
@./client "USER Enter Your Username:"
@./client "PASS Enter your Password:"
@./client "ERROR There is an Error:"
@./client "END Ending the Connection"
@./client "LIST Here are the options: \ 1 1.Option 1 \\"
@./client "LIST Here are the options: \ 2 1.Option 1 \ 2. Option 2 \\"
@./client "LIST Here are the options: \ 3 1.Option 1 \ 2. Option 2 \ 3. Option 3 \\"
@./client "LIST Here are the options: \ 4 1.Option 1 \ 2. Option 2 \ 3. Option 3 \ 4. Option 4 \\"
@ echo
@ echo
@./client "START" "1"
@./client "CHOICE" "1"
@./client "END" "1"
@./client "ERROR" "1"
clean:
rm client

```

## 7.2 Results

```
work@dastier:~/Documents/School/spring_2020/client_server_programming/project1/part1_2/client$ make test_client
Testing unmarshal
USER 0 Enter Your Username:
Testing unmarshal
PASS 0 Enter your Password:
Testing unmarshal
ERROR 0 There is an Error:
Testing unmarshal
END 0 Ending the Connection
Testing unmarshal
LIST 1 Here are the options:
1.Option 1
Testing unmarshal
LIST 2 Here are the options:
1.Option 1
2. Option 2
Testing unmarshal
LIST 3 Here are the options:
1.Option 1
2. Option 2
3. Option 3
Testing unmarshal
LIST 4 Here are the options:
1.Option 1
2. Option 2
3. Option 3
4. Option 4

Testing Marshall
START 1
Testing Marshall
CHOICE 1
Testing Marshall
END 1
Testing Marshall
ERROR 1
```

Figure 3: Results for the client marshalling

## 7.3 Server Marshal

### 7.3.1 Header File

```
#ifndef _SERVER_MARSHAL_H_
#define _SERVER_MARSHAL_H
#include <iostream>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include <vector>
using namespace std;
class client_message
{
public:
string command;
int decision;
void printing()
{
cout<<command<<" " <<decision<<endl;
}
};
```

```

class server_message
{
public:
string command;
string messa;
vector<string> lines;
int num_lines;
};

client_message unmarshal(string message)
{
string command;
int decision;
client_message res;
istringstream ss(message);
ss>>command>>decision;
res.command=command;
res.decision=decision;
return res;
}

string marshal(server_message sm)
{
stringstream ss;
string result;
string c[5]={"USER","PASS","LIST","ERROR","END"};
if(sm.command == c[0])
{
ss<<sm.command<<" "<<sm.messa<<"\\";
}
else if(sm.command == c[1])
{
ss<<sm.command<<" "<<sm.messa<<"\\";
}
else if(sm.command == c[2])
{
ss<<sm.command<<" "<<sm.num_lines<<" "<<sm.messa<<"\\";
for(int i=0;i<sm.num_lines;i++)
{
ss<<sm.lines[i]<<"\\n";
}
ss<<"\\n\\n";
}
else if(sm.command == c[3])
{
ss<<sm.command<<" "<<sm.messa;
}
else if(sm.command == c[4])
{
ss<<sm.command<<" "<<sm.messa;
}
else
{
}
getline(ss,result);
return result;
}
#endif

```

### 7.3.2 Test program

```
#include "server_marshall.h"
using namespace std;

int main(int argc, char*argv[])
{
    if(argc<3)
    {
        cout<<"Testing unmarshal"<<endl;
        client_message test=unmarshal(argv[1]);
        test.printing();
    }
    else
    {
        cout<<"Testing Marshal"<<endl;
        server_message test1;
        test1.command=argv[1];
        test1.messa=argv[2];
        switch(argc)
        {
            case 5:
            {
                test1.num_lines=atoi(argv[3]);
                test1.lines.push_back(argv[4]);
                break;
            }
            case 6:
            {
                test1.num_lines=atoi(argv[3]);
                test1.lines.push_back(argv[4]);
                test1.lines.push_back(argv[5]);
                break;
            }
            case 7:
            {
                test1.num_lines=atoi(argv[3]);
                test1.lines.push_back(argv[4]);
                test1.lines.push_back(argv[5]);
                test1.lines.push_back(argv[6]);
                break;
            }
        }
        cout<<marshal(test1)<<endl<<endl;
    }
    return 0;
}
```

### 7.3.3 Makefile

```
all:server  server_marshall.h
echo Testing the client

server:server.cpp server_marshall.h
g++ server.cpp -o server

test_server: server server.cpp server_marshall.h
@./server "START 2"
@./server "CHOICE 5"
```

```

@./server "END 1"
@./server "ERROR 2"
@./server "USER" " Enter your Username: "
@./server "PASS" "Enter your password: "
@./server "ERROR" "You have an error "
@./server "END" "Ending Connection"
@./server "LIST" "This are the options" "3" " 1. Option 1 " " 2. Option 2 " " 3. Option 3 "
@./server "LIST" "This are the options" "2" " 1. Option 1 " " 2. Option 2 "
@./server "LIST" "This are the options" "1" " 1. Option 1 "
clean:
rm server

```

### 7.3.4 Result

```

work@dastier:~/Documents/School/spring_2020/client_server_programming/project1/part1_2/server$ make test_server
g++ server.cpp -o server
Testing unmarshal
START 2
Testing unmarshal
CHOICE 5
Testing unmarshal
END 1
Testing unmarshal
ERROR 2
Testing Marshal
USER Enter your Username: \

Testing Marshal
PASS Enter your password: \

Testing Marshal
ERROR You have an error

Testing Marshal
END Ending Connection

Testing Marshal
LIST 3 This are the options\ 1. Option 1 \ 2. Option 2 \ 3. Option 3 \\\

Testing Marshal
LIST 2 This are the options\ 1. Option 1 \ 2. Option 2 \\\

Testing Marshal
LIST 1 This are the options\ 1. Option 1 \\\

```

Figure 4: Results for my message protocol

## 8 TCP Protocols

### 8.1 Client

#### 8.1.1 Header File

```

#ifndef _TCP_CLIENT_H_
#define _TCP_CLIENT_H_
#include <iostream>
#include <string>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>

```



```

#include<stdlib.h>
#include<unistd.h>
#define BACKLOG 5
#define DE_SERVER_PORT 12345
#define DE_CLIENT_PORT 12345
#define MAXBUFFERLEN 1024
#define SERV_IP "127.0.0.1"
using namespace std;

class client
{
private:
int sockfd_server;
int connection;
int sockfd_client;
int sockfd_client_acc;
string server_route;
string client_route;
int server_port;
int client_port;
sockaddr_in server_address={0};
sockaddr_in client_address={0};
public:
client();
client(string serverIP,int serverPort);
int send_message(string message);
string receive_message();
void closing_all();
void close_single(int option);
};

client::client()
{
connection=1;
sockfd_client_acc=0;
server_route=SERV_IP;
sockfd_server=socket(AF_INET,SOCK_STREAM,0);
sockfd_client=socket(AF_INET,SOCK_STREAM,0);
if(sockfd_server==-1)
{
perror("Failed to create socket for Communicating with the server");
exit(EXIT_FAILURE);
}

if(sockfd_client==-1)
{
perror("Failed to create socket for listening for message from Server");
exit(EXIT_FAILURE);
}

//To send messages
server_address.sin_family=AF_INET;
server_address.sin_port=htons(DE_SERVER_PORT);
inet_pton(AF_INET,server_route.c_str(),&(server_address.sin_addr));
memset(&(server_address.sin_zero),'\0',8);

//To make the client listen but especially to bind it to a port
client_address.sin_family=AF_INET;
client_address.sin_port=ntohs(DE_CLIENT_PORT);
client_address.sin_addr.s_addr=INADDR_ANY;

```

```

memset(&(client_address.sin_zero),'\0',8);

}

client::client(string serverIP,int port)
{
connection=1;
sockfd_client_acc=0;
server_route=serverIP;
sockfd_server=socket(AF_INET,SOCK_STREAM,0);
sockfd_client=socket(AF_INET,SOCK_STREAM,0);
if(sockfd_server== -1)
{
perror("Failed to create socket for Communicating with the server");
exit(EXIT_FAILURE);
}

if(sockfd_client== -1)
{
perror("Failed to create socket for listening for message from Server");
exit(EXIT_FAILURE);
}

//To send messages
server_address.sin_family=AF_INET;
server_address.sin_port=htons(port);
inet_pton(AF_INET,server_route.c_str(),&(server_address.sin_addr));
memset(&(server_address.sin_zero),'\0',8);

//To make the client listen but especially to bind it to a port
client_address.sin_family=AF_INET;
client_address.sin_port=ntohs(port);
client_address.sin_addr.s_addr=INADDR_ANY;
memset(&(client_address.sin_zero),'\0',8);
}

int client::send_message(string message)
{
char buffer[MAXBUFFERLEN];
FILE *stream;
//Converting string to char[]
int tn=message.size();
char temp[tn+1];
strcpy(temp,message.c_str());
//Creating the stream for our message
stream=fmemopen(temp,strlen(temp),"r");
int num_char_read=fread(buffer+1,sizeof(char),sizeof(buffer),stream);
buffer[0]=num_char_read;
sockfd_server=socket(AF_INET,SOCK_STREAM,0);
connection=connect(sockfd_server,(const struct sockaddr*)&server_address,sizeof(struct sockaddr));
if(connection== -1)
{
perror("Failure to connect to server");
exit(EXIT_FAILURE);
close(sockfd_server);
return -1;
}
}

```

```

int sending=write(sockfd_server,(const char*)buffer,strlen(buffer)+1);
if(sending==-1)
{
perror("Failure to send package");
exit(EXIT_FAILURE);
close(sockfd_server);
return -1;
}
close(sockfd_server);
return 1;
}

string client::receive_message()
{
FILE *stream;
char *bp;
size_t size;
stream=open_memstream(&bp,&size);
char buffer[MAXBUFFERLEN];

if(sockfd_client_acc==0)
{
int rc=bind(sockfd_client,(const struct sockaddr*)&client_address,sizeof(client_address));
if(rc==-1)
{
perror("Failed to bind client_address to socket");
close(sockfd_client);
exit(EXIT_FAILURE);
}
int listening=listen(sockfd_client,BACKLOG);
if(listening==-1)
{
perror("Failed to listen to socket");
close(sockfd_client);
exit(EXIT_FAILURE);
}
}

unsigned int sin_size=sizeof(struct sockaddr_in);
sockfd_client_acc=accept(sockfd_client,(struct sockaddr *)&server_address,&sin_size);
int received=read(sockfd_client_acc,(char *)buffer,MAXBUFFERLEN);
if(received==-1)
{
perror("Failed to received package");
close(sockfd_client);
close(sockfd_client_acc);
exit(EXIT_FAILURE);
}
fwrite(buffer+1,sizeof(char),buffer[0],stream);
fflush(stream);
string temp(bp);
return temp;
}

void client::closing_all()
{
close(sockfd_server);
close(sockfd_client);
close(sockfd_client_acc);
}

```

```

}

void client::close_single(int option)
{
    if(option==0)
    {
        close(sockfd_server);
    }
    else if(option==2)
    {
        close(sockfd_client);
    }
    else if(option==3)
    {
        close(sockfd_client_acc);
    }
    else
    {
        close(sockfd_server);
        close(sockfd_client);
        close(sockfd_client_acc);
    }
}
#endif

```

### 8.1.2 Test Program

Test to make sure the client can send a message

```

#include "tcp\_client.h"
using namespace std;
int main()
\{
    client test;
    test.send\_message("START 2");
    test.send\_message("CHOICE 5");
    test.send\_message("END 1");
    test.send\_message("ERROR 2");
    test.send\_message("end");
    test.closing\_all();
    return 0;
\}

```

Test to make sure that the server can receive a message and then send that message back

```

#include "tcp_client.h"
using namespace std;

int main()
{
    client test;
    client test2("127.0.0.1",5555);
    string temp="follow";
    while (temp!="end")
    {
        temp=test.receive_message();
        cout<<temp<<endl;
        sleep(1);
        test2.send_message(temp);
    }
    test.closing_all();
}

```

```
return 0;
}
```

I used the sleep command because both the server and the client are running synchronously which mean that when the server tries to send back the message there is no one listening.

We can simply just run this two functions on two threads which is simpler.

### 8.1.3 Makefile

```
all:client server
client:tcp\_client.h testing\_client.cpp
g++ -g testing\_client.cpp -o client
server:tcp\_client.h testing\_server.cpp
g++ -g testing\_server.cpp -o server
test: all
./server >test.txt | ./client
clean:
rm client server
```

### 8.1.4 Results

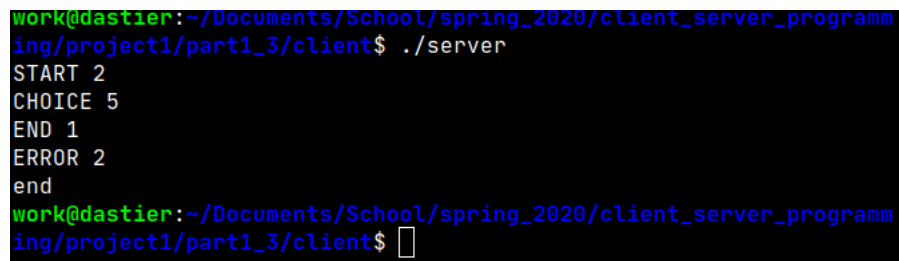
A terminal window with a black background and green text. The prompt is 'work@dastier:~/Documents/School/spring\_2020/client\_server\_programming/project1/part1\_3/client\$'. The user has entered './server'. The output is: 'START 2', 'CHOICE 5', 'END 1', 'ERROR 2', 'end'. The prompt is now 'work@dastier:~/Documents/School/spring\_2020/client\_server\_programming/project1/part1\_3/client\$' with a cursor.

Figure 5: Results on the server side of the message from the client

A terminal window with a black background and green text. The prompt is 'work@dastier:~/Documents/School/spring\_2020/client\_server\_programming/project1/part1\_3/client\$'. The user has entered './server2'. The output is: 'START 2', 'CHOICE 5', 'END 1', 'ERROR 2', 'end'. The prompt is now 'work@dastier:~/Documents/School/spring\_2020/client\_server\_programming/project1/part1\_3/client\$' with a cursor.

Figure 6: Results of the server side of the message send from the client which is then sent back to the client

## 8.2 Client

### 8.2.1 Header File

Is the same as the server

### 8.2.2 Test Program

This program simply test if the client can send messages

```
\#include "tcp\_client.h"
using namespace std;
int main()
```

```

\{
client test;
string temp="follow";
while (temp!="end")
\{
temp=test.receive\_message();
cout<<temp<<endl;
\}
test.closing\_all();
return 0;
\}

```

This program simply test that I can send a message to the server and then the server will return it back

```

#include "tcp_client.h"
using namespace std;

int main()
{
client test;
client test2("127.0.0.1",5555);
string temp;
test.send_message("START 2");
temp=test2.receive_message();
cout<<temp<<endl;

test.send_message("CHOICE 5");
temp=test2.receive_message();
cout<<temp<<endl;

test.send_message("END 1");
temp=test2.receive_message();
cout<<temp<<endl;

test.send_message("ERROR 2");
temp=test2.receive_message();
cout<<temp<<endl;

test.send_message("end");
temp=test2.receive_message();
cout<<temp<<endl;

test.closing_all();
return 0;
}

```

### 8.2.3 Makefile

```

all:client server
client:tcp\_client.h testing\_client.cpp
g++ -g testing\_client.cpp -o client
server:tcp\_client.h testing\_server.cpp
g++ -g testing\_server.cpp -o server
test: all
./server >test.txt | ./client
clean:
rm client server

```

#### 8.2.4 Results

```
work@dastier:~/Documents/School/spring_2020/client_server_programming/project1/part1_3/client$ ./server
START 2
CHOICE 5
END 1
ERROR 2
end
work@dastier:~/Documents/School/spring_2020/client_server_programming/project1/part1_3/client$
```

Figure 7: Results on the server side of the message from the client

```
work@dastier:~/Documents/School/spring_2020/client_server_programming/project1/part1_3/client$ ./client2
START 2
CHOICE 5
END 1
ERROR 2
end
work@dastier:~/Documents/School/spring_2020/client_server_programming/project1/part1_3/client$
```

Figure 8: Results of the message send to the server from the client which was sent back to to the client