Traffic Monitoring
Computer Networks Project

Alexandru Neagu

Faculty of Computer Science
"Alexandru Ioan Cuza" University, Iași

Abstract. This paper aims to describe the implementation of a client-server application designed for traffic monitoring, written in C. The main topics of discussion are: technologies used by the application, communication protocols best suited for this kind of task, the application protocol, implementation details, an in-code representation of a real map using graphs, and the realistic simulation of traffic.

Keywords: TCP · UDP · IPv4 · SQLite3 · Multithreading · BSD Sockets · Graphs · POSIX Threads · Traffic Simulation

1 Introduction

1.1 Motivation

The number of cars has dramatically increased over the last 20 years. There are more and more drivers on the road and the risk of unexpected situations is high. There are several factors a driver should be aware of when on the road. Some of these factors are the speed limit on different parts of the road, recent accidents, streets affected by traffic jams, or nearby police patrols. Thus, a traffic monitoring application should constantly inform the clients if they should lower their travel speed, or if some noticeable events are taking place on the road.

1.2 Application Description

The application described in this paper is based on the **client-server** paradigm. All the drivers that might use this application are seen as clients. The drivers have dynamic traveling speeds and move according to a specified map. The application uses the map of Chișinău in order to simulate the traffic.

The clients might sign up, log in, or request some information by sending specific commands to a specialized process called **server**. These messages are described further in the **Application Protocol** subsection.

The client application realizes the following actions simultaneously:

1. Updates the speed of the car after short periods of time and checks whether the driver's location (street) has changed.
2. Reads a user command from the standard input, sends the command to the server, and receives the response.
3. **Automatically** sends encoded messages to the server regarding the travel speed and the current location of the driver and receives messages about speed limit warnings.
4. **Automatically** sends encoded messages to the server requesting the last notification received from other users and receives it.

On the other hand, the server application simultaneously takes care of the following events that might occur

1. **Concurrently** reads messages from the clients and sends back the response.

1.3 Application Protocol

Following commands might be sent by a client

> **"sign-up"** - The user might create an account on the application. This command will be followed by a prompt, asking for details such as the first name, last name, the desired username, desired password, and whether or not the user wants to subscribe to channels such as sports news, weather condition, and pecos information.
> **"login username"** - Logs the user into the application. This command is followed by a password prompt.
> **"report evcode"** - Reports an event codified by **evcode** that is taking place on the street the user is currently located on. The events are codified as follows:
> evcode = 0 ⇒ Accident
> evcode = 1 ⇒ Traffic Jam
> evcode = 2 ⇒ Road Repair
> evcode = 3 ⇒ Police Patrol
> **"get-police-info"** - Prints information about every police patrol reported by other users.
> **"get-accidents-info"** - Prints information about all the accidents reported by other users.
> **"get-repair-info"** -Prints information about every road that is being repaired.
> **"get-jam-info"** - Prints information about every traffic jam recorded.
> **"get-sports-info option"** - For users who are subscribed to the sports channel, it prints important news about a sport specified by the **option** argument as follows
> option missing ⇒ All Sports
> option = 1 ⇒ Football
> option = 2 ⇒ Basketball
> option = 3 ⇒ Tennis
> **"get-weather-info [date]"** - For users who are subscribed to the weather channel, it prints information about each date in the database. Optionally, a date argument might be provided to get the weather on a specified **date**.
> **"get-peco-info [name]"** - For users who are subscribed to the peco channel, it prints information about all available filling stations. Optionally a name argument might be provided to filter the filling stations available by **name**.
> **"subscribe-peco"** - Subscribe to the peco channel.
> **"subscribe-weather"** - Subscribe to the weather channel.
> **"subscribe-sports"** - Subscribe to the sports channel.
> **"unsubscribe-peco"** - Terminate subscription to the peco channel.
> **"unsubscribe-weather"** - Terminate subscription to the weather channel.
> **"unsubscribe-sports"** - Terminate subscription to the sports channel.
> **"logout"**
> **"quit"** - exit the application

2 Technologies Used

2.1 Multithreading

One possible way of implementing the client concurrency, along with the tasks that should be realized at the same time, is to use the **fork**() primitive and work with multiple related processes. However, this concept uses the copy-on-write mechanism, which might slow the application considerably for a big number of clients. Because of these issues with multiprocessing, the application is using **threads**.

Even though threads share some properties with processes, there is a crucial difference between them:

- Threads are not independent from each other. Each thread shares its code section, data section, and OS resources with other threads.

However, threads do have their own program counter, register set, and stack space. Because of these features, threads are also called "lightweight processes". [4]

In order to use threads in the application, **POSIX Threads** (Pthreads) standard is used.

2.2 TCP/IP

The communication between the server and the client is done using the **TCP/IP** networking framework. TCP/IP organizes the set of communication protocols into the following layers: Application Layer, Transport Layer, Internet Layer, and Link Layer.

The **Transport Layer** establishes channels used for data exchange. This layer provides host-to-host connectivity in the form of end-to-end message transfer services, independent of the underlying network. [5] There are two important protocols used in this layer: **UDP** (connectionless) and **TCP** (connection-oriented). The application uses the TCP protocol because of its reliability. The TCP protocol guarantees the following features [6]

- Data arrives in order.
- Transmitted data is correct.
- Lost packets are resent.
- Duplicate data is discarded.

Establishing a connection between endpoints is a complex process that requires several actions to be completed (i.e. Three Way Handshaking). Thus, the TCP protocol is slower than the UDP protocol. For this application, I have chosen reliability and data correctness over performance. Of course, these features are crucial for such an application.

The **Internet Layer** sends data from the source network to the destination network through the routing process[5]. This layer has the responsibility of sending packets across multiple networks. The main component of this layer is the Internet Protocol (IP), used for identifying the network hosts. Internet Protocol version 4 (IPv4) uses a 32-bit IP address and it was used for a long time. Nowadays, Internet Protocol version 6 (IPv6), which uses 128-bit IP addresses, is widely used. For simplifying reasons, the application uses an IPv4 address system.

2.3 Sockets

For computer network programming, several APIs might be used in order to use the necessary communication protocols. This application uses **BSD Sockets** (also known as POSIX Sockets) API.

A socket is an abstract representation of the local endpoint of a network communication path. It is a general facility, independent of the hardware architecture and the type of data transmission [3]. In the BSD Sockets API, each socket is identified by a **file descriptor**. A socket uses the existing I/O programming interface (similar to regular files). Being seen as file descriptors, well-known system calls **write()** and **read()** might be used for data exchange between two sockets. Other similar primitives such as **recvmsg()** and **sendmsg()** are used in the application as well, in order to specify optional flags.

In order to create a new endpoint of a network communication path, the **socket()** primitive is used. To specify the use of the **TCP** transport protocol mentioned above, the domain parameter is set to AF_INET and the type parameter is set to SOCK_STREAM. For assigning an IP address and a Port to a socket, the **bind()** primitive is used.

2.4 Databases

There is a lot of information regarding the city map, logged users, weather details, sports news, and filling stations that should be stored in a database. The database engine used by the application is **SQLite3**.

SQLite3 is a C-library that implements a simple SQL database engine. SQLite does not have a separate server process. Instead, it reads and writes directly to ordinary disk files (usually .db files). There are three main SQLite primitives used by the application: **sqlite3_open()**, **sqlite3_exec()**, and **sqlite3_close()**. [7]

Thus, SQLite3 is smaller, faster, more reliable, and simpler than server-based database management systems. It is an excellent choice for this relatively small application.

3 Architecture

3.1 City Map Representation

Every client who enters the application is moving with a dynamic traveling speed, across different streets. In order to simulate the movement of a car, the speed is slowly increased after small periods of time. If the speed limit is exceeded, the speed of the driver abruptly decreases to satisfy the rules. Even though this is not the best simulation mechanism, it behaves well for testing purposes.

The application uses the map of **Chișinău** to simulate directions and streets. In order to represent the map in code, I took a small part of the map and transformed it into a weighted unoriented graph as follows:

- Every intersection is represented by a **node** in the graph. Each node has a unique id.
- Every street between two intersections is represented by an **edge** in the graph. Each edge also stores the street length (in meters) and the real name of the street.

A graphical representation of the graph described above can be found below. The orange markers are nodes and the blue lines between markers are edges.

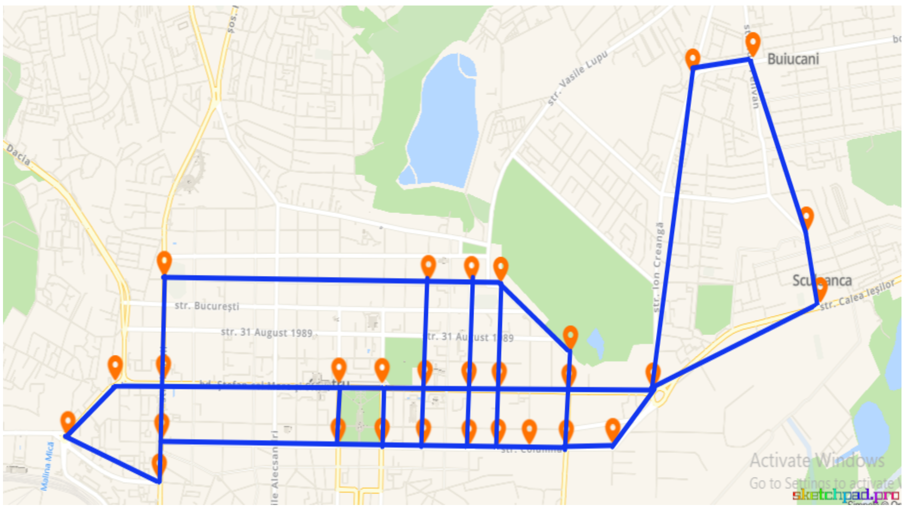
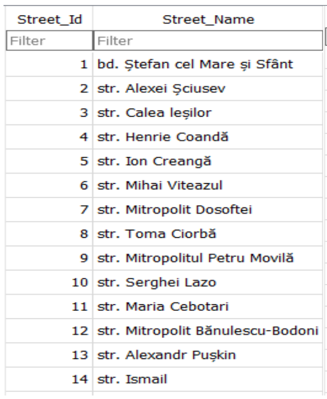


Fig. 1: Map used by the application

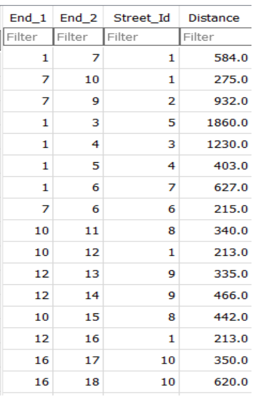
If a driver reaches an intersection, a **random** edge is chosen as the next street the driver will travel.

Both the client and server processes load the map from a relational **SQLite3** database that contains the following information



Street_Id	Street_Name
Filter	Filter
1	bd. Ștefan cel Mare și Sfânt
2	str. Alexei Șciusev
3	str. Calea Ieșilor
4	str. Henrie Coandă
5	str. Ion Creangă
6	str. Mihai Viteazul
7	str. Mitropolit Dosoftei
8	str. Toma Ciorbă
9	str. Mitropolitul Petru Movilă
10	str. Serghei Lazo
11	str. Maria Cebotari
12	str. Mitropolit Bănulescu-Bodoni
13	str. Alexandr Pușkin
14	str. Ismail

(a)



End_1	End_2	Street_Id	Distance
Filter	Filter	Filter	Filter
1	7	1	584.0
7	10	1	275.0
7	9	2	932.0
1	3	5	1860.0
1	4	3	1230.0
1	5	4	403.0
1	6	7	627.0
7	6	6	215.0
10	11	8	340.0
10	12	1	213.0
12	13	9	335.0
12	14	9	466.0
10	15	8	442.0
12	16	1	213.0
16	17	10	350.0
16	18	10	620.0

(b)

Fig. 2: (a) Streets Table (b) Roads Table

3.2 Client-Server Communication Diagrams

– **sign-up.** If the user wants to sign up, a request will be sent to the server, containing the name of the command. Several prompts will appear, asking the client to introduce his first name, last name, username, password, and whether or not he wants to subscribe to peco, sports, or weather channels. The data collected from the client will also be sent to the server. If all the data is valid (unique username and username of minimum 5 characters), the server will insert the information into an SQLite3 table called **Users** and will send a success message back to the client. Otherwise, a specific error message will be sent. An explicit diagram that describes the process might be seen below:

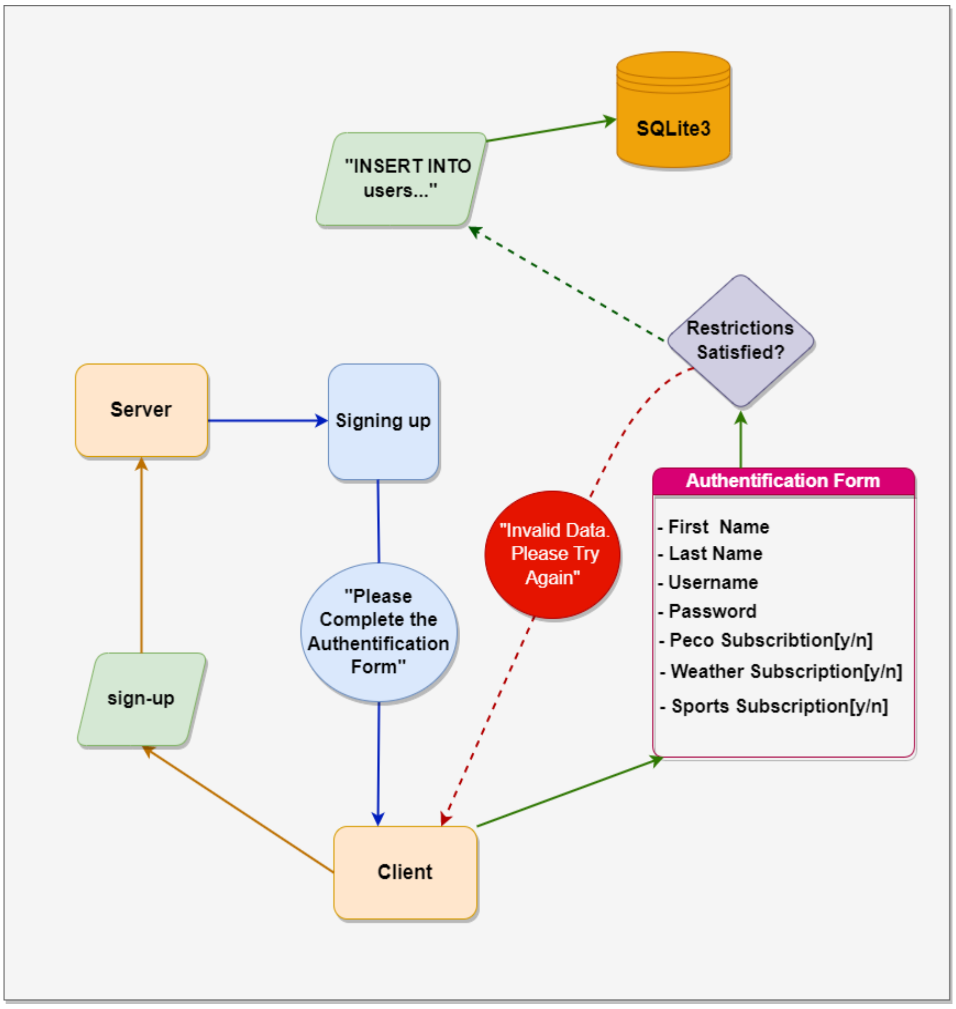


Fig. 3: The user types the "sign-up" command

– **login username.** If the user wants to log in he types this command, which will be immediately followed by a prompt asking for the password. After the user presses **enter**, the data will be transmitted to the server. The server will make a query to the SQLite3 database, asking for all the lines from the **Users** table. A line-by-line check will be performed. If the tuple (username, password) will be found in the database, the server will store the information about the user in a local array of structs that contains the client's socket descriptor, the username associated with that client, and boolean values representing subscriptions to the channels. In this way, the server will map each client descriptor to a real driver connected to the application. If the tuple (username, password) isn't found in the database or the user is already logged in, a specific error message will be sent to the client. All the procedures described above might be resumed by the following diagram:

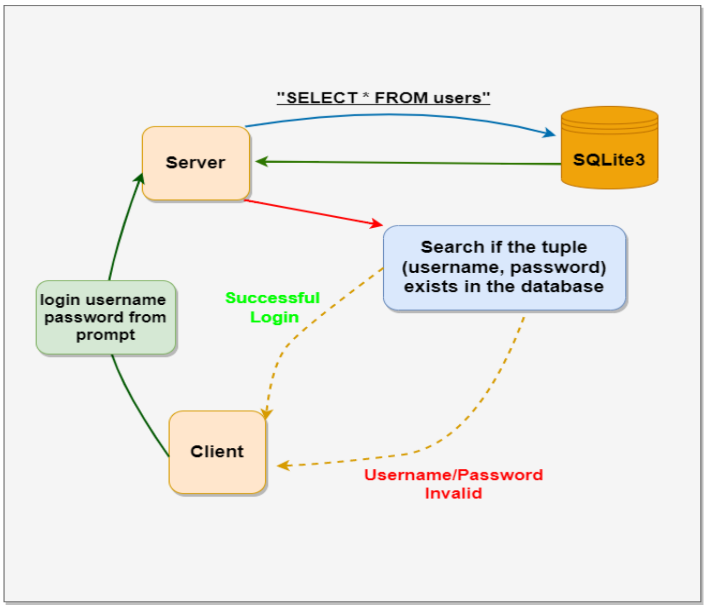

"SELECT * FROM users"
Server
SQLite3
Search if the tuple (username, password) exists in the database
login username password from prompt
Successful Login
Client
Username/Password Invalid

Fig. 4: ”login username” command description

– **report stname evcode**. If the user wants to report an event happening on a specific street, the report command is to be used. Once the server processed the command, it stores the **stname** and **evcode** parameters into a struct called event. This struct is then sent to **all** the clients (broadcasting process). When a client receives this notification through its own thread, it stores the event struct into a local array of structs called **notifications**. After sending the event struct to the clients, the server will also store the name of the street into the arrays **police**, **accidents**, **jams** or **repaired**, depending on the type of event. These arrays are helpful when dealing with other commands. A brief description of the process is shown below.

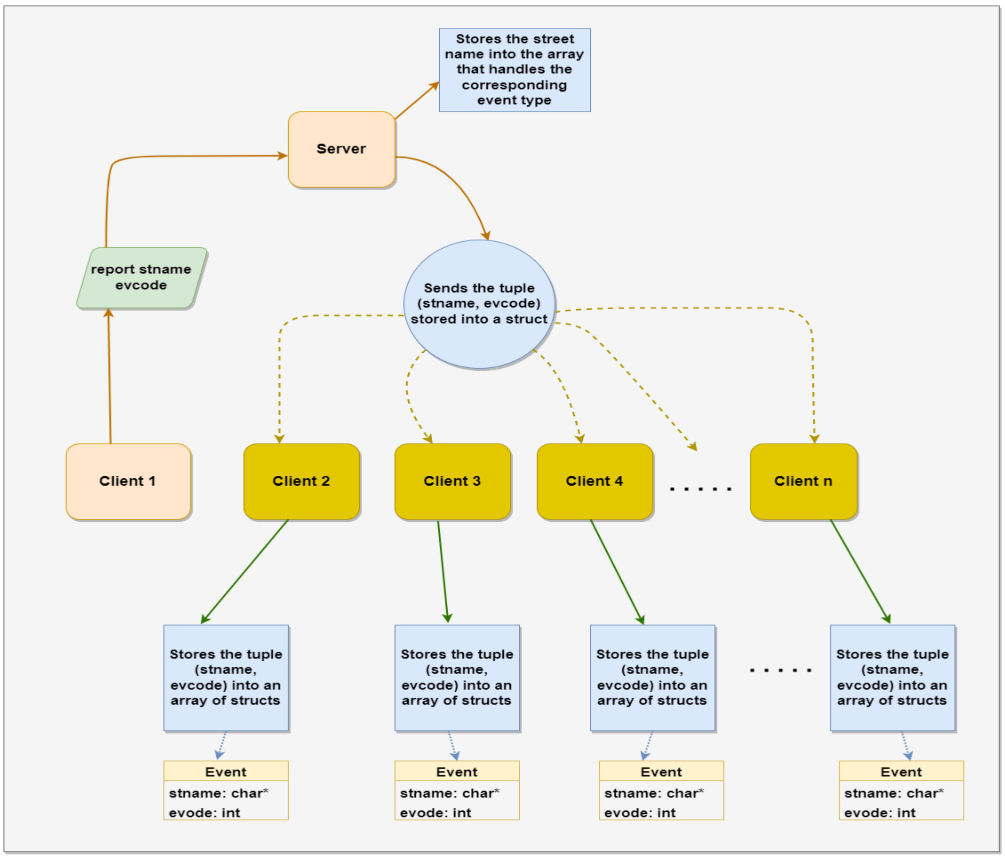

Stores the street name into the array that handles the corresponding event type
Server
report stname evcode
Sends the tuple (stname, evcode) stored into a struct
Client 1
Client 2
Client 3
Client 4
. . . . .
Client n
Stores the tuple (stname, evcode) into an array of structs
Stores the tuple (stname, evcode) into an array of structs
Stores the tuple (stname, evcode) into an array of structs
. . . . .
Stores the tuple (stname, evcode) into an array of structs
Event
stname: char*
evode: int
Event
stname: char*
evode: int
Event
stname: char*
evode: int
Event
stname: char*
evode: int

Fig. 5: ”report” command description

– **get-events**. The user constantly asks for the last notification received notifications sent by others, the client iterates through the **notifications** array (that contains event structs) and prints selects the specific notification.

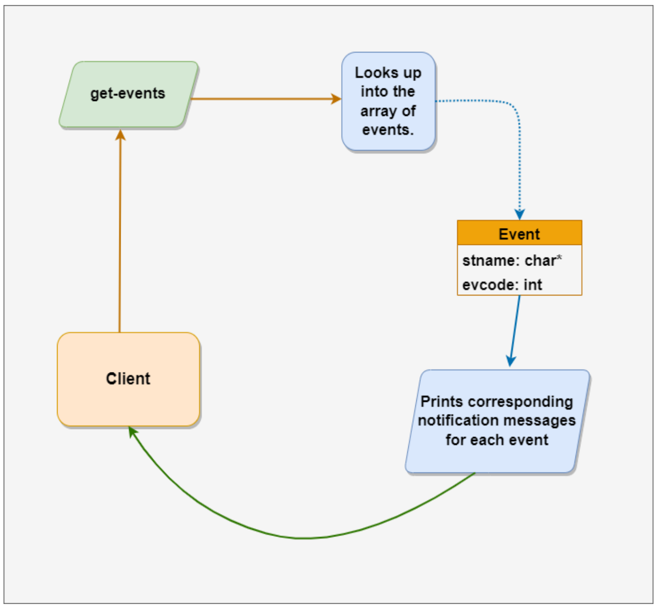

get-events
Looks up into the array of events.
Event
stname: char*
evcode: int
Prints corresponding notification messages for each event
Client

Fig. 6: "get-events" command description

– **get-<event>-info**. If the user wants to find out details about specific events, he might type one of the following commands: get-police-info, get-jam-info, get-accident-info, and get-repair-info. After processing the command, the server will determine the event code, will iterate through one of the **police**, **accidents**, **jams** or **repaired** arrays, and will collect data specific to the event required. This data is sent back to the client.

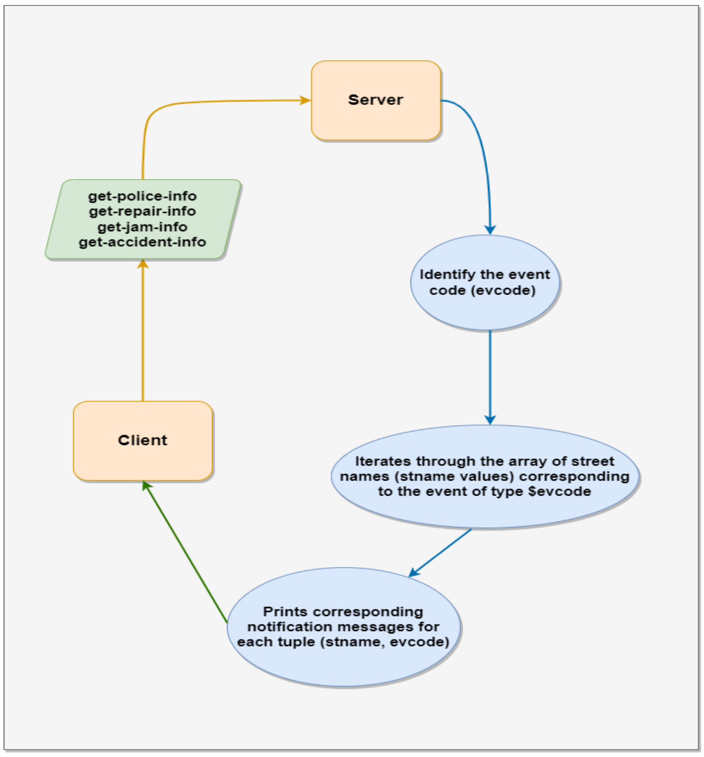

Server
get-police-info
get-repair-info
get-jam-info
get-accident-info
Identify the event code (evcode)
Client
Iterates through the array of street names (stname values) corresponding to the event of type $evcode
Prints corresponding notification messages for each tuple (stname, evcode)

Fig. 7: "get-<event>-info" command desription

– **get-<channel>-info**. The user might send one of the following commands to the server **get-sports-info option**, **get-peco-info** and **get-weather-info date** only if he is subscribed to the corresponding channel. The data about channels is stored in specific SQLite3 tables called **"Sports"**, **"Pecos"**, and **"Weather"**. The server identifies the table that corresponds to the <channel> parameter and sends data from it to the client.

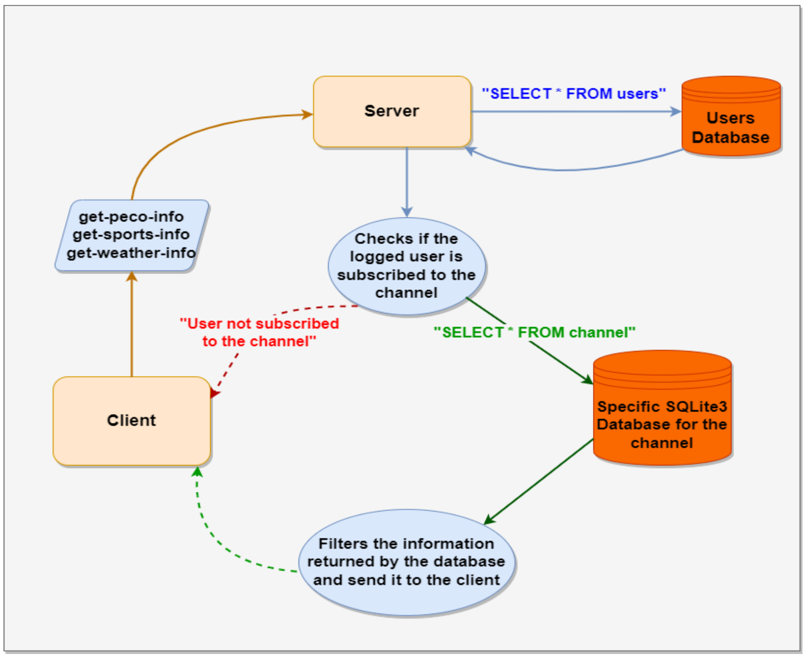

Server
"SELECT * FROM users"
Users Database
get-peco-info
get-sports-info
get-weather-info
Checks if the logged user is subscribed to the channel
"User not subscribed to the channel"
"SELECT * FROM channel"
Client
Specific SQLite3 Database for the channel
Filters the information returned by the database and send it to the client

Fig. 8: "get-<channel>-info" command description

– Apart from usual user commands, the client also sends to the server (through a separate thread) data about the traveling speed and the current location of the driver. This data is stored in a struct called GPS and a message is sent as follows: **auto-message GPS**. The GPS structure is sent in binary form. The server queries the **SQLite3** table Speed_Limits to check if the client has surpassed the speed limit. A warning message is sent to the client in an affirmative case.

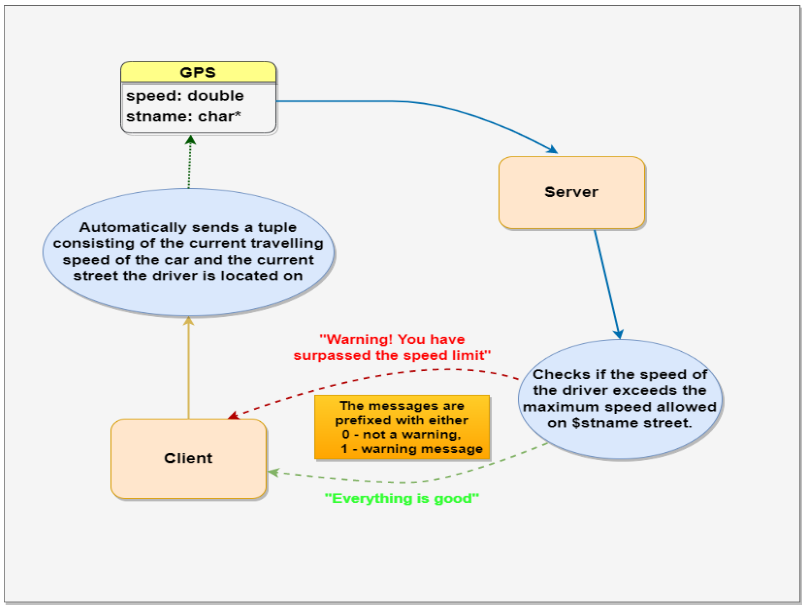

GPS
speed: double
stname: char*
Server
Automatically sends a tuple consisting of the current travelling speed of the car and the current street the driver is located on
"Warning! You have surpassed the speed limit"
Checks if the speed of the driver exceeds the maximum speed allowed on $stname street.
The messages are prefixed with either 0 - not a warning, 1 - warning message
Client
"Everything is good"

Fig. 9: Automatic messages

3.3 Client-Server Architecture

A general view of the application's architecture is shown below.

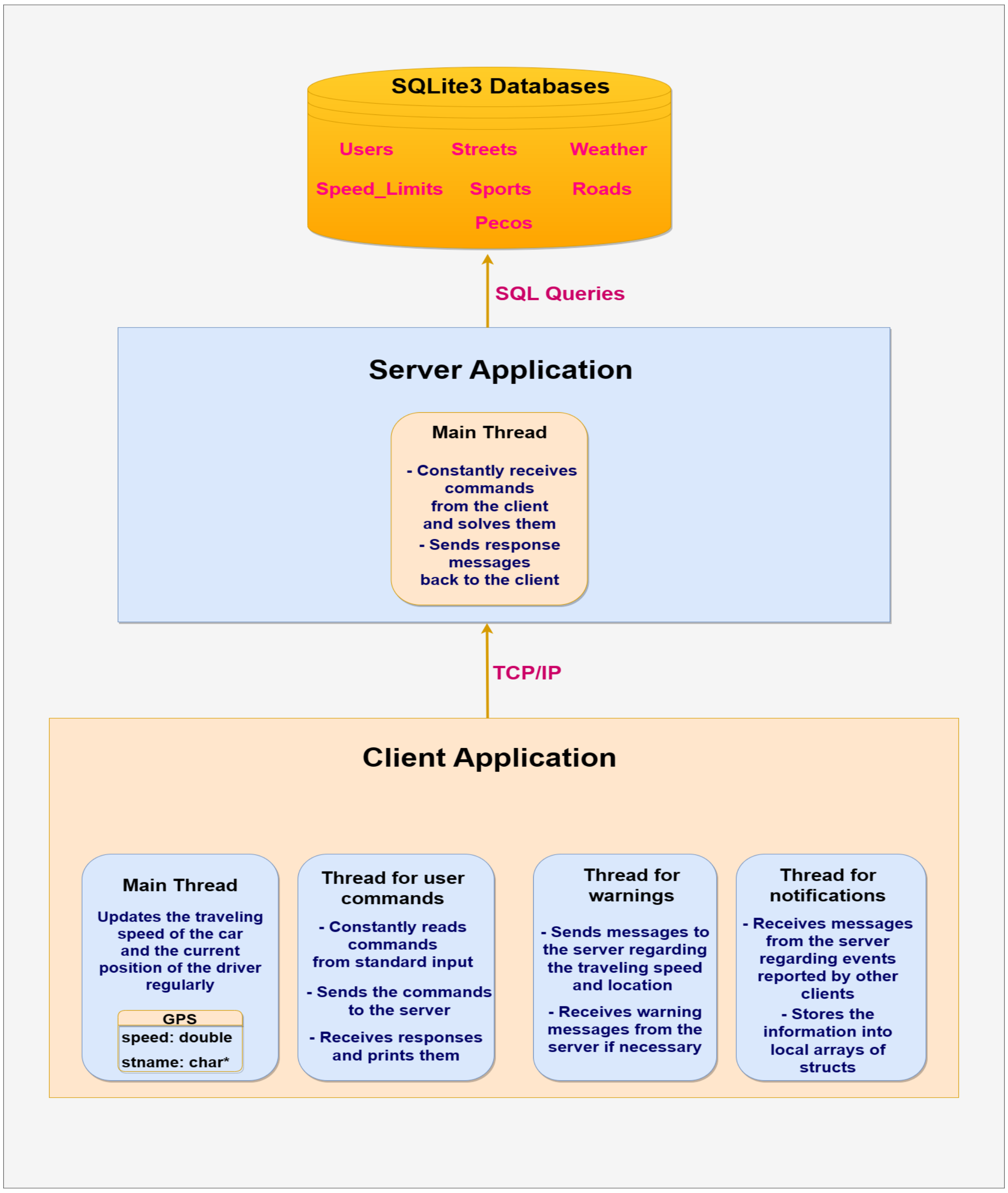

SQLite3 Databases
Users
Streets
Weather
Speed_Limits
Sports
Roads
Pecos
SQL Queries
Server Application
Main Thread
- Constantly receives commands from the client and solves them
- Sends response messages back to the client
TCP/IP
Client Application
Main Thread
Updates the traveling speed of the car and the current position of the driver regularly
GPS
speed: double
stname: char*
Thread for user commands
- Constantly reads commands from standard input
- Sends the commands to the server
- Receives responses and prints them
Thread for warnings
- Sends messages to the server regarding the traveling speed and location
- Receives warning messages from the server if necessary
Thread for notifications
- Receives messages from the server regarding events reported by other clients
- Stores the information into local arrays of structs

Fig. 10: General Architecture

3.4 Database Contents

There are several tables used in the application. As previously described, the **SQLite3** database stores data about signed-up users and information about sports news, weather forecasting, and filling stations. The city map is also stored in the database.
All the tables and attributes from the database are shown below.

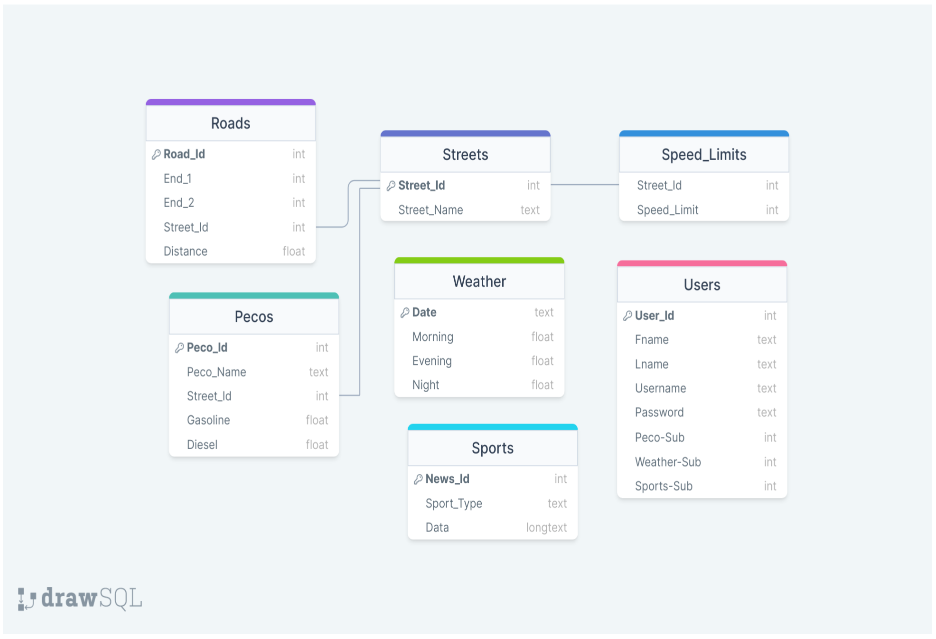


Fig. 11: Attributes and Tables from the database

4 Implementation Details

– Important important **structures** used in code

```
struct node {
    int cross_id;
    int distance;
    char street_name[MAX_STR_NAME];
    struct node *next;
};

struct city_map {
    struct node *adj[ROADS];
};

struct GPS {
    double speed;
    char street_name[MAX_STR_NAME];
}

struct Event {
    int evcode,
    char street_name[MAX_STR_NAME];
}

struct client_data {
    char username[MAX_CRED];
    short logged, peco_sub, sport_sub, weather_sub;
};
struct user_creds {
    char username[MAX_CRED];
    char password[MAX_CRED];
```

```c
};

struct auth_creds {
      char fname[MAX_NAME], lname[MAX_NAME];
      char username[MAX_CRED];
      char password[MAX_CRED];
      short peco_sub, sport_sub, weather_sub;
};
```

– A function that extracts the city map from the database and uses adjacency lists to store the graph

```c
void get_map_info() {
      map = malloc(sizeof(struct city_map));
      rc = sqlite3_open("Proj_Database.db", &db);
      DB_CHECK(!rc, sqlite3_errmsg(db));
      rc = sqlite3_exec(db, map_qry_statement, get_map,
          (void *)map, &errMsg);
      DB_CHECK(!rc, errMsg);
      // the map was successfully extracted
      sqlite3_close(db);
}
```

– The above-mentioned function uses the **sqlite3_exec** command that processes each line returned by the SQL query one by one. The callback function **get_map** is used to update the edges of the graph.

```c
static int get_map(void *data, int argc, char **argv, char
    **col_name) {
      struct city_map *cmap = (struct city_map*) data;
      int u = atoi(argv[1]);
      int v = atoi(argv[2]);
      add_edge(&cmap->adj[u], v, atoi(argv[4]), argv[5]);
      add_edge(&cmap->adj[v], u, atoi(argv[4]), argv[5]);
      return 0;
}

void add_edge(struct node **group, int vec, int dist, char
    *str_name) {
      struct node *obj = malloc(sizeof(struct node));
      obj->cross_id = vec;
      obj->distance = dist;
      strncpy(obj->street_name, str_name, MAX_STR_NAME);
      obj->next = *group;
      *group = obj;
}
```

– The client-thread that constantly updates the speed and the location of the user is described below:

```c
for(int i = 0; i < ROADS; ++i) {
      if(map->adj[i]) {
            j = map->adj[i];
            while(j != NULL) {
                  neighbours[i]++;
                  j = j->next;
            }
      }
}

```

```c
int init_cross = rand() % ROADS;
while(map->adj[init_cross] == NULL) {
    init_cross = rand() % ROADS;
}

int pos = rand() % neighbours[init_cross];
j = map->adj[init_cross];

while(pos && j->next != NULL) {
    j = j->next;
    pos--;
}

gps->prd_cross = init_cross;
gps->nxt_cross = j->cross_id;
gps->distance = j->distance;
strncpy(gps->street_name, j->street_name, MAX_STR_NAME);
gps->speed = 20 + rand() % 31;

// printf("%d %d %f %f\n", gps->prd_cross, gps->nxt_cross, gps->distance, gps->speed);

while(1) {
    // updates after every second
    sleep(1);
    // speed * 1000 / 3600 * 3 seconds (1 second in reality = 3 seconds in the simulation)
    double travelled_distance = gps->speed * (1000.0 / 3600.0) * 3;
    gps->distance -= travelled_distance;

    if(gps->distance < 0) {
        // randomly choose the next intersection to go
        int nxt_pos = rand() % neighbours[gps->nxt_cross];
        j = map->adj[gps->nxt_cross];

        while (nxt_pos && j->next != NULL) {
            j = j->next;
            nxt_pos--;
        }
        // update everything in the gps structure
        int aux = gps->nxt_cross;
        gps->nxt_cross = j->cross_id;
        gps->prd_cross = aux;
        gps->distance = j->distance;
        strncpy(gps->street_name, j->street_name, MAX_STR_NAME);
    }

    int decision = rand() % 3;
    // (0 -> lower the speed with 1 km/h) (1 -> maintain the current speed) (2 -> increase the speed with 1 km/h)
    if(decision == 0 && gps->speed > 10) {
        gps->speed--;
    }
    else if(decision == 1) {
        gps->speed++;
    }
}
```

– Every message between the client and the server is prefixed by its length. Thus, when a message is read, the first 4 bytes are always stored in an **int** value representing the length!
– There are situations when two or more threads want to access a shared resource at the same time. In this situations, **pthread mutexes** are used. The primitives **pthread_mutex_lock** and **pthread_mutex_unlock** are used before and after the shared resource is being accessed. In this way, an order is preserved and there is only one thread managing a given resource at a time.
– **Ansi Escape Codes** [11] are also used in the application for coloring the text, moving the cursor and erasing lines of text in the terminal.

5 Conclusion

There are still some details that might be improved in the application:

– The **UDP** transport protocol is faster than the **TCP** protocol. However, the UDP protocol doesn't resend the lost packets, so there is a chance that some data is lost. The most efficient implementation should use **both** TCP and UDP protocols. UDP might be used for commands like "get-peco-info, get-weather-info" and "get-sports-info".
– Another improvement would be to use IPv6 instead of IPv4 because it has way more addresses and is widely used nowadays.
– This application is just a model for a real and practical traffic monitoring system. Instead of manually changing the speed and location values, a GPS tracking system should be used.
– More stations and intersections should be added to the map, in order to make the application more practical.
– A graphic library (eg. SFML, QT, etc.) might be used to improve the appereance of the application.

References

1. Alboaie L. Concurrent Server Implementation. https://profs.info.uaic.ro/~computernetworks/files/NetEx/S12/ServerConcThread/servTcpConcTh2.c
2. Alboaie L. Concurrent Client Implementation. https://profs.info.uaic.ro/~computernetworks/files/NetEx/S12/ServerConcThread/cliTcpNr.c
3. Alboaie L. and Panu A. Computer Networks Course. https://profs.info.uaic.ro/~computernetworks/cursullaboratorul.php
4. Geeks for Geeks. Details On Multithreading. https://www.geeksforgeeks.org/thread-in-operating-system/
5. Wikipedia. TCP/IP details. https://en.wikipedia.org/wiki/Internet_protocol_suite
6. Wikipedia. TCP Transport Protocol details. https://en.wikipedia.org/wiki/Transport_layer
7. SQLite. General overview of the SQLite3 library. https://www.sqlite.org/about.html
8. Soft For Diagrams. https://www.draw.io/index.html
9. Map Of Chișinău. https://map.md
10. Soft For Table Visualization. https://drawsql.app/
11. About ANSI Escape Codes. https://en.wikipedia.org/wiki/ANSI_escape_code
12. LNCS Homepage. http://www.springer.com/lncs