Raport tehnic - Mersul Trenurilor

Sandu Theodor

12.12.2024

1 Introducere

Proiectul propus presupune crearea unui server multithreading dedicat gestionării conexiunilor client-server pentru o aplicație de monitorizare a trenurilor.

Scopul proiectului constă în furnizarea unei experiențe eficiente și sincronizate, unde fiecare client poate trimite și primi date despre trenuri în timp real.

Obiectivele principale ale proiectului includ gestionarea conexiunilor multiple ale clienților, primirea și trimiterea mesajelor către clienți, actualizarea datelor despre trenuri și gestionarea deconectărilor clienților, cât și a închiderii serverului.

2 Tehnologii Aplicate

În cadrul acestui proiect, am ales să utilizez limbajul de **programare C** pentru implementare, combinându-l cu tehnologii specifice cât si **Object-Oriented Programming** pentru a dezvolta o aplicație bazată pe comunicarea prin TCP și gestionarea concurentă a clienților. Limbajul C oferă control detaliat asupra resurselor sistemului și este potrivit pentru dezvoltarea aplicațiilor de rețea eficiente.

Prin integrarea comunicării prin **TCP**, asigurăm o transmitere fiabilă și ordonată a datelor între server și clienți. **Gestionarea concurentă** a clienților este implementată pentru a permite unui număr nelimitat de participanți să interacționeze simultan cu serverul. Conexiunile TCP sunt caracterizate de capacitatea lor full-duplex, ceea ce înseamnă că permit transmiterea și primirea datelor simultan și independent. Această funcționalitate este esențială pentru a asigura o interacțiune fluentă între server și clienți în cadrul aplicației.

Tehnologiile specifice includ utilizarea firelor de execuție (thread-urilor) pentru a gestiona eficient multiplele conexiuni în mod concurent. Această abordare permite serverului să comunice cu mai mulți clienți simultan, fără a afecta performanța generală a aplicației.

3 Structura Aplicației

Aplicația este divizată în două componente principale: **server și clienți**, conectate prin intermediul **socket-urilor**. Programarea cu socket-uri este o modalitate de a conecta două noduri pe o rețea pentru a comunica între ele. Un socket (nod) ascultă pe un port specific la o adresă IP, în timp ce celălalt socket inițiază conexiunea pentru a se conecta la celălalt. Serverul formează socket-ul ascultător, în timp ce clientul se conectează la server.

Serverul are responsabilitatea de a efectua operațiuni de citire și scriere către clienți. Aici, sunt gestionate operațiuni precum primirea și trimiterea mesajelor, actualizarea datelor despre trenuri și gestionarea deconectărilor clienților.

Clienții, în schimb, au un rol simplu și interacționează cu datele despre trenuri primite de la server. Aceștia primesc datele, le procesează și trimit răspunsurile înapoi către server pentru a fi procesate. Interfața simplă a clienților le permite să se concentreze pe monitorizarea trenurilor fără a gestiona aspecte complexe ale logicilor de rețea sau stocării datelor. Prin această structură, serverul acționează ca o entitate centrală, coordonând comunicarea și actualizarea datelor, iar clienții interacționează activ cu datele primite. Comunicarea eficientă între aceste două componente permite o experiență fluidă și sincronizată pentru toți participanții.

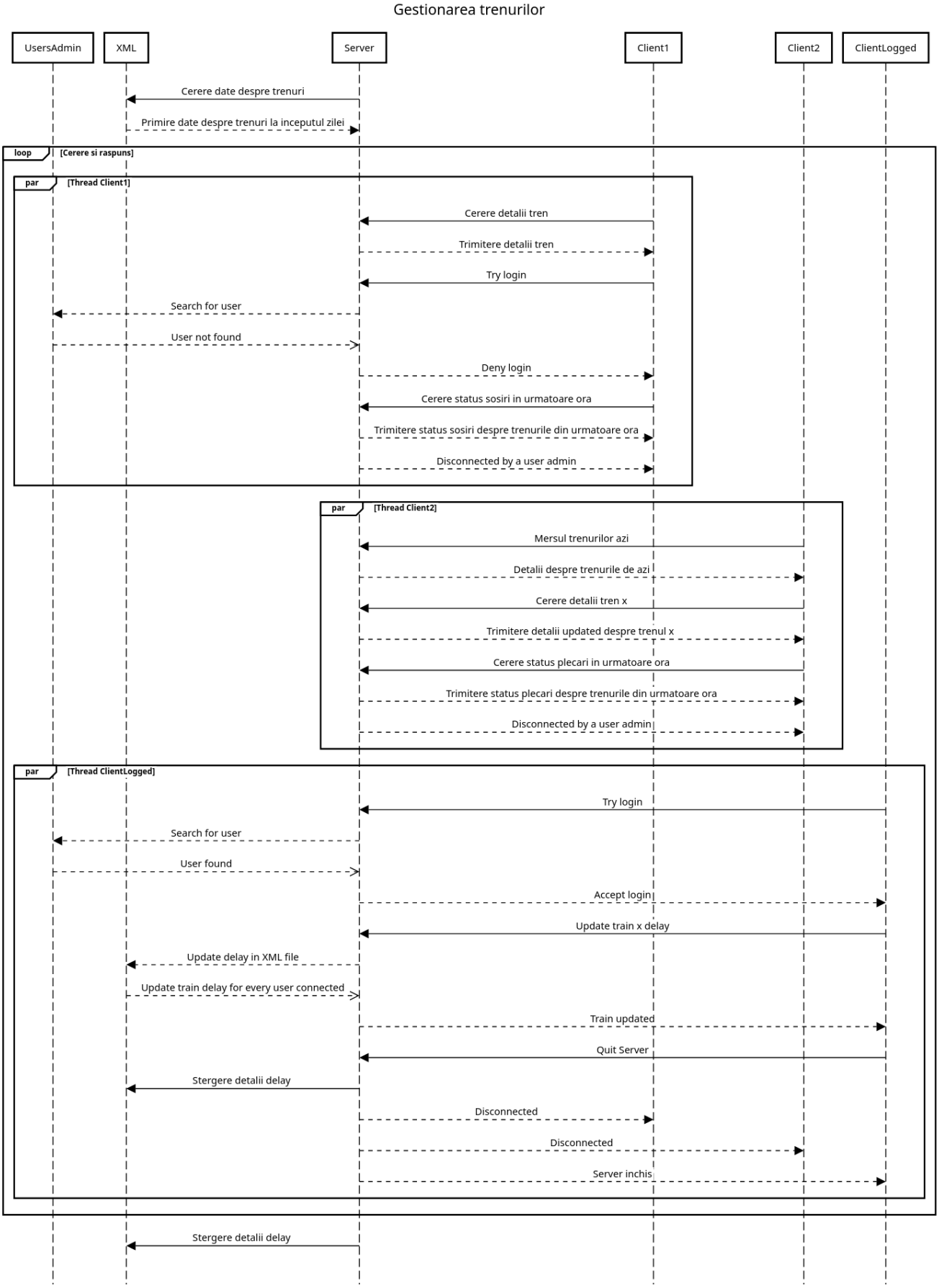
Implementarea serverului multithreading se realizează folosind biblioteca POSIX Threads (pthread). Alegerea thread-urilor în locul altor metode, precum select sau pipe-uri, aduce multiple avantaje. Fiecare conexiune client-server este gestionată de un thread separat, permițând paralelism real și utilizarea eficientă a procesoarelor multicore. Acest model simplifică logica aplicației, deoarece fiecare conexiune urmează un flux liniar, spre deosebire de abordarea bazată pe evenimente, necesară în cazul metodei select.

Thread-urile oferă o scalabilitate mai bună, gestionând eficient un număr mare de conexiuni simultane, spre deosebire de select, care devine ineficient pe măsură ce numărul de conexiuni crește. În plus, ele permit o comunicare directă între conexiuni prin partajarea aceluiași spațiu de memorie, evitând copia datelor între buffer-uri, ca în cazul pipe-urilor.

Modul de gestionare a cererilor
Clienții ne logați pot cere informații despre ***Mersul trenurilor azi***,***Detalii despre un tren***, ***Status sosiri în următoarea oră***, ***Status plecări în următoarea oră***, ***Traseu de la A la B***

Modul de gestionare a cererilor de către userii autentificați
Clienții logați pot ***Updata delay-ul trenurilor pentru toți userii***, ***Inchide serverul (si automat deconectând toți userii)***

Diagrama de mai jos ilustrează fluxurile de date dintre componentele de bază:



4 Aspecte de Implementare

- Un exemplu de cod inovativ este metoda `update_train_data` din clasa `Trains`, care actualizează întârzierile trenurilor și modifică fișierul XML corespunzător:

```
char *filename = "map.xml";
xmlDoc *doc = xmlReadFile(filename, NULL, 0);
if (!doc) {
    fprintf(stderr, "Failed to load XML file: %s\n", filename);
    return false;
}

xmlNode *root = xmlDocGetRootElement(doc);
if (!root) {
    fprintf(stderr, "Empty XML document: %s\n", filename);
    xmlFreeDoc(doc);
    return false;
}

xmlNode *cur_node = NULL;
for (cur_node = root->children; cur_node; cur_node = cur_node->next) {
    if (cur_node->type == XML_ELEMENT_NODE && strcmp((char *) cur_node->name, "XmlMts") == 0) {
        xmlNode *mts_node = cur_node->children;
        for (; mts_node; mts_node = mts_node->next) {
            if (mts_node->type == XML_ELEMENT_NODE && strcmp((char *) mts_node->name, "Mt") == 0) {
                xmlNode *trenuri_node = mts_node->children;
                for (; trenuri_node; trenuri_node = trenuri_node->next) {
                    if (trenuri_node->type == XML_ELEMENT_NODE && strcmp(
                            (char *) trenuri_node->name, "Trenuri") == 0) {
                        xmlNode *tren_node = trenuri_node->children;
                        for (; tren_node; tren_node = tren_node->next) {
                            if (tren_node->type == XML_ELEMENT_NODE && strcmp(
                                    (char *) tren_node->name, "Tren") == 0) {
                                xmlChar *numar = xmlGetProp(tren_node, (const xmlChar *) "Numar");
                                if (numar && atoi((char *) numar) == train_id) {
                                    xmlNode *restrictii_node = NULL;
                                    for (xmlNode *child = tren_node->children; child; child = child->next) {
                                        if (child->type == XML_ELEMENT_NODE && strcmp(
                                                (char *) child->name, "Intarziere") == 0) {
                                            restrictii_node = child;
                                            break;
                                        }
                                    }

                                    if (restrictii_node) {
                                        xmlSetProp(restrictii_node, BAD_CAST "delay",
                                                   BAD_CAST delay.c_str());
                                    } else {
                                        restrictii_node = xmlNewChild(
                                            tren_node, NULL, BAD_CAST "Intarziere", NULL);
                                        xmlNewProp(restrictii_node, BAD_CAST "delay",
                                                   BAD_CAST delay.c_str());
                                    }

                                    xmlFree(numar);
                                    break;
                                }
                                if (numar)
                                    xmlFree(numar);
                            }
                        }
                    }
                }
            }
        }
    }
```

- Un alt exemplu este clasa `Graph.h` care realizează un graf orientat între toate stațiile, creând o rețea, pentru a putea aplica algoritmi eficienți pentru drumul cel mai scurt:

```cpp
TrainGraph::TrainGraph(const std::vector<Train*>& trains) {
    int index = 0;

    for (const auto& train : trains) {
        std::string start_station = train->get_start_station();
        std::string end_station = train->get_end_station();

        // putem statia de plecare in map daca nu exista
        if (station_index.find(start_station) == station_index.end()) {
            station_index[start_station] = index++;
            lista_adiacenta.push_back({});
        }

        // putem statia finala in map daca nu exista
        if (station_index.find(end_station) == station_index.end()) {
            station_index[end_station] = index++;
            lista_adiacenta.push_back({});
        }

        // adaugam muchia in list de adiacenta
        add_muchie(start_station, end_station);
    }
}

void TrainGraph::add_muchie(const std::string& start_station, const std::string& end_station) {
    int start_index = station_index[start_station];
    std::set<std::string> muchii(lista_adiacenta[start_index].begin(), lista_adiacenta[start_index].end());
    muchii.insert(end_station);
    lista_adiacenta[start_index] = std::vector<std::string>(muchii.begin(), muchii.end());
}
```

- Realizarea drumului cel mai scurt de la o stație la alta se realizează prin aplicarea algoritmului lui **Dijkstra** astfel:

```cpp
std::string TrainGraph::find_shortest_path(const std::string& start_station, const std::string& end_station) const {
    if (station_index.find(start_station) == station_index.end() || station_index.find(end_station) == station_index.end()) {
        return "Start or end station wrong";
    }

    int n = station_index.size();
    std::vector<int> distances(n, 2e9);
    std::vector<int> previous(n, -1);
    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, std::greater<>> pq;

    int start_index = station_index.at(start_station);
    distances[start_index] = 0;
    pq.push({0, start_index});

    while (!pq.empty()) {
        int current_distance = pq.top().first;
        int current_index = pq.top().second;
        pq.pop();

        if (current_distance > distances[current_index]) {
            continue;
        }

        for (const auto& nod : lista_adiacenta[current_index]) {
            int nod_index = station_index.at(nod);
            int new_distance = current_distance + 1; // inca n-am implementat cu timpii trenurilor

            if (new_distance < distances[nod_index]) {
                distances[nod_index] = new_distance;
                previous[nod_index] = current_index;
                pq.push({new_distance, nod_index});
            }
        }
    }
```

- Datele despre sosirile din urmatoarea ora dintr-un oras dat se afla astfel:

```cpp
std::string Trains::next_hour_trains_city(const char station[],const bool &caz) {
    std::string station_string = this->get_city(station);
    if (station_string ==
        return "Orasul nu
    const auto now = std::chrono::system_clock::now();
    const std::time_t currentTime = std::chrono::system_clock::to_time_t(now);
    const std::tm localTime = *std::localtime(&currentTime);
    const int secondsSinceMidnight = localTime.tm_hour * 3600 + localTime.tm_min * 60 + localTime.tm_sec;

    std::string response;
    for (const auto &train: trains) {
        std::string arrival_time;
        std::string text;
        if (caz == false ) {  ///false pentru arrivals
            arrival_time = train->arrival_time();
            text = " ajunge de la ";
        } else {
            arrival_time = train->departure_time();
            text = " pleaca la ";
        }
        std::string hours = arrival_time.substr(0, 2);
        std::string minutes = arrival_time.substr(3, 2);
        std::string seconds = arrival_time.substr(6, 2);
        int arrival_seconds = stoi(hours) * 3600 + stoi(minutes) * 60 + stoi(seconds);
        if (secondsSinceMidnight <= arrival_seconds && arrival_seconds <= secondsSinceMidnight + 3600 && (
                caz ? train->get_start_station() : train->get_end_station()) == station_string)
            response.append(
                "Trenul: " + std::to_string(train->getId()) + text + (caz ? train->get_end_station() : train->get_start_station()) + " la ora " +
                arrival_time + '\n');
    }
    std::string seconds = std::to_string(secondsSinceMidnight);

    return response.empty() ? "Nu exista trenuri" : response;
}
```
namespace std::__1::chrono
chrono

- Fiecare comandă primită de la client este prelucrată de obiectul **CommandHandler** iar apoi apelează metoda potrivită din clasa Trains:

```cpp
    command[strlen(command) - 1] = '\0';
    char *command1 = strtok(command, ":");
    std::cout<<"Commanda este:"<<command1<<std::endl;

    if (strcmp(command1, "Mersul trenurilor azi") == 0) {
        return trains.get_trains_data();
    }
    if (strcmp(command1, "Detalii tren") == 0) {
        // std::cout<<"Trimit inapoi:" << trains.get_train_data(10242);
        const char *train = strtok(nullptr, ":");
        int train_id = atoi(train);
        return trains.get_train_data(train_id);
    }
    if (strcmp(command1, "login") == 0) {
        char *username = strtok(nullptr, ":");
        if (cauta_username_xml(username) == true) {
            this->logged = true;
            return "Logged in";
        }
        else {
            this->logged = false;
            return "Username incorect";
        }
    }
    if (strcmp(command1, "logout") == 0) {
        this->logged = false;
        return "Logged out";
    }
    if (strcmp(command1, "quitserver") == 0) {
        if (this->logged == false)
            return "Not logged in";
        else {
            this->logged = false;
            trains.delete_intarziere_arrival(std::nullopt, true);
            trains.delete_intarziere_departure(std::nullopt, true);
            return "quit";
        }
    }
    if (strcmp(command1, "Traseu") == 0) {
        char* start_station = strtok(nullptr, " ");
        char* end_station = strtok(nullptr, "\n");
        if (start_station == nullptr || end_station == nullptr)
            return "Orase lipsa";
        //trains.print_graphs("București", "Cluj");
        return trains.print_graphs(start_station, end_station);
        return "printed";
    }
     if (strcmp(command1, "quit") == 0) {
        this->logged = false;
        return "quit";
    }
    if (strcmp(command1, "Next hour departures") == 0) {
        return trains.next_hour_trains(true);
    }
    if (strcmp(command1, "Next hour departures from") == 0) {
        char* station = strtok(nullptr, " ");
        if (station == nullptr)
            return "Oras lipsa";
        return trains.next_hour_trains_city(station, true);
    }
    if (strcmp(command1, "Next hour arrivals") == 0) {
        return trains.next_hour_trains(false); ///false pentru arrivals
    }
    if (strcmp(command1, "Next hour arrivals from") == 0) {
        char* station = strtok(nullptr, " ");
        if (station == nullptr)
            return "Oras lipsa";
        return trains.next_hour_trains_city(station, false);
    }
    if (strcmp(command1, "Update arrival") == 0) {
        if (this->logged == true) {
```

- Orice **tren** are următoarele atribute:

```
You, yesterday | 2 authors (theo and one other)
class Train{
private:
    int id;                 // Numarul trenului
    std::string departure;     // Ora plecare
    std::string arrival;       // Ora sosire
    std::string category;      // Categorie tren
    std::string start_station; // Statia de plecare
    std::string end_station;   // Stația finală
    std::string delay_arrival; // Intarziere sosire
    std::string delay_departure; // Intarziere plecare


public:
    Train(int, const std::string &, const std::string &, const std::string &, const std::string &, const std::string &);
    [[nodiscard]] int getId() const;
    [[nodiscard]] std::string get_train_data() const;
    void update_delay(const std::string& delay_received);
    void update_delay_departure(const std::string& delay_received);
    [[nodiscard]] std::string arrival_time() const;
    [[nodiscard]] std::string departure_time() const;
    [[nodiscard]] std::string get_end_station() const;
    [[nodiscard]] std::string get_start_station() const;
};
```

- **Protocolul la Nivelul Aplicației:** Protocolul nostru la nivelul aplicației implică trimiterea și primirea de mesaje XML prin TCP/IP pentru a actualiza și a obține informații despre trenuri. Fiecare mesaj conține informații structurate despre trenuri, inclusiv întârzieri și stații.

- **Scenarii Reale de Utilizare:**

  - **Monitorizarea Trenurilor:** Utilizatorii pot vizualiza informații actualizate despre trenuri, inclusiv orele de plecare și sosire, și eventualele întârzieri.
  - **Actualizarea Întârzierilor:** Administratorii pot actualiza întârzierile trenurilor în timp real, iar aceste modificări sunt reflectate imediat în aplicație.

5 Concluzii

Soluția propusă poate fi îmbunătățită prin:

- Implementarea unui sistem de notificări pentru utilizatori în cazul întârzierilor majore.

- Optimizarea performanței aplicației pentru a gestiona un număr mai mare de trenuri și utilizatori.

- Extinderea funcționalităților pentru a include mai multe detalii despre trenuri și rute.

6 Bibleografie

https://en.cppreference.com/ (pentru type cast de chrono to int/string)
https://edu.info.uaic.ro/computer-networks/cursullaboratorul.php
https://gnome.pages.gitlab.gnome.org/libxml2/devhelp/index.html