Mersul Trenurilor* - a proposed concurrent server

Tudor Ilade

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

Abstract. This paper proposes a design of a concurrent server for the project "Mersul Trenurilor". The projects consists on implementing a basic network protocol that contains a server which concurrently handles various type 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 it seems to me that it has an important practical application and I am interested to see what problems can arose during its development and how should I tackle efficiently design problems.

In this paper I will propose an application design of "Mersul Trenurilor" project using network communication. The main focus will be on developing a scalable and concurrent architecture of the server.

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 loose 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 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 SDK

I aim to wrap the client-server communication around a GUI implemented with the help of Qt SDK for C++.

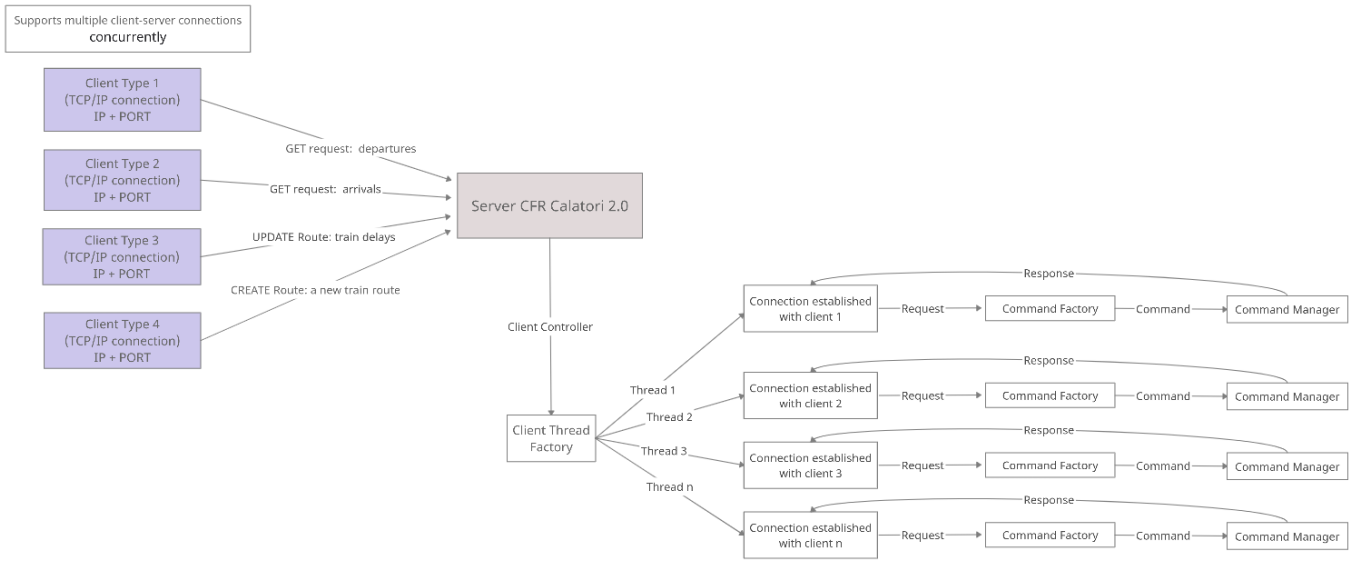
2.3 Qt XML C++

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. [4]

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 → GET DEPARTURES: Any connected client can ask for information about departures of trains in the current day.
2. Client type 2 → GET 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 GET ARRIVALS/DEPARTURES commands, I will implement three flags:

1. $-hour$
2. $-id < train\_id >$
3. $-station < station\_name >$.

If no flag provided, the server will yield all train schedules. $-hour$ flag will ask for arrivals/departures in the next hour. $< train\_id >$ will ask for arrivals/departures of the given train id, in the given day. A combination of both will result in getting the arrivals/departures of the given train id in the next hour. If any delays are present at the request time, will be reflected in the result message with estimated time of arrival/departure. $-station$ flag can be added with any combination of the other two flags and will show the departures/arrivals of the given station.

1. Client type 3 → UPDATE ROUTE: Any connected client can modify the schedule details of a train by providing delay time in minutes.
2. Client type 4 → CREATE request: A client with CREATE permissions will be able to add a new route.

UPDATE ROUTE and CREATE ROUTE commands will have the format:

$$\rightarrow \textbf{UPDATE ROUTE} \quad < train\_id > < delay\_time > .$$

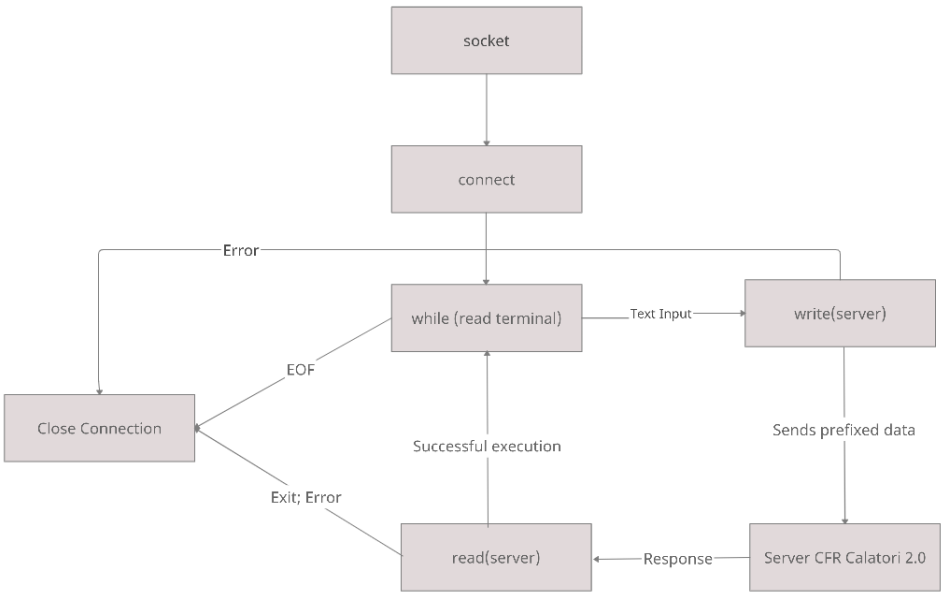
$$\rightarrow \textbf{CREATE ROUTE} < train\_id >$$

CREATE ROUTE command will check first for the provided train id whether exists in XML file. If it doesn't, then a dashboard will appear where he or she will have to insert the following details, comma separated:

→ $station_1$-*departs-arrival-delay*, $\cdots$,station$_n$-*departs-arrival-delay*

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

However the Client will have implemented a GUI with the help of Qt C++ module to wrap the entire process of send-receive request around an application interface.

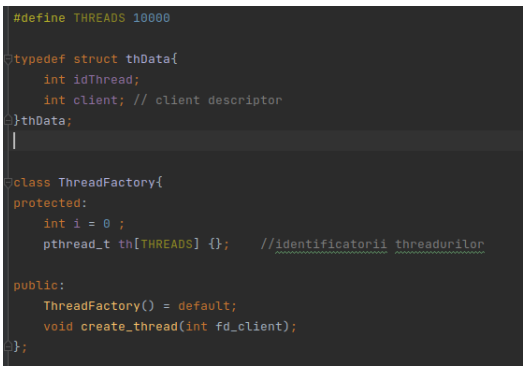
4 Implementation

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 responsability 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.

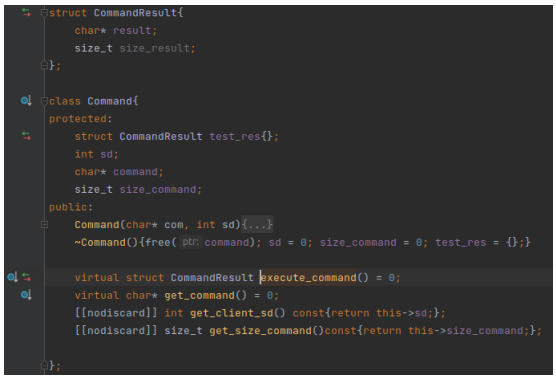
The aforementioned design patterns are implemented as follows:



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 responsabilities 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.



The Command Manager implements a Command Queue where requests are stored and served according to LIFO rule. The Command Manager has is 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.

```cpp
#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 will be 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 the write requests, the critical section will be protected with the help of flock struct from fcntl header.

The XML file will have the following structure:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <Trains>
    <Train>
        <id>Train Id<\id>
        <stations>
            <station id=1 departs=14:01 arrives=0 delay=0>Ramnicu Valcea<\station>
            <station id=2 arrives=14:34 departs=14:36 delay=0 >Pitesti<\station>
            <station id=3 departs=0 arrives=16:01 delay=10 >Bucuresti Nord<\station>
        <\stations>
    </Train>

    <Train>
        <id>Train Id<\id>
        <stations>
```

```
            <station id=1 departs=06:21 arrives=0 delay=10>Iasi<\station>
            <station id=2 arrives=08:43 departs=08:44 delay=12 >Focsani<\station>
            <station id=3 departs=0 arrives=14:55 delay=7 >Bucuresti Nord<\station>
        <\stations>
    </Train>

    ...

    <Train>
        <id>Train Id<\id>
        <stations>
            <station id=1 departs=02:01 arrives=0 delay=0>Cluj-Napoca<\station>
            <station id=2 arrives=03:23 departs=03:30 delay=0 >Sighisoara<\station>
            <station id=3 departs=0 arrives=06:42 delay=10 >Brasov<\station>
        <\stations>
    </Train>
  </Trains>
</xs:schema>
```

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 the following scenario: 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 transform the request in an UnRecognizedCommand. After the command is queued, it starts execution and sends back to client an appropiate 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.

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 accessed 1 dec 2022
3. Mutex lock for threads, `https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/`. Last accessed 1 dec 2022
4. Qt XML docpage, `https://doc.qt.io/qt-6/qtxml-module.html`. Last accessed 1 dec 2022