Technical Report

Paicu Eliza

Facultatea de Informatică, Universitatea "Alexandru Ioan Cuza" din Iași

1 Introduction

The app "Mersul trenurilor" is a client-server app that facilitates communication about real-time updates and upcoming train schedules. It provides functionalities like requesting arrivals and departures, updating train delays and ensuring concurrent client-server communication.

2 Applied Technologies

2.1 Sockets

Sockets provide communication between the client and the server. They provide reliable and easy communication, making them a natural choice.

2.2 TCP (Transmission Control Protocol)

TCP is a connection-oriented protocol. It provides reliable, connection-oriented communication, ensuring data integrity and delivery order. The application requires both requests and updates to be handled without data loss or reordering, making TCP an obvious choice over UDP.

2.3 Multithreading

Multithreading allows the server to handle multiple client connections simultaneously. The use of threads ensures that one client's requests do not block or delay the handling of other clients.

2.4 XML

XML is used to store and retrieve train schedule data in a structured format. The libxml2 library is used for handling XML data. It provides features for parsing XML files, navigating the document tree, querying elements, and modifying the data. Libxml2 ensures efficient and reliable interaction with the XML file.

3 Application Structure

Below is a diagram illustrating the architecture of the system:

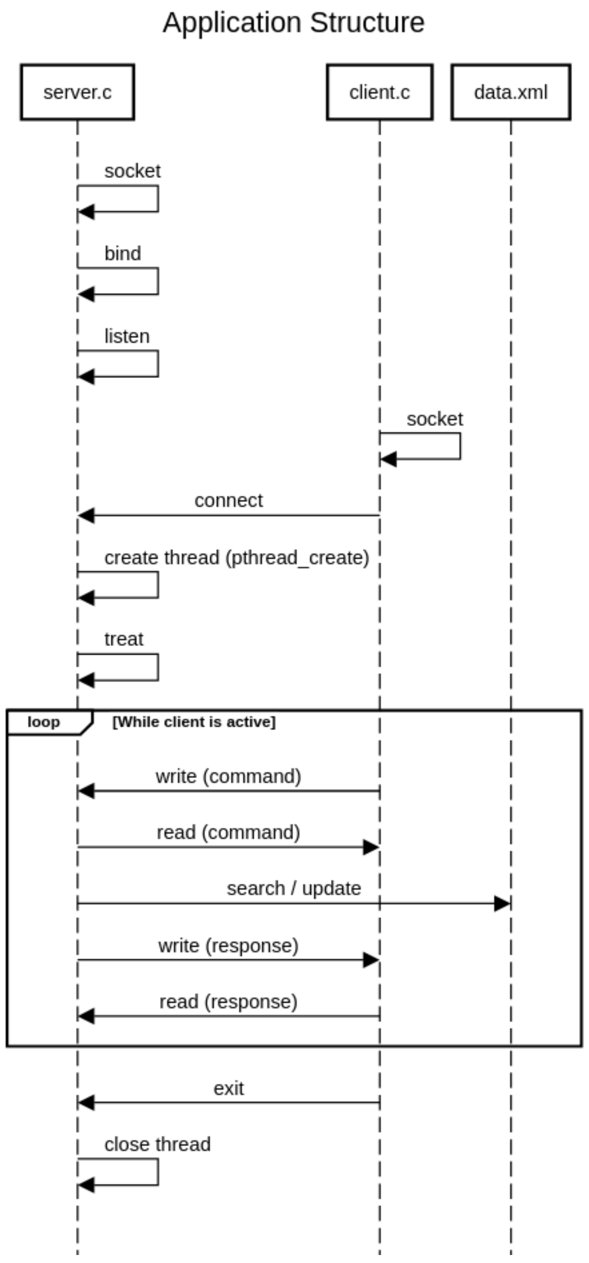


Fig. 1. A diagram representing the structure of the application.

The server initializes and begins listening for incoming client connections. When a client connects, the server creates a new thread to handle the interaction. The client sends commands and the server processes these commands, interacts with the data, and sends the response back. The client exits the session with the exit command.

4 Implementation Aspects

The server is concurrent and creates a thread for every new client.

```c
while (1) {
    int* client = malloc(sizeof(int));
    int length = sizeof(from);

    if ((*client = accept (sd, (struct sockaddr *) &from, &length)) < 0)
     {perror ("Accept error\n");}

    pthread_create(&th[i], NULL, &treat, client);

    i++;
}
```

The *treat* function is the one that receives commands, sends responses based on them and calls other functions that will actually implement the command.

```c
void* treat(void* arg) {
    int client = *(int*)arg;
    char command[100] = {0}, response[1000] = {0};

    while (1) {
        ssize_t size = read (client, command, sizeof(command) - 1);
        if (size < 0) {perror ("Read error\n");}
        else {command[size] = '\0';}

        if (!strcmp(command, "request_daily_arrivals")) {send_data(
    client, 0);}
        else if (!strcmp(command, "request_daily_departures")) {
    send_data(client, 1);}
        else if (!strcmp(command, "request_hourly_arrivals")) {send_data
    (client, 2);}
        else if (!strcmp(command, "request_hourly_departures")) {
    send_data(client, 3);}
        else if (!strcmp(command, "update_train")) {update_data(client)
    ;}
        else if (!strcmp(command, "exit")) {
            strcpy(response, "Exiting the client application...\n");
            if (write (client, response, strlen(response) * sizeof(char)
    ) <= 0) {perror ("Write error\n");}
            printf("Client %d has exited\n", client);
            return NULL;
        }
        else {
            strcpy(response, "Valid commands: request_daily_arrivals,
    request_daily_departures, request_hourly_arrivals,
    request_hourly_departures, update_train, exit\n");
            if (write (client, response, strlen(response) * sizeof(char)
    ) <= 0) {perror ("Write error\n");}
            continue;
        }

        printf("Client %d sent the following command: %s\n", client,
    command);
    }
}
```

The functions called by the *treat* function are responsible with asking for further parameters. For example, in the *send_data* function, the server asks for the station and saves it in a string.

```c
void send_data(int client, int mode) {
    char command[100] = {0}, response[1000] = {0};
    char station[100] = {0};

    strcpy(response, "Station: ");
    if (write (client, response, strlen(response) * sizeof(char)) <= 0) {perror ("Write error\n");}

    ssize_t size = read (client, station, sizeof(station) - 1);
    if (size < 0) {perror ("Read error\n");}
    else {station[size] = '\0';}

    find_trains_at_station(station, mode, client);
}
```

After that, the function calls another function, *find_trains_at_station*, that will actually parse the XML file.

```c
void find_trains_at_station(char *target_station, int mode, int client) {
    reset_delays();

    char response[5000] = {0};
    char train_info[50][500] = {0};
    int index = 0;

    int hour, minute;
    get_current_time(&hour, &minute);

    if (mode == 2) {
        strcpy(response, "Arrivals in the next hour to ");
        strcat(response, target_station);
        strcat(response, ":\n");
    }
    else if (mode == 3) {
        strcpy(response, "Departures in the next hour from ");
        strcat(response, target_station);
        strcat(response, ":\n");
    }
    else if (mode == 0) {
        strcpy(response, "Arrivals in the next 24 hours to ");
        strcat(response, target_station);
        strcat(response, ":\n");
    }
    else if (mode == 1) {
        strcpy(response, "Departures in the next 24 hours from ");
        strcat(response, target_station);
        strcat(response, ":\n");
    }

    xmlDoc *doc = xmlReadFile(FILE, NULL, 0);
    if (doc == NULL) {perror("Error with the XML file\n");}

    xmlNode *root = xmlDocGetRootElement(doc);
    xmlChar *destination;

    for (xmlNode *train = root->children; train; train = train->next) {
        if (train->type == XML_ELEMENT_NODE && strcmp((char *)train->name, "train") == 0) {
            xmlChar *delay_prop = xmlGetProp(train, (const xmlChar *)"delay");
            int delay = atoi((char *)delay_prop);
        xmlChar *train_name = xmlGetProp(train, (const xmlChar *)"id");

        for (xmlNode *station = train->children; station; station = station->next) {
            if (station->type == XML_ELEMENT_NODE && !strcmp((char *)station->name, "station")) {
                xmlChar *station_name = xmlNodeGetContent(station);
                if (station_name)
                    destination = station_name;
            }
        }

        for (xmlNode *station = train->children; station; station = station->next) {
            if (station->type == XML_ELEMENT_NODE && !strcmp((char *)station->name, "station")) {
                xmlChar* station_name = xmlNodeGetContent(station);

                if (strcmp((char *)station_name, target_station) == 0) {
                    xmlChar *arrival_time = xmlGetProp(station, (const xmlChar *)"arrival");
                    xmlChar *departure_time = xmlGetProp(station, (const xmlChar *)"departure");
                    if (mode == 2 && arrival_time) {
                        int arrival_hours, arrival_minutes;
                        convert_time(arrival_time, &arrival_hours, &arrival_minutes);

                        if (arrival_hours == 0 && hour == 23) arrival_hours = 24;

                        if (arrival_hours * 60 + arrival_minutes >= hour * 60 + minute && arrival_hours * 60 + arrival_minutes < hour * 60 + minute + 60) {
                            sprintf(train_info[index], "Scheduled arrival: %s, Delay: %d minute(s), Train: %s, Destination: %s\n", arrival_time, delay, train_name, destination);
                            index++;
                        }
                    }
                    else if (mode == 3 && departure_time) {
                        int departure_hours, departure_minutes;
                        convert_time(departure_time, &departure_hours, &departure_minutes);

                        if (departure_hours == 0 && hour == 23) departure_hours = 24;

                        if (departure_hours * 60 + departure_minutes >= hour * 60 + minute && departure_hours * 60 + departure_minutes < hour * 60 + minute + 60) {
                            sprintf(train_info[index], "Scheduled departure: %s, Delay: %d minute(s), Train: %s, Destination: %s\n", departure_time, delay, train_name, destination);
                            index++;
                        }
                    }
                    else if (mode == 0 && arrival_time) {
                        sprintf(train_info[index], "Scheduled arrival: %s, Delay: %d minute(s), Train: %s, Destination: %s\n", arrival_time, delay, train_name, destination);
                        index++;
                    }
                    else if (mode == 1 && departure_time) {
                            sprintf(train_info[index], "Scheduled departure: %s, Delay: %d minute(s), Train: %s, Destination: %s\n", departure_time, delay, train_name, destination);
                            index++;
                        }
                    }

                    xmlFree(station_name);
                }
            }

            xmlFree(train_name);
        }
    }

    xmlFreeDoc(doc);
    xmlFree(destination);
    xmlCleanupParser();

    char aux;
    for (int i = 0; i < index - 1; i++)
        for (int j = i + 1; j < index; j++)
            if (strcmp(train_info[i], train_info[j]) > 0)
                for (int k = 0; train_info[i][k] || train_info[j][k]; k++)
                    aux = train_info[i][k], train_info[i][k] = train_info[j][k], train_info[j][k] = aux;

    for (int i = 0; i < index; i++)
        strcat(response, train_info[i]);

    if (index == 0)
        strcat(response, "No trains found\n");

    if (write (client, response, strlen(response) * sizeof(char)) <= 0) {perror ("Write error\n");}
}
```

As we can see, this also handles the situation in which the current hour is 23, because in that case it changes the hour 0 to 24 so the client can still receive arrivals/departures in the next hour even if they are past midnight. We are also sorting the lines based on the time, to make it easier to read by the client.

The *convert_time* function extracts the hours and minutes from an xmlChar pointer into integers. Somewhat similarly, the *get_current_time* function extracts the current hours and minutes. The *reset_delays* function checks the trains which have arrived for at least an hour, including the delay, and resets the delay to 0.

The *update_data* function works somewhat similarly to *send_data*. It asks the client for two parameters: the train id and the delay in minutes. After that, it calls on another function, *update_train_delay*, that will actually update the XML file.

```
int update_train_delay(char *train_id, char *delay) {
    pthread_mutex_t lock;
    pthread_mutex_init(&lock, NULL);

    xmlDoc *doc = xmlReadFile(FILE, NULL, 0);
    if (doc == NULL) {perror("Error with the XML file\n");}

    xmlNode *root = xmlDocGetRootElement(doc);

    xmlNode *target_train = NULL;
```

```c
    for (xmlNode *train = root->children; train; train = train->next) {
        if (train->type == XML_ELEMENT_NODE && strcmp((char *)train->name, "train") == 0) {
            xmlChar *id = xmlGetProp(train, (const xmlChar *)"id");
            if (id && strcmp((char *)id, train_id) == 0) {
                target_train = train;
                xmlFree(id);
                break;
            }
            xmlFree(id);
        }
    }

    if (target_train == NULL) {return 1;}

    xmlSetProp(target_train, (const xmlChar *)"delay", (const xmlChar *)delay);

    pthread_mutex_lock(&lock);
    if (xmlSaveFile(FILE, doc) == -1) {perror("Failed to update the XML file\n");}
    pthread_mutex_unlock(&lock);

    xmlFreeDoc(doc);
    xmlCleanupParser();

    pthread_mutex_destroy(&lock);

    return 0;
}
```

As we can see, the function returns an error code in case the train id is invalid. Even though it is unlikely that two threads will try to update the XML file at the exact same time, it is better to err on the side of caution and use a mutex.

The XML file used in the project represents the schedule of trains and their respective stations, including details like arrival and departure times, train IDs and delays. The structure is shown in the following example:

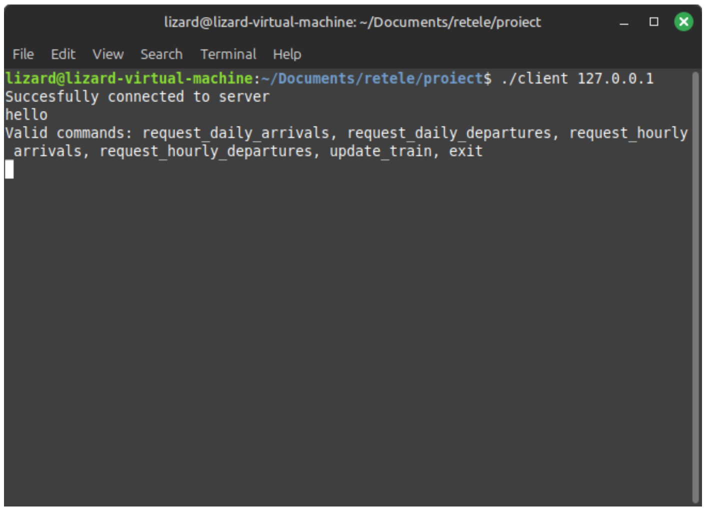
```xml
<train delay="20" id="IR1662">
    <station departure="14:20">Iasi</station>
    <station arrival="14:24" departure="14:25">Nicolina</station>
    <station arrival="15:06" departure="15:07">Buhaiesti</station>
    <station arrival="15:26" departure="15:28">Vaslui</station>
    <station arrival="15:44" departure="15:45">Crasna</station>
    <station arrival="16:17" departure="16:19">Barlad</station>
    <station arrival="16:57" departure="17:12">Tecuci</station>
    <station arrival="17:55" departure="17:57">Focsani</station>
    <station arrival="18:39" departure="18:40">Ramnicu Sarat</station>
    <station arrival="19:08" departure="19:09">Buzau</station>
    <station arrival="19:43" departure="19:44">Mizil</station>
    <station arrival="20:24" departure="20:25">Ploiesti Sud</station>
    <station arrival="21:03">Bucuresti Nord</station>
</train>
```

The XML file includes schedules for four such trains. The train schedules used in this XML file are actual data sourced from the CFR website. This ensures that the project uses realistic data, making it applicable to real-world scenarios.

Now, let's look at how the application behaves.

After starting the server, we can start a client, and we can give the server's IP address as a parameter. Otherwise, it will ask for the IP address.

We can enter commands from the client. The valid commands are: *request_ daily_ arrivals, request_ daily_ departures, request_ hourly_ arrivals, request_ hourly_ departures, update_ train, exit*. If we try to use any other command, the server will remind us of the valid commands.


lizard@lizard-virtual-machine: ~/Documents/retele/proiect
File Edit View Search Terminal Help

```
lizard@lizard-virtual-machine:~/Documents/retele/proiect$ ./client 127.0.0.1
Succesfully connected to server
hello
Valid commands: request_daily_arrivals, request_daily_departures, request_hourly
_arrivals, request_hourly_departures, update_train, exit
```

Fig. 2. A screenshot showing an example of a client connecting to the server and trying an invalid command

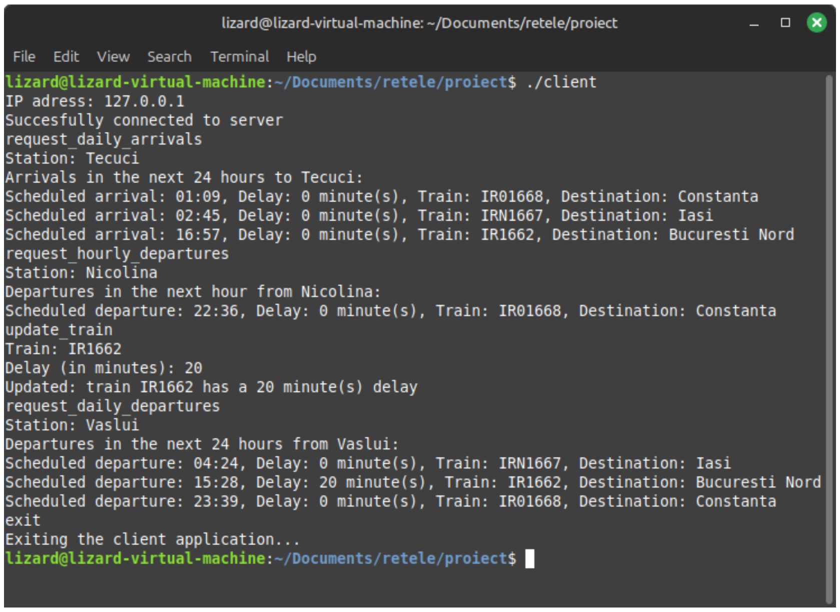
If we use one of the first four commands, the server will ask us which station we are interested in. The server then searches the XML file for the specific station and sends the requested data to the client, or lets the client know that no data was found.

The first two commands, *request_ daily_ arrivals* and *request_ daily_ departures*, will give us data about the trains arriving or departing in the next 24 hours, whereas the next two commands, *request_ hourly_ arrivals* and *request_ hourly_ departures*, will give us data about the trains arriving or departing in the next hour.

For the next command, *update_ train*, the server will ask for the train that is late, and the delay in minutes. The server then modifies the XML file to account for the delay, or lets the client know if the train is invalid.

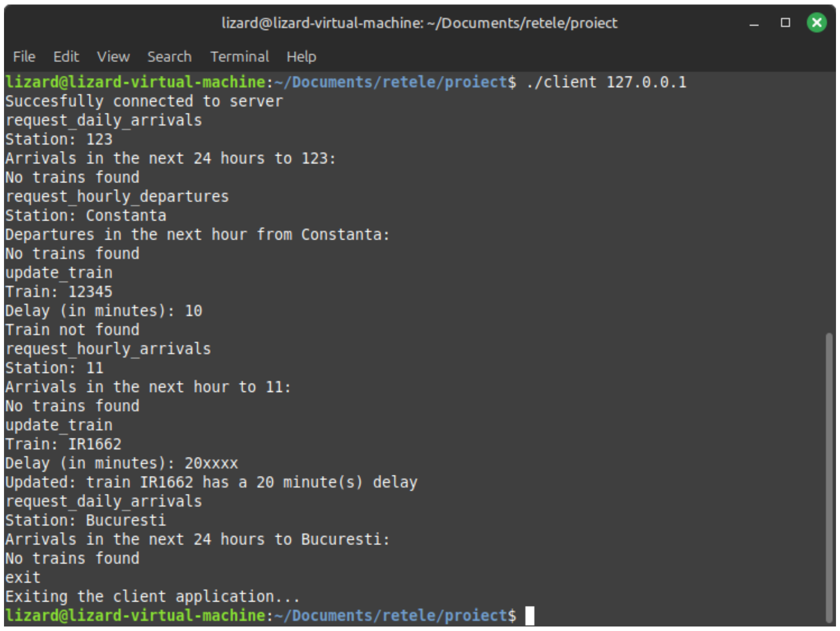
If we use the *exit* command, the client application will be simply shut down.

Each time we send a valid command to the server, the server will display it, along with the client that sent it.



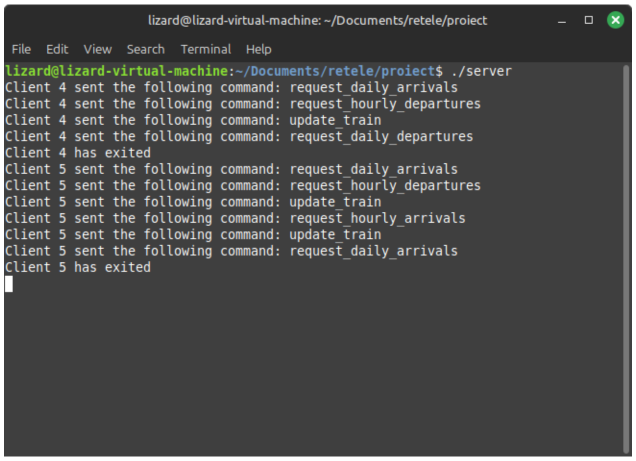
```
lizard@lizard-virtual-machine:~/Documents/retele/proiect$ ./client
IP adress: 127.0.0.1
Succesfully connected to server
request_daily_arrivals
Station: Tecuci
Arrivals in the next 24 hours to Tecuci:
Scheduled arrival: 01:09, Delay: 0 minute(s), Train: IR01668, Destination: Constanta
Scheduled arrival: 02:45, Delay: 0 minute(s), Train: IRN1667, Destination: Iasi
Scheduled arrival: 16:57, Delay: 0 minute(s), Train: IR1662, Destination: Bucuresti Nord
request_hourly_departures
Station: Nicolina
Departures in the next hour from Nicolina:
Scheduled departure: 22:36, Delay: 0 minute(s), Train: IR01668, Destination: Constanta
update_train
Train: IR1662
Delay (in minutes): 20
Updated: train IR1662 has a 20 minute(s) delay
request_daily_departures
Station: Vaslui
Departures in the next 24 hours from Vaslui:
Scheduled departure: 04:24, Delay: 0 minute(s), Train: IRN1667, Destination: Iasi
Scheduled departure: 15:28, Delay: 20 minute(s), Train: IR1662, Destination: Bucuresti Nord
Scheduled departure: 23:39, Delay: 0 minute(s), Train: IR01668, Destination: Constanta
exit
Exiting the client application...
lizard@lizard-virtual-machine:~/Documents/retele/proiect$
```

Fig. 3. A screenshot showing an example of commands being used by a client.



```
lizard@lizard-virtual-machine:~/Documents/retele/proiect$ ./client 127.0.0.1
Succesfully connected to server
request_daily_arrivals
Station: 123
Arrivals in the next 24 hours to 123:
No trains found
request_hourly_departures
Station: Constanta
Departures in the next hour from Constanta:
No trains found
update_train
Train: 12345
Delay (in minutes): 10
Train not found
request_hourly_arrivals
Station: 11
Arrivals in the next hour to 11:
No trains found
update_train
Train: IR1662
Delay (in minutes): 20xxxx
Updated: train IR1662 has a 20 minute(s) delay
request_daily_arrivals
Station: Bucuresti
Arrivals in the next 24 hours to Bucuresti:
No trains found
exit
Exiting the client application...
lizard@lizard-virtual-machine:~/Documents/retele/proiect$
```

Fig. 4. A screenshot showing another example of commands being used by a client.


lizard@lizard-virtual-machine: ~/Documents/retele/proiect
File Edit View Search Terminal Help

```
lizard@lizard-virtual-machine:~/Documents/retele/proiect$ ./server
Client 4 sent the following command: request_daily_arrivals
Client 4 sent the following command: request_hourly_departures
Client 4 sent the following command: update_train
Client 4 sent the following command: request_daily_departures
Client 4 has exited
Client 5 sent the following command: request_daily_arrivals
Client 5 sent the following command: request_hourly_departures
Client 5 sent the following command: update_train
Client 5 sent the following command: request_hourly_arrivals
Client 5 sent the following command: update_train
Client 5 sent the following command: request_daily_arrivals
Client 5 has exited
```

Fig. 5. A screenshot showing what is printed on the screen from the server when we execute the clients in Fig. 3 and Fig. 4.

5 Conclusions

The development of this project has been both a rewarding and challenging experience. It required close attention to multiple aspects, including the communication protocols and efficient handling of concurrency. The use of TCP, sockets and multithreading provided the necessary foundation for reliable, real-time communication between clients and the server. However, implementing these technologies in C posed a significant challenge.

There is ample room for improvement and expansion. For example, the communication protocol could be expanded to support more detailed queries and responses, or a graphical user interface could be developed to provide a more intuitive and user-friendly interaction.

Overall, this project has been a valuable learning experience.

References

1. Geeks for Geeks. *Differences Between TCP and UDP.* `https://www.geeksforgeeks.org/differences-between-tcp-and-udp/`
2. The Computer Networks course. *A concurrent server that creates a thread for each connected client.* `https://edu.info.uaic.ro/computer-networks/files/NetEx/S12/ServerConcThread/servTcpConcTh2.c`. `https://edu.info.uaic.ro/computer-networks/files/NetEx/S12/ServerConcThread/cliTcpNr.c`
3. YouTube. *Unix Threads in C* `https://youtube.com/playlist?list=PLfqABt5AS4FmuQf70psXrsMLEDQXNkLq2&si=c1AhuJList7D2NDV`
4. Sequence Diagram. *Instructions and Examples* `https://sequencediagram.org/instructions.html`
5. Mersul trenurilor CFR. *Trenul meu* `https://mersultrenurilor.infofer.ro/ro-RO/Trains`