Mersul trenurilor [Raport Tehnic]

Mandrea Radu-Alexandru

December 12, 2023

Introducere

Proiectul propune implementarea unui sistem de actualizare si afisare a informatiilor despre mersul trenurilor in timp real, inclusiv statusul plecarilor, sosirilor dintr-o anumita statie, dar si prezenta intarzierilor si estimarilor intervalului de sosire. Serverul foloseste drept baza de date fisiere XML si le actualizeaza in functie de cererile primite de la clienti. Obiectivul proiectului este de a construi un produs ce ofera atat informatii precise cat si actualizate in timp real catre toti clientii inregistrati in aplicatie.

Tehnologii Aplicate

Pentru a asigura conexiunea intre clienti si server, in dezvoltarea proiectului vom folosi protocolul de comunicare TCP, datorita avantajelor acestuia. Utilizam **TCP**, deoarece acesta asigura fiabilitatea, corectitudinea si ordinea transferului de date in ceea ce priveste schimbul de mesaje dintre utilizatori si server. Mai mult decat atat, pentru a ne asigura de faptul ca fiecare client primeste date actualizate in timp real, vom implementa in program o **coada de comenzi**(Command Queue), astfel incat, requesturile ce au drept scop modificarea fisierului XML, sa se execute inaintea requesturilor ce doresc doar o iterare a **bazei de date**. Asadar, implementarea acestei tehnici asigura o transmitere ordonata, dar si sincronizata a informatiilor catre clienti
Pentru a implementa concurenta, proiectul Mersul Trenurilor se bazeaza pe utilizarea a unor multiple **fire de executie**. Unul principal ce asigura parsarea continua a cozii de comenzi, dar si cate un fir de executie pentru fiecare client nou, ce se conecteaza la server. De asemenea, fiecare comanda ce modifica delay-ul privind sosirea sau plecarea ternurilor se executa iterativ pentru a nu modifica baza de data cu informatii eronate.
Implementarea foloseste de asemenea **design pattern-ul** Command pentru a gestiona comenzile primite de la clienti, idee ce adauga atat lizibilitate codului cat si formeaza un mecanism mai usor pentru a executa comenzile primite.

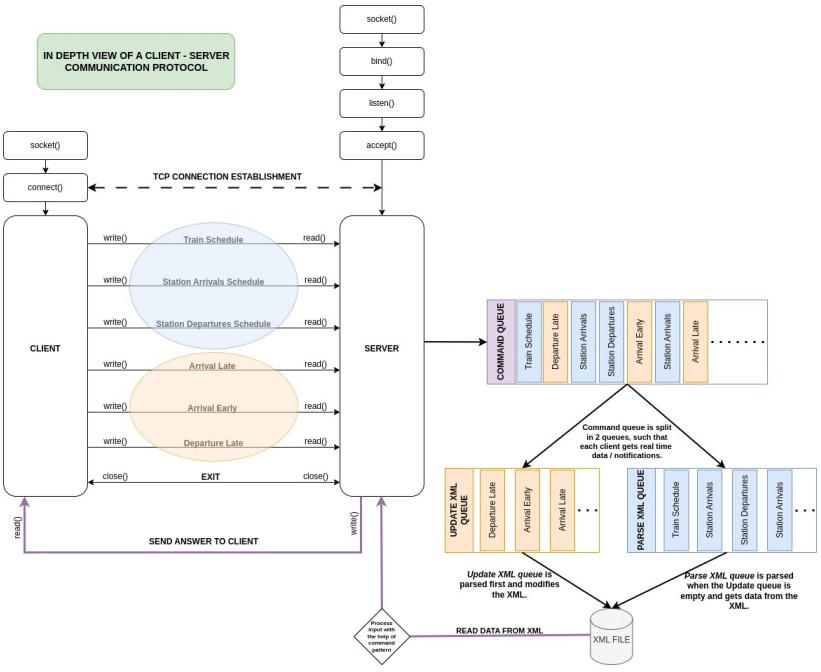
Structura Aplicatiei

Partea de server este structurată în mod modular, fiind compusa din mai multe componente. Primul modul are rolul de a crea o conexiune intre client si server. Deoarece folosim **TCP**, schema prezinta procesul de **three way handshaking** ca mai apoi clientul sa poata sa inceapa schimbul de mesaje cu serverul prin intermediul primitivelor **read()** si **write()**. Daca clientul este acceptat, acesta poate trimite comenzi catre server(ce necesita o sintaxa obligatorie).

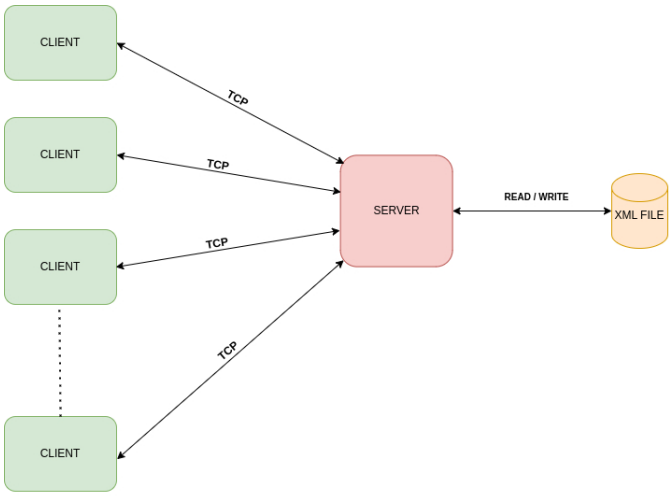
A doua componenta a aplicatiei este formata din cozile de comenzi. Fiecare comanda primita de server este trimisa catre o **coada de comenzi** principala. Aceasta mai departe imparte comenzile in alte doua cozi.

A treia componenta a proiectului contine un **command design pattern** care colaboreaza continuu cu o "baza de date" sub forma unui **fisier XML**. Asadar, CDP proceseaza comanda primita, genereaza raspunsul in server care apoi prin intermediul primitivei **write()** este trimis la client(traseu mov).

Pentru o vizualizare detaliata in ceea ce priveste conexiunea dintre un singur client si server si ce valorifica modulele prezentare mai sus, putem urmari schema de mai jos:



O imagine de ansamblu, asupra intregului proiect ce promoveaza ideea de concurenta in utilizarea protocolului TCP dar si capacitatea serverului de a deservi un numar nelimitat de clienti este prezentata mai jos:



In continuare vom observa aspecte importante ce tin de implementarea structurilor de date folosite, a mecanismelor de iterare prin acestea dar si a functiilor din standardul POSIX in configurarea conexiunii TCP client - server.

Aspecte de implementare

In primul rand, proiectul Mersul Trenurilor este scris integral in limbajul C++ atat partea de server cat si partea de client.

Pentru a comunica intre client si server vom folosi primitivele de sistem **read()** si **write()** ce au un caracter blocant si asigura buna mentinere a unui control ce priveste fluxul erorilor sau al congestiilor de sistem.

Pentru fiecare client acceptat de server, vom folosi fire de executie din structura **pthread**. De asemenea, firul general (**main**) verifica daca coada de comenzi are comenzi ce asteapta sa fie solutionate. Alte thread-uri sunt folosite pentru a executa comenzile ce modifica informatiile din fisierul XML.

Coada de comenzi **CommandQueue** este reprezentata printr-un vector<struct> de structuri ce contin id-ul clientului, descriptorul intors de accept, dar si string-ul ce contine mesajul trimis prin write() de catre client. Cand un client trimite o comanda serverului, aceasta ajunge sub forma unui struct in coada de comenzi cu urmatoarele atribute: cd(client descriptor), id(client id), tm*(time struct pentru a retine cand comanda a fost trimisa), string(mesaj trimis de client).

```
typedef struct thData
{
    int idThread; // the thread's ID kept in evidence by the serv.cpp
    int cl;       // client's socket descriptor returned by accept function
} thData;

struct request_command{
    thData client_id;   //who made the request
    tm *when;           //when was the request made
    string command;     //what did the client request(the stdin command)
};

struct updated_request_command{
    string command_name;    //the command_key to call the command's class
    request_command old;    //copy of old struct(to be sent in fn. execute)
};

vector<request_command>Command_queue;   //command queue
```

Mai departe, coada este impartita in doua cozi diferite, in functie de caracterul comenzilor(fie modifica XML, fie doar citesc din el). Asadar comenzile ce modifica baza de date vor fi executate prima data, in mod iterativ pentru a obtine cel mai recent update dpdv. al timpului. Dupa ce coada de comenzi de actualizare este goala, incepe parsarea comenzilor ce doar preiau date din XML si ce urmeaza a fi trimise spre client. Asadar, acest sistem reuseste sa aduca clientilor date corecte si actualizate in timp real.

Command design pattern-ul este format principal dintr-o clasa abstracta Command ce contine o metoda pur virtuala (**void execute()**) si un destructor. Din clasa abstracta vom deriva 6 clase ce iau numele comenzilor pe care le poate trimite un client. Metoda execute() primeste drept parametri socket descriptorul pentru clientul care asteapta sa citesca si comanda ce urmeaza a fi impartita conform unei sintaxe prestabilite.

Implementarea poate fi vizualizata in imaginea urmatoare:

```
//MAIN CLASS TO DESIGN A COMMAND PATTERN
class Command {…

//Get a TrainID from the client and prints the Train's schedule in that day
class  TrainSchedule: public Command {…

//Get a StationID from the client and prints all the Train which will arrive in the next hour
class  StatusArrival: public Command {…

//Get a StationID from the client and prints all the Train which will depart in the next hour
class  StatusDeparture: public Command {…

//Get a TrainID, StationID, and Time in minuts and updates the server data with a arrival late delay
class ArrivalLate: public Command{…

//Get a TrainID, StationID, and Time in minuts and updates the server data with a arrival early delay
class ArrivalEarly: public Command{…

//Get a TrainID, StationID, and Time in minuts and updates the server data with a departure late delay
class DepartureLate: public Command{…
```

Sintaxa pentru fiecare comanda este prezentata mai jos:

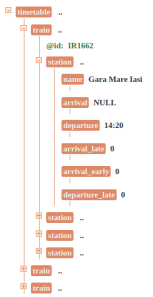
Train Schedule command $-> 1$: TrainID
Station Arrrival command $-> 2$: StationID
Station Departure command $-> 3$: Station ID
Arrival Late command $-> 4$: TrainID : StationID : Time
Arrival Early command $-> 5$: TrainID : StationID : Time
Departure Late command $-> 6$: TrainID : StationID : Time
Exit $->$ exit (se inchide conexiunea client - server)

Fisierul XML timetable.xml contine o structura arborescenta de dispunere a datelor. Radacina principala(timetable) contine mai multi fii acestia jucand rolul trenurilor disponibile in program. Fiecare tren reprezinta radacina, iar statiile prin care acesta trece sunt fii. Pentru a parsa fisierul si a stoca informatiile in structuri de date ce simplifica modul de lucru, vom folosi biblioteca **RapidXml**. Pentru a stoca trenurile disponibile in program, dar si a statiilor folosim doi vectori de structuri: vector<struct Train>, vector<string Station>.

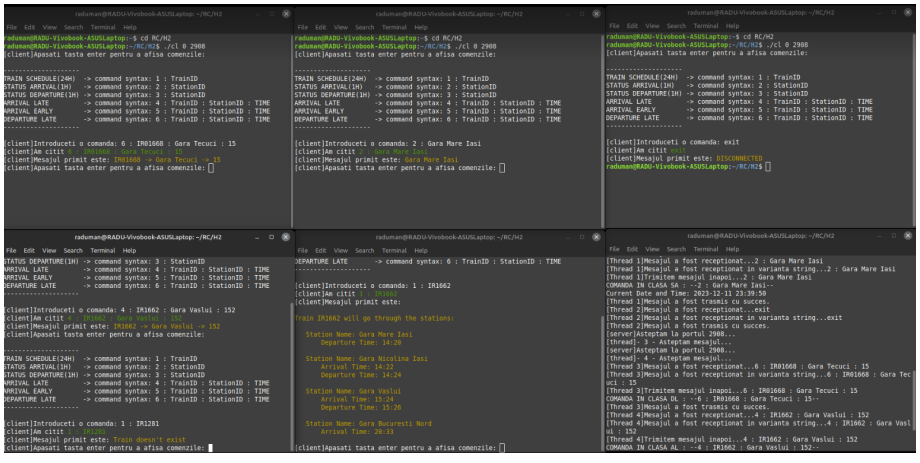
```
struct Station{
    string name;            //station name
    string departure_time;  //departure time in format HH:MM (if it's final st. def:NULL)
    string arrival_time;    //arrival time   in format HH:MM (if it's start st. def:NULL)
    int departure_late;     //departure late in format MMM ex: 120, 60, 5... def:0
    int arrival_late;       //arrival late   in format MMM ex: 120, 60, 5... def:0
    int arrival_early;      //arrival early  in format MMM ex: 120, 60, 5... def:0
};

struct Train{
    string ID;                //train ID ex: IR1972, IR13601, R5009 ...
    vector<Station> stations; //vector of stations
};

vector<Train>Trains;            //vector of Struct Trains
vector<string>Station_map;      //vector with all the Stations names
```



Pentru a testa functionalitatea codului, avem prezentate urmatoarele scenarii:



Concluzii

Versiunea actuala a codului respecta cerintele proiectului si acopera obiectivele stabilite pana acum(implementarea concurentei, primirea si trimiterea de informatii actualizate in timp real, transmise corect si in ordine). Privind mijloacele de imbunatatire a proiectului, putem integra diferite functionalitati ce ofera o experienta user-friendly. In acest sens, putem adauga protocolului de comunicare o comanda **help()** ce afiseaza clientului trenurile si statiile disponibile pentru care poate face interogari sau modificari. O imbunatatire ce ofera un impact impresionant privind experienta clientului cu utilizarea aplicatiei este adaugarea unui **GUI** cu ajutorul bibliotecii Qt sau Gtk C++. De asemenea, putem adauga o comanda **update()** care poate fi apelata doar de un inginer de sistem pentru a actualiza baza de date la un anumit interval orar, sau pentru a modifica structura acesteia(adaugarea de noi trenuri, scoaterea unora din uz...etc).

Referinte Bibliografice

[1] Site-ul disciplinei pentru server/client TCP concurent.
`https://profs.info.uaic.ro/~computernetworks`

[2] Command design pattern.
`https://en.wikipedia.org/wiki/Command_pattern`

[3] TCP diagrama.
`https://en.wikipedia.org/wiki/Transmission_Control_Protocol`

[4] Soft pentru desen schema aplicatie.
`https://app.diagrams.net`

[5] Biblioteca pentru parsare XML.
`https://rapidxml.sourceforge.net`

[6] Vizualizator arborescent pentru fisiere XML.
`https://jsonformatter.org/xml-viewer`