

Mersul Trenurilor^{*}

Tudor Ilade

Department of Computer Science, "Alexandru Ioan Cuza" University, Iasi
tudor.ilade@yahoo.com

Abstract. This paper proposes a design of a concurrent client/server project "Mersul Trenurilor". The project consists of implementing a basic network protocol that contains a server which concurrently handles various types of requests from multiple clients. The communication client-server is implemented using TCP/IP protocol.

Keywords: Mersul Trenurilor · Network · TCP/IP protocol.

1 Introduction

I choose this project because building a scalable app for train scheduling is a difficult task. Many problems can arise during development such as a great architecture for making the code easy to maintain and extend for developers without affecting the performances whilst serving millions of users concurrently.

In this paper I will propose an application design of "Mersul Trenurilor" project using network communication. The main focus is on building a scalable architecture that can be easily maintained by developers and easy to use by clients.

2 Technologies

2.1 TCP/IP Protocol

The project uses as client-server communication protocol the TCP/IP. I am interested to implement a reliable, straightforward way to exchange data without having to worry about lost packets or jumbled data. This feature is important because clients cannot only read information from server, but also can modify existing data from a XML file.

When a client sends a request to the server that modifies data, I want to be sure that the data arrives at server is correct and complete and it is updated accordingly. The integrity and accuracy of data is important, otherwise clients who just want to know when next train leaves, might lose the train because of an incorrect update from a previous request.

Thus, I find TCP/IP protocol the best fit for this project [2] because it is reliable, connection-oriented and full-duplex. The three-way-shake algorithm facilitates the connection between server and client and guarantees data transmission without any loss.

^{*} Project proposed by Continental

2.2 Qt XML C++

The application will perform Read/Write operations from/to an XML file. This operations will be implemented with the help of Qt XML C++ Classes module. [5]

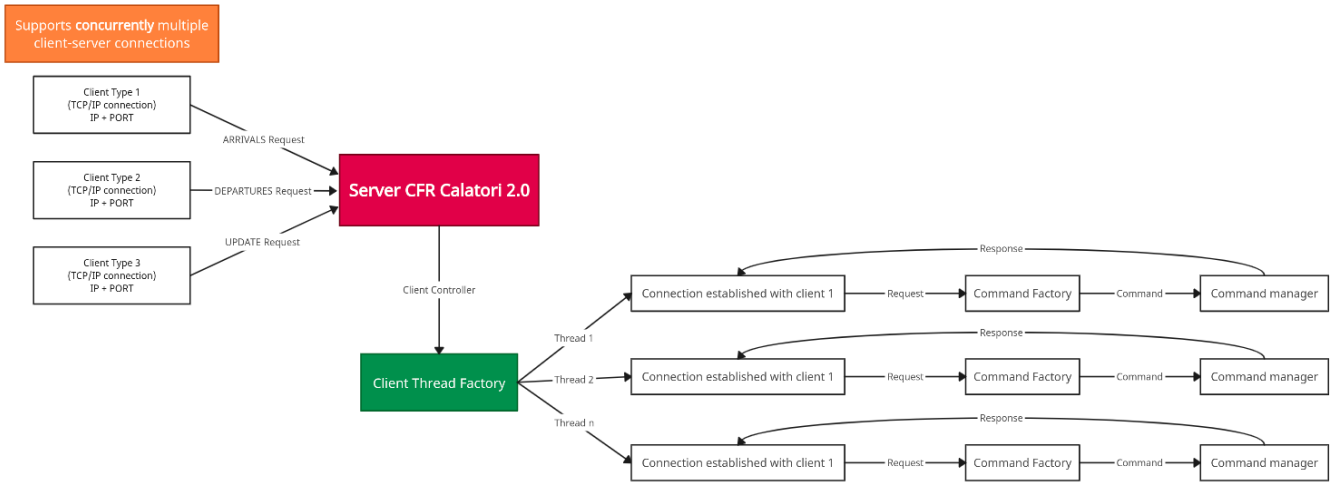
2.3 Requirements

The server has been developed with the help of CMake files. The development process has been conducted using Qt Creator. Before running the server, Qt 6.4.1 version should be installed. [6]

3 Architecture

The architecture of application is presented in the following diagram:

3.1 Server



The application will support two classes of commands:

- Retrieving data
- Manipulating data

Retrieving data commands will consist in GET requests, as follows:

1. Client type 1 → DEPARTURES: Any connected client can ask for information about departures of trains in the current day.
2. Client type 2 → ARRIVALS: Any connected client can ask for information about arrivals of trains from the following hour or current day or for a given train if ID is provided.

For ARRIVALS/DEPARTURES commands, I will implement four flags:

1. *-stationPS*
2. *-stationD*
3. *-fromHour*
4. *-toHour*

The format of ARRIVALS/DEPARTURES command is:

GET *- stationPS < target > -stationD < dest > -fromHour < HH : MM > -toHour < HH : MM >*

GET - type of request. The GET commands are: **ARRIVALS** for arrivals and **DEPARTURES** for departures.

-stationPS - specifies the target railway station from where trains depart/s/arrives. This flag is mandatory, which means that the command will not be executed and it sends back appropriate message.

-stationD - specifies the railway station of interest. When it is used in combination with **-stationPS**, only trains that departs/arrives from target station and head towards station of interest will be displayed. This flag is optional.

-fromHour - specifies the starting hour for displaying arrivals/departures of trains. If it is used, displays trains that arrives/departs in the next hour. (-fromHour + 1h). Default is midnight. If it is not used, will be displayed all the departures/arrivals from that day.

-toHour - specifies the stopping hour for displaying arrivals/departures of trains. If it is used, displays trains that arrives/departs starting from midnight until given hour. If it is used in combination with -fromHour flag, then all trains that arrives/departs in the time frame provided by the 2 flags will be displayed.

Example:

DEPARTURES *- stationPS Cluj - Napoca - stationD Huedin - fromHour 11 : 23 - toHour 23 : 30*

Client type 3 → UPDATE: Any connected client can modify the schedule details of a train by providing delay time in minutes.

UPDATE command has the following the format:

UPDATE *- train < id > -delay < minutes > -fromStation < name > -toStation < name >*

-train - specifies the train ID to which delay time should be added. The train ID is a combination of Type of train and a number (e.g IR322, R3322). The flag is mandatory.

-delay - specifies delay time which should be added to a given train. The delay time has format MM (minutes). The flag is mandatory.

-fromStation - specifies the station from where the delay should be propagated. If it is not specified, the delay will be propagated from first station on the train route. The flag is optional.

-toStation - specifies the station where the propagation of delay time should stop. If it is not specified, the delay will stop at the last station on the train route. The flag is optional.

If **-fromStation** and **-toStation** are not specified, the delay time will be adjusted to the entire train route. Any combination these two flags results in an adjustment only between specified stations. If station specified by **-toStation** is located before **-fromStation**, the command will be considered invalid.

UPDATE command is thread safe.

The server has implemented a manual command which provides information about aforementioned commands and some examples how them should be used.

MAN command has the following the format:

MAN - *info* < *command* >

Command arguments are: ARRIVALS, DEPARTURES and UPDATE.

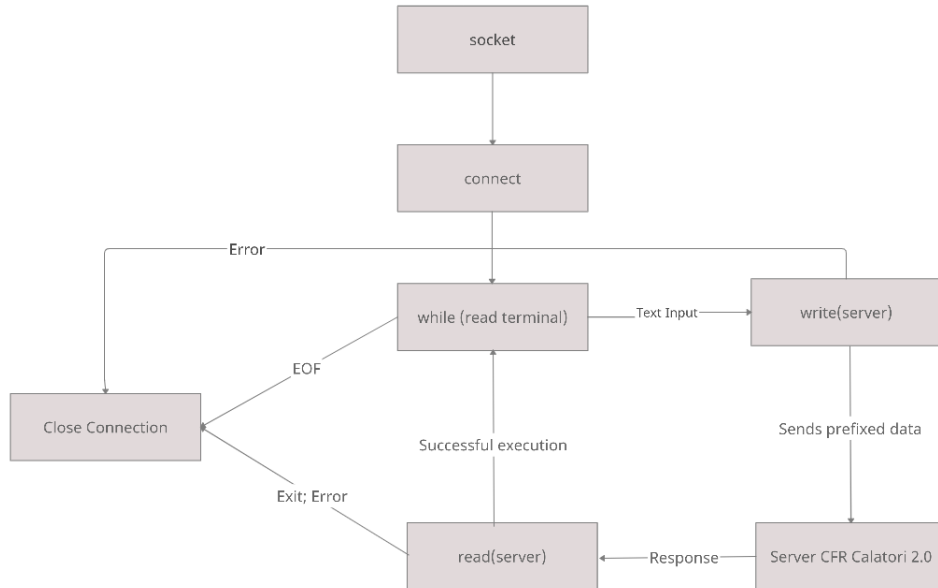
Example:

MAN - *info* **UPDATE**

The command will display information about update command and how it should be used.

3.2 Client

The client architecture is presented bellow:



As the main focus is on developing the server, the Client has a simple implementation that consists of three elements:

1. Reading data from Terminal until EOF is parsed.
2. Sends data to server
3. Read data from server

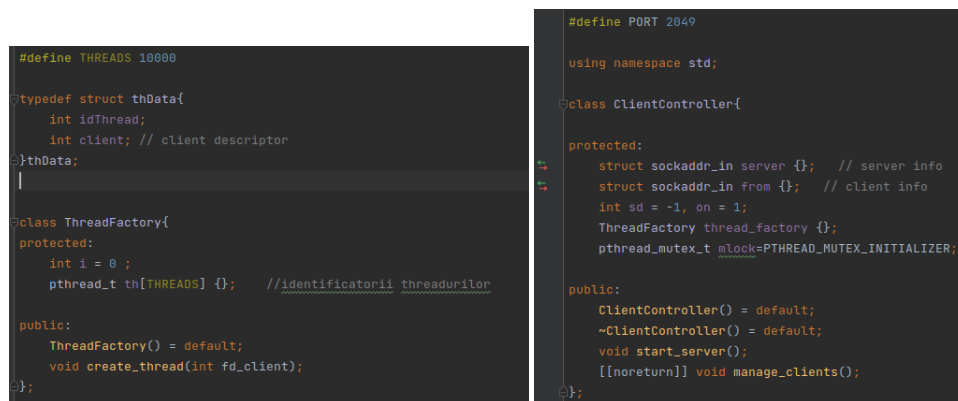
4 Implementation

All clients are served concurrently. The mechanism to serve clients in parallel will be based on multithreading. This mechanism is implemented with the help of a factory design pattern called Client Thread Factory. Clients are served to Client Thread Factory by a Client Controller.

The Client Controller is a class that has the main responsibility of listening and serving new clients, once the connection is successfully established, to Client Thread Factory class.

The Client Thread Factory design pattern is responsible with the handling of new connection requests. The problem with handling multiple connection requests at the same time is tackled with a prethreading protection mechanism that serves requests once at a time and send descriptors to the next empty slot from threading pool. This is achieved with the help of **Accept** primitive. The server uses mutexes for protecting accept() primitive.

The aforementioned design patterns are implemented as follows:



```
#define THREADS 10000

typedef struct thData{
    int idThread;
    int client; // client descriptor
}thData;

class ThreadFactory{
protected:
    int i = 0 ;
    pthread_t th[THREADS] {}; //identificatori threadurilor
public:
    ThreadFactory() = default;
    void create_thread(int fd_client);
};

#define PORT 2049

using namespace std;

class ClientController{
protected:
    struct sockaddr_in server {}; // server info
    struct sockaddr_in from {}; // client info
    int sd = -1, on = 1;
    ThreadFactory thread_factory {};
    pthread_mutex_t mlock=PTHREAD_MUTEX_INITIALIZER;
public:
    ClientController() = default;
    ~ClientController() = default;
    void start_server();
    [[noreturn]] void manage_clients();
};
```

After client is successfully connected to server, clients can now send requests to server. All the requests are handled by a Request Controller. Its main responsibilities are:

1. Receive request
2. Send the request to Command Factory and convert it to corresponding Command
3. Handle the Command to Command Manager
4. Queue the command

Requests are transformed into the corresponding command by CommandFactory class. Commands are implemented with the help of a Command abstract class that is inherited by all available commands. Commands instances store all the relevant info needed for processing, such as client descriptor, CommandResult. Once the request is converted in command by CommandFactory, it is parsed to Command Manager to queue it.

```

struct CommandResult{
    string result;
    size_t size_result;
};

class Command{
protected:
    struct CommandResult resultCommand{};
    int sd;
    string command_t;
    size_t size_command;
    bool incorrectCommand = false;
public:
    Command(const char* com, int sd){
        this->size_command = strlen( $ com);
        this->sd = sd;
        this->command_t.assign( $ com, $ this->size_command);
        this->resultCommand.result.assign( $ "Command test", $ 12);
    }
    Command() = default;
    ~Command(){command_t = ""; sd = 0; size_command = 0; resultCommand = {};}

    struct CommandResult getResult(){return this->resultCommand;};
    virtual struct CommandResult execute(XmlController&) = 0;
    virtual string get_command() = 0;
    virtual bool isCommandValid() = 0;
    virtual bool isElementValid(QDomElement&) = 0;
    virtual TrainData toTrainData(QDomElement&) = 0;
    [[nodiscard]] int get_client_sd() const{return this->sd;};
    [[nodiscard]] size_t get_size_command()const{return this->size_command;};
};

class GetRequests : public Command {...};
class GetArrivals : public GetRequests {...};
class GetDepartures : public GetRequests {...};
class UpdateTrain : public Command {...};
class ExitCommand : public Command {...};
class UnRecognizedCommand : public Command {...};
class ManCommand : public Command {...};

#include "command_ABC.h"
class CommandFactory{
public:
    static Command* create_command(char* command, int sd);
};

```

The result of **GET** request is stored with the help of TrainData class. TrainData class stores information about one route which meet the requirements given by command. After processing the entire XML, all eligible routes are stored in a vector of TrainData instances which is converted into a string. The result of this process is then transformed into a CommandResult instance and sent back to client.

```

class TrainData
{
    /*
     * Class containing needed info about a train
     */
private:
    string numeTren;
    string statieP; // arrival/departure station
    string statieN; // next station
    string statieD; // destination. Last station if not provided
    unsigned int intarziereP; // delay arrival/departure
    unsigned int intarziereD; // delay at final destination
    unsigned int intarziereN; // delay at next station
    unsigned int timpStationareP; // station time at arrival/departure station
    unsigned int timpStationareD; // station time at final destination station
    unsigned int timpStationareN; // station time at next station
    unsigned int timpSosireP; // arrival time at stationP
    unsigned int timpSosireD; // arrival time at stationD
    unsigned int timpSosireN; // arrival time at next station
    unsigned int timpPlecareP; // departure time at stationP
    unsigned int timpPlecareD; // departure time at stationD
    unsigned int timpPlecareN; // departure time at next station
    bool valid = true; // when statieD is provided and we found same statieP but with different statieD than provided

public:
    TrainData(
        string, string, string, string, unsigned int, unsigned int,
        unsigned int, unsigned int, unsigned int, unsigned int,
        unsigned int, unsigned int, unsigned int, unsigned int,
        unsigned int, unsigned int
    );
    TrainData(bool);
    TrainData() = default;
    ~TrainData() = default;
    bool isValid();
    const string toDeparturesString();
    const string toArrivalsString();
};

```

The Command Manager implements a Command Queue where requests are stored and served according to FIFO rule. The Command Manager has its own threading pool from where it constantly checks the queue. If any command is added to queue, it will be popped from top of the queue and sent to execution. Once the result is obtained, it sends the response to the client. Afterwards the Command Manager will look at Command Queue and send to processing the next command.

```

#include <thread>
#include <deque>
#include "../CommandControl/command_ABC.h"

class CommandManager{
private:
    bool running { false };
private:
    std::thread commandThreads;
    std::deque<Command*> queue {};
public:
    CommandManager() = default;
    ~CommandManager();
    void ManageCommands();
    void RunCommands();
    void QueueCommands(Command*);
    void executeCommands(Command*);
};

```

The mechanism of reading/writing the XML file is done concurrently. In the implementation, will be used a locks to prevent data race condition[3] Each client will read the data is available at the time of processing. For update requests pthread_mutex_t mutex is used.

The XML File used is a sample that is actually used in production by cfrcalatori [4].It has the following structure:

```
<?xml version='1.0' encoding='UTF-8'?>
<XmlIf encoding="UTF-8" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespa
<XmlMts>
  <Mt Versiune="1" DataExport="20151228" MtValabilDeLa="20151213" MtValabilPinaLa="20161210"
  <Trenuri>
    <Tren KmCum="57100.0" Numar="2872" Operator="6100826" Putere="E" CategorieTren="R" Lungi
    <Trase>
      <Trasa Tip="Implicita" Id="1" CodStatieInitiala="23428" CodStatieFinala="10938">
        <ElementTrasa Secventa="1" DenStaOrigine="Trgu Jiu" OraP="44400"
        Ajustari="0" Restrictie="0" DenStaDestinatie="Ram. Budieni"
        VitezaLivret="80" StationareSecunde="0" Rco="R" Km="1600"
        Lungime="250" TipOprire="N" CodStaDest="27888" Tonaj="500"
        OraS="44580" Rci="R" CodStaOrigine="23428"/>
        <ElementTrasa Secventa="3" DenStaOrigine="Amaradia" OraP="45000"
        Ajustari="0" Restrictie="0" DenStaDestinatie="Drgueti h."
        VitezaLivret="80" StationareSecunde="60" Rco="R" Km="3400"
        Lungime="250" TipOprire="C" CodStaDest="24587" Tonaj="500"
        OraS="45300" Rci="R" CodStaOrigine="24575"/>
        <ElementTrasa Secventa="4" DenStaOrigine="Drgueti h."
        OraP="45360" Ajustari="0" Restrictie="0"
        DenStaDestinatie="Crbeti Hm." VitezaLivret="80"
        StationareSecunde="60" Rco="R" Km="3000" Lungime="250"
        TipOprire="C" CodStaDest="24599" Tonaj="500" OraS="45600"
        Rci="R" CodStaOrigine="24587"/>
      <ModificariInParcurs/>
    </Trasa>
  </Trase>
  <Grupe/>
  <RestrictiiTren>
    <CalendarTren DeLa="20151213" Tip="Da" Id="1" Zile="16767" PinaLa="20161210"/>
  </RestrictiiTren>
</Tren>
</Trenuri>
</Mt>
</XmlMts>
</XmlIf>
```

The entire I/O operations of the XML are done with the help of XmlController class. The constructor of XmlController opens the XML and loads it to buffer on which the operations are performed. Before any command is executed,

the XML buffer is refreshed to load the latest data, in case any update has been made.

```
#define XMLPath "data/sample_data.xml" // it works by providing another absolute path too, this is standard
using namespace std;

class XmlController
{
/*
 * Xml Controller class

Class responsible by handling the xmlFile with I/O operations.
*/

private:
    string xmlPath;
    size_t pathSize = 0;
    bool opened = false;
    QDomDocument document;
    QDomElement trenuriNode;

public:
    XmlController();
    XmlController(char*);
    QDomElement getTrenuriNode();
    QDomDocument& getDocument();
    bool writeDocumentOnDisk(QDomDocument);
};
```

The communication between client and server are in both way prefixed with the length of data ready to be sent. The prefixed communication consists in writing the size of data as integer at the beginning of each message, followed by data.

Scenario I will present some scenarios how the application should be used and how it behaves when certain commands are sent. Let us suppose client sends an invalid request, meaning a command that is not recognized by server. In this case, server will receive the request and will convert it in an `UnRecognizedCommand`. After the command is queued, it starts execution and sends back to client an appropriate message.

In case client wants to close connection, has to send an exit request that will be processed by server. The server shuts down the connection with client once executes all commands from queue and sends all responses to client. Once EXIT response is received from server, the connection is shutted down. In case if a connection is closed abnormal (i.e. SIGINT, SIGTSTP or SIGTERM), a handler is attached to these signals which ensures that the connection will be shut down correctly on the server side.

Another edge case is when the server crash for some reason and is closed unexpectedly. In this scenario, all the connections with all clients are closed and all threads are joined.

Let us suppose the user sends a command where he or she duplicates the the flags and arguments. For example, consider the following command:

DEPARTURES – *stationPS ClujNapoca* – *stationPS Huedin*

In this case, the command will be considered valid. The behavior is that the server consider only the last argument from the chain of arguments when they

are the same. Here, only departures from Huedin will be displayed, and not Cluj Napoca. If a flag is invalid, the command will be considered invalid, even though the other ones are valid.

5 Conclusion

This is my proposed implementation of the Mersul Trenurilor project. Further improvements can be done by adding more features to it. Some possible features that can be added are:

→ The possibility of client to choose the type of connection they want (TCP/IP or UDP) for GET requests. In this case, usually the client wants to know fast some details about a train. In majority of cases this protocol will yield the desired result without data loss. And even so, the Client can send another request if he or she sees that nothing has been returned. The majority of requests of this server will be of type GET, rather than PUT or CREATE. Thus, the feature can result into much faster exchange of information.

→ A backup mechanism implemented for XML file to make sure that in case of failures, data is backed up periodically and prevents massive data loss.

→ A secured connection by providing a TOKEN in case of writing data on XML. The TOKEN can be generated uniquely in UUID4 format, thus only trusted users can perform Write operations.

References

1. W. Richard Stevens, Bill Fenner, Andrew M. Rudoff, UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API, 3rd. ed. Publisher, Addison Wesley (2003)
2. Computer Networks Homepage, <https://profs.info.uaic.ro/~computernetworks/index.php>. Last access 1 dec 2022
3. Mutex lock for threads, <https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/>. Last access 1 dec 2022
4. XML sample from government of Romania <https://data.gov.ro/dataset/c4f71dbb-de39-49b2-b697-5b60a5f299a2/resource/d0ec6682-5c00-4666-89dc-6a5e7831b8dd/download/mers-trenntfc2015-2016.xml>. Last access 25 dec 2022
5. Qt XML docpage, <https://doc.qt.io/qt-6/qtxml-module.html>. Last access 26 dec 2022
6. Qt downloader - <https://www.qt.io/download>. Last access 26 dec 2022