Project - Traffic Monitoring

Bianca Mariciuc (2E1)

1 Introduction

Traffic Monitoring is an application designed to make traffic more organized and efficient, ensuring smoother journeys for drivers. Drivers are allowed to connect and receive traffic assistance. The main purpose for this application is to make the driver's life easier during driving. Drivers can receive information on the weather, fuel prices at nearby stations, and various sports events that are happening in the city they are driving through. Also, in this application, drivers can alert and be alerted by other registered users about accidents. In addition, drivers can receive warnings based on their speed, which is automatically updated every minute during their journey, helping them drive more safely and responsibly.

2 Applied Technologies

For this project, I chose to implement a concurrent **TCP (Transmission Control Protocol)** server that creates a thread for each connected client. I chose this protocol over others because TCP ensures accurate information transmission by establishing a virtual connection between the sender and receiver, which is crucial for my project, since drivers rely on precise and reliable data. Moreover, the implementation must be concurrent because the application needs to support multiple drivers simultaneously. This issue can be solved by creating a separate thread for each connected client, in this way the server can manage each client independently. Also, threads share the same memory space and resources of the parent process, making them more lightweight compared to processes.

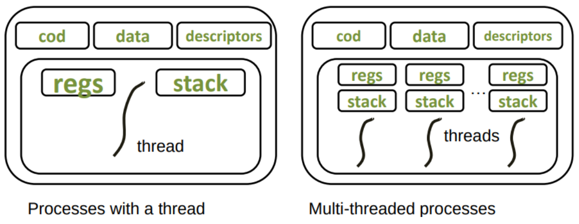

Figure 1: Threads share the memory space and resources of the parent process but maintain their own stack and registers.

3 Application Structure

The structure of the application is quite simple. Drivers are considered clients that communicate with the application server. Every client must follow a specific logic when requesting information from the server, this means that the client cannot receive any data unless they are logged in or registered. During a session, every client is allowed to send requests and wait for a response. Below is a diagram illustrating how communication between the client and server is made.

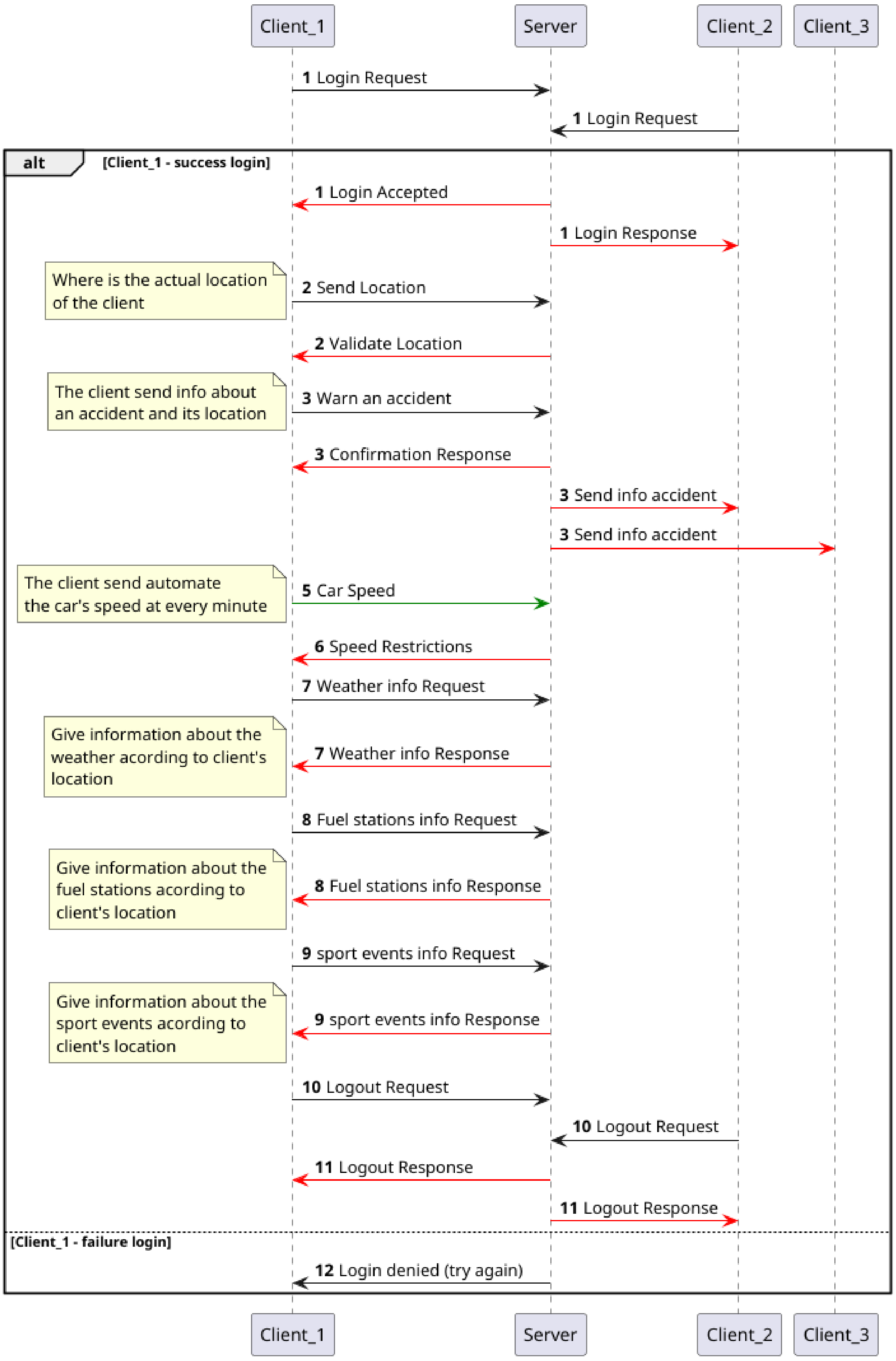
Client_1
Server
Client_2
Client_3
1 Login Request
1 Login Request
alt
[Client_1 - success login]
1 Login Accepted
1 Login Response
Where is the actual location of the client
2 Send Location
2 Validate Location
The client send info about an accident and its location
3 Warn an accident
3 Confirmation Response
3 Send info accident
3 Send info accident
The client send automate the car's speed at every minute
5 Car Speed
6 Speed Restrictions
7 Weather info Request
Give information about the weather acording to client's location
7 Weather info Response
8 Fuel stations info Request
Give information about the fuel stations acording to client's location
8 Fuel stations info Response
9 sport events info Request
Give information about the sport events acording to client's location
9 sport events info Response
10 Logout Request
10 Logout Request
11 Logout Response
11 Logout Response
[Client_1 - failure login]
12 Login denied (try again)
Client_1
Server
Client_2
Client_3

Figure 2: Sequence Diagram - Requests from points 1 to 3 must be executed in order, while requests from points 4 to 9 can be made in any order.

The diagram shows that certain information must be stored to provide a response to a request. To fix this problem, we will use a database to store all the necessary information.

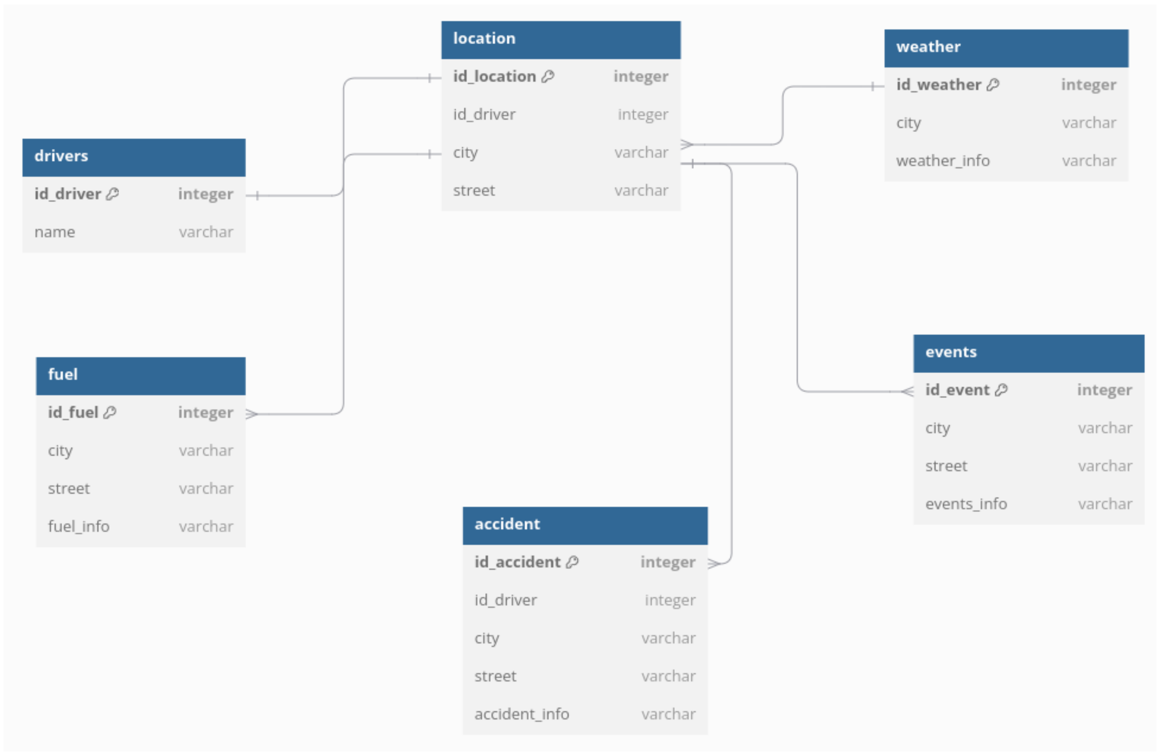

Figure 3: Information Organized In A Database

In this diagram we have 6 tables: drivers, location, events, weather, fuel and accident. The "drivers" table stores client information. The "location" table stores the current location of the client. The "events" table contains details about events occurring in a city. The "weather" table provides weather data for each city. The "fuel" table contains information about fuel stations in a city, and the "accident" table lists active accidents registered in specific city.

4 Implementation Aspects

For implementation I have used the **TCP** protocol with a concurrent server that creates a thread for each connected client and for databse I have used **SQLite**.

First of all, each table is represented as an object (e.g., for the table "*location*" I have implemented a *Location.h* and *Location.cpp*) with specific methods and attributes. Below is a function that creates the tables using an *sql* file where the commands are stored to create specific tables.

```
sqlite3 *db;
void create_tabel() {
    char *err_msg = new char[100];

    if (sqlite3_open("tabels.db", &db)) {
        std::cerr << "Can't open database: "
            << sqlite3_errmsg(db) << std::endl;
        exit(1);
    }

    char *sql = read_from_file("./sql/server_data_creation.sql");

    if (sqlite3_exec(db, sql, nullptr, nullptr, (char **)
    &err_msg) != SQLITE_OK) {
        std::cerr << "SQL error: " << err_msg << std::endl;
        sqlite3_free(err_msg);
    }

    delete_data_from_tabels();
    populate_tabels();

    delete[] sql;
}
```

For this project, I had to handle three types of commands: the first type is a regular request from the client, followed by a response from the server. The second type is a remark from a client that must be broadcast to all connected clients. The third type is when the server, without any client request, sends a message to all connected clients every minute.

Each command, along with all the relevant information about it, is stored in a structure.

```
typedef struct {
    bool subscription;
    char cmd[CMD_SIZE];
    char city[CITY_SIZE];
    char street[STREET_SIZE];
    char city_acc[CITY_SIZE];
    char street_acc[STREET_SIZE];
    char name[NAME_SIZE];
    int socket_id;
} Command;
```

For the first type of command, the process is quite simple: the client sends the type of command, and the server generates a response and sends it back. Here is an example for "login" command:

```
//server.cpp

void login(Command *c, char r[500]) {
    Person p;
    if (p.searchByName(c->name, db) && !find_client_logged(c->name))
    {
        strcpy(r, "You are logged ");
        strcat(r, c->name);
        strcat(r, ".\nWould you like a subscription for more
        information about weather, events and fuel stations?");

        if (!find_client_logged(c->name))
            logged.push_back(c->name);
    } else if (!p.searchByName(c->name, db)
    && !find_client_logged(c->name)) {
        strcpy(r, "You are not register ");
        strcat(r, c->name);
        strcat(r, ".  You can register with command 'sign-up:name'");
    } else {
        strcpy(r, "You are already logged ");
        strcat(r, c->name);
    }
}
```

For the second type of command, when a client reports an accident, all connected clients must be notified immediately. This situation can be handled by promptly sending (from server to client) the response to all connected clients and creating a separate thread to handle receiving the message (in client). Without this approach, the message would be blocked by the *read* primitive, which is a blocking operation (it waits until a message is received).

```
//client.cpp

//create the thread in main, after connection
 pthread_create(&write_thread, nullptr, handle_write, NULL);

//function for handling writing
 void *handle_write(void *arg) {


    while (1) {

        char r[500];

        if (read(socket_des, &r, sizeof(r)) < 0) {
            perror("[client]error read from server.\n");
            return NULL;
        }
        if(printMsj == true) {
            printf("%s\n", r);
        }
    }
}
```

For the third type of command, at every minute the server sends to each connected client (driver) a message regarding their ”speed”. To handle this situation, I have created a thread in client that does this.

```
//client.cpp

//create the thread in main, after connection
pthread_create(&speed_thread, NULL, give_info_speed, NULL);

//function associated
void *give_info_speed(void *arg) {

    srand(time(NULL));
    time_t last_time = time(NULL);
    time_t current_time;
    while (1) {
        current_time = time(NULL);
        if (difftime(current_time, last_time) >= 60) {
            advice_speed();//function to generate and print speed
                            //information
            last_time = current_time;
        }
    }

}
```

5 Conclusion

The Traffic Monitoring application is created to improve traffic organization and make driving easier by giving drivers real-time updates about weather, fuel prices, sports events, and accident alerts. It also helps drivers stay safe by sending speed warnings every minute, encouraging more responsible driving.

The application uses a concurrent TCP server to ensure reliable data transfer and smooth communication between the server and multiple drivers. By assigning each client to its own thread, the server can handle multiple users at the same time without being affected by slower or problematic clients.

To make the application better, adding data encryption could increase security and protect drivers' information. Also, using machine learning for traffic predictions could provide more advanced and helpful advice for drivers.

6 Bibliography

1. https://www.geeksforgeeks.org/what-is-transmission-control-protocol-tcp/
2. https://www.geeksforgeeks.org/thread-in-operating-system/
3. Lecture 7 - Computer Networks UAIC
4. https://plantuml.com/sequence-diagram
5. https://docs.dbdiagram.io/
6. https://www.overleaf.com/learn