

**Computer Network Programming**  
**Project report of the team number 1**

**Grzegorz Rypeść**  
**김소빈 (Kim So Bin)**  
**김문정 (Kim Mun Jeong)**  
**Guillaume Tabard**

## 1. Overview

Our application is an attendance checking system written in C programming language. It consists of 3 files:

- client application (client.c)
- server application (server.c)
- makefile script, that is used to automate compilation process.

To compile the program go to directory where these files are located and run “make”. From now on you can run them with “./server” and “./client” commands. The output of the program will be displayed on a console and in a file called “attendanceList.txt”.

## 2. What does our application do?

After the server is turned on, it will wait for a random amount of time (for test purposes from 5 to 10 seconds) until it starts accepting connections. Then a client can connect and send his/her name and a surname. This information will be stored in a list implemented on a server. The system checks whether client's input is correct or not. Multiple client connections can be handled thanks to creating a new thread for each one of them. Every new name and surname are added to the list and before server shuts down it saves the list to a “.txt” file. Server will accept connections only for a certain time that is defined by macro ACCEPTINGTIME. We handle signals and interruptions to provide functionalities that are not possible without them, like stopping accept function after given time.

## 3. Server

The system requires exactly 1 server to be run. In server.c file we implemented a list that consists of nodes of type “struct Node”. Each such a node contains name, surname, and a pointer to next node. This structure is used to contain information delivered by clients. Server requires a port argument to be run and then it will wait for a random amount of time (for test purposes 5-10 seconds) before it starts accepting connections. After this time server starts accepting connections from clients and store the data in the list. When a connection arrives a new thread is created that will handle the client. There are 2 ways to stop the server – either send a killing signal or wait for the timer to run out of time. In both cases interrupts are handled and the list is saved to a .txt file.

The file server.c consists of 7 core functions:

- void addToList(char data[]) - This function converts argument string into 2 strings that are: name and surname. This can be done because there is a space character between name and a surname. Then the function creates a new node, fills it with name and surname. What's more it prepends the node to the list.
- void printList() - Prints the list to the console.
- void saveList() - Saves the list to a file with extension of .txt so that it looks very similar to output of printList() function.
- void alert() - This function is called when SIGALRM signal appears in the program. This signal appears when the time is up and system no longer accepts connections. The function alert() saves the list to a file and exits the program.

- `void *threadBody(void *socket_desc)` - This is a definition of each thread that starts when connection to a client is established. It receives data from a client and stores it in `message[]` string. It checks if the message makes sense and adds it to the list. Because many threads can work in parallel I created a semaphore named `mutex` that synchronizes access of threads to the list and removes race condition. Without it, if two threads added new node to the list simultaneously there would be a danger of creating a wrong connection in the list, and some data could be lost. Therefore adding to list is placed in a critical section. At the end the thread sends response to client to inform him/her if the attendance was successful, then it exits.
- `void handleInterruption(int signalType)` - Handles interruptions that come to the program by saving the list and ending the program.
- `int main(int argc, char *argv[])` - Checks if the program arguments are right, creates a socket, binds an address to it, starts listening, calls all above functions and creates new threads when clients connect.

#### **4. Client**

Every student who wants to confirm his/her attendance is supposed to run client program when server accepts connections. In all other cases the connection will fail. Client application has 2 arguments - address of the server and a port number. It consists of 2 functions:

- `int checkName(char data[])` - Checks if message entered by the user is correct.
- `int main(int argc, char *argv[])` - Checks if arguments are correct, gets input from user, creates socket, connects to server, binds an address to the socket, connects, sends message to the server and waits for response. At the end it prints the response to the console.

#### **5. Problems encountered**

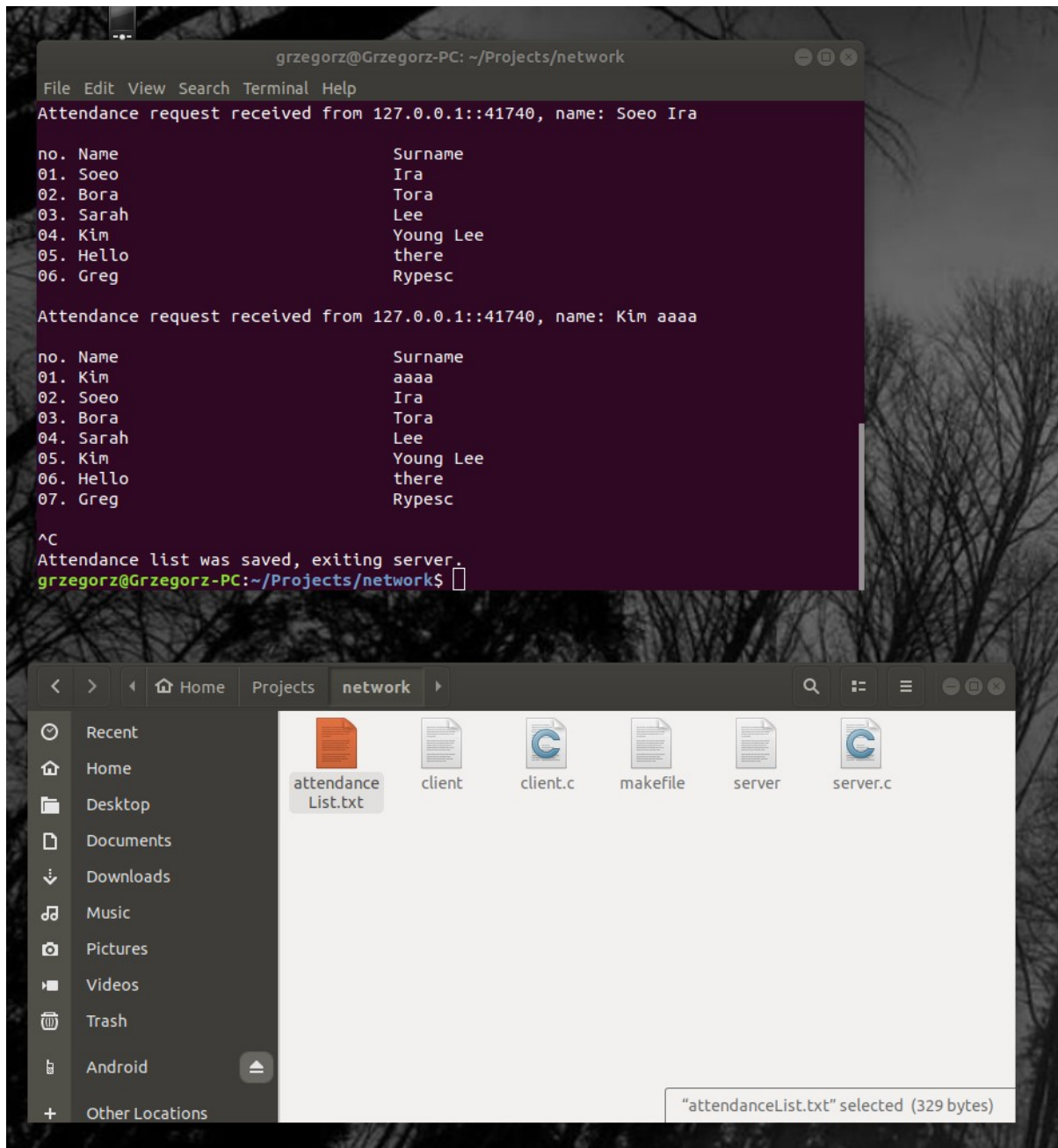
In first version of the program it didn't use threads nor fork. Therefore only one client could be handled at the time when others had to wait in a queue. We decided to fix this by forking other processes but this didn't work. It was so due to a fact that forked processes do not share the same memory addresses, therefore we couldn't add client message to the list. We fixed that by implementing threads that do share the same memory.

#### **6. Testing**

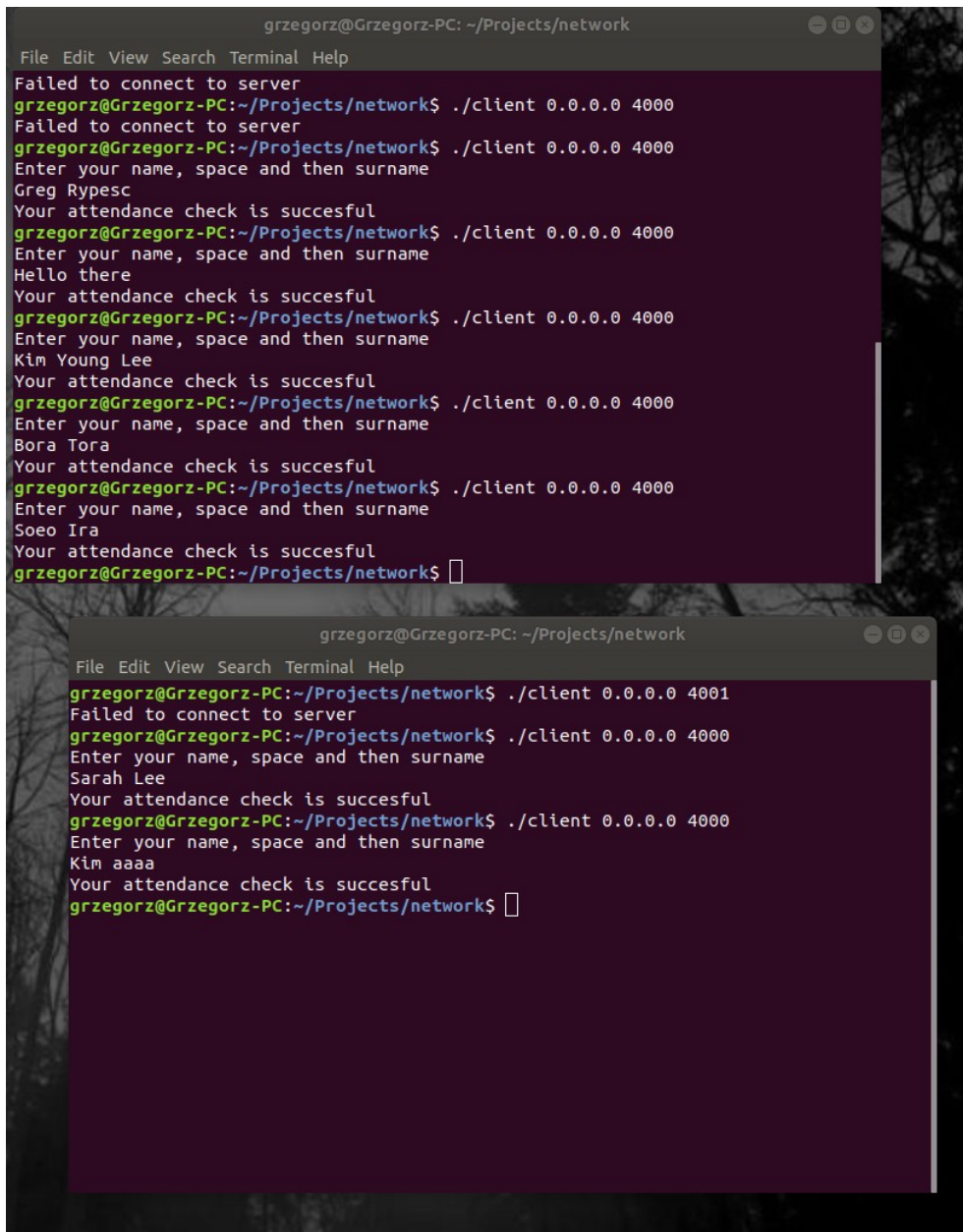
The system in final version was tested many times and we didn't find any bugs.

#### **7. Screenshots and running**

We attached a video showing how our application works. What's more here we provided screenshots that show system's server handling attendance checks by clients, even in parallel and saving student list to a file.



Here we can see the server accepting attendances and displaying the list. At the end it was closed and the list was exported to attendanceList.txt file in project's directory.



```
grzegorz@Grzegorz-PC: ~/Projects/network
File Edit View Search Terminal Help
Failed to connect to server
grzegorz@Grzegorz-PC:~/Projects/network$ ./client 0.0.0.0 4000
Failed to connect to server
grzegorz@Grzegorz-PC:~/Projects/network$ ./client 0.0.0.0 4000
Enter your name, space and then surname
Greg Rypesc
Your attendance check is succesful
grzegorz@Grzegorz-PC:~/Projects/network$ ./client 0.0.0.0 4000
Enter your name, space and then surname
Hello there
Your attendance check is succesful
grzegorz@Grzegorz-PC:~/Projects/network$ ./client 0.0.0.0 4000
Enter your name, space and then surname
Kim Young Lee
Your attendance check is succesful
grzegorz@Grzegorz-PC:~/Projects/network$ ./client 0.0.0.0 4000
Enter your name, space and then surname
Bora Tora
Your attendance check is succesful
grzegorz@Grzegorz-PC:~/Projects/network$ ./client 0.0.0.0 4000
Enter your name, space and then surname
Soeo Ira
Your attendance check is succesful
grzegorz@Grzegorz-PC:~/Projects/network$

grzegorz@Grzegorz-PC: ~/Projects/network
File Edit View Search Terminal Help
grzegorz@Grzegorz-PC:~/Projects/network$ ./client 0.0.0.0 4001
Failed to connect to server
grzegorz@Grzegorz-PC:~/Projects/network$ ./client 0.0.0.0 4000
Enter your name, space and then surname
Sarah Lee
Your attendance check is succesful
grzegorz@Grzegorz-PC:~/Projects/network$ ./client 0.0.0.0 4000
Enter your name, space and then surname
Kim aaaa
Your attendance check is succesful
grzegorz@Grzegorz-PC:~/Projects/network$
```

Here clients connect to server, send and receive messages in parallel. They do not interfere with each other because they are handled by different threads.

We also included a video file that shows how our application works.