Tehnical Documentation MersulTrenurilor

Valentin Roşca
Grupa B2

Faculty of Computer Science of University "Alexandru Ioan Cuza"

Abstract. This project involves the homework for the Networking class at the Faculty of Computer Science of University "Alexandru Ioan Cuza" and also a solution for the poor train timetable information problems in my country. This paper describes the basic layout of data structures and the process of serving a request and sending it back to the client. This document does not describe any of the algorithms used for different types of requests, only how to server divides it's tasks and code description.

1 Introduction

1.1 Overview

A server that updates and offers information in real time for all connected clients for:

- arrival status
- departures status
- delays and arrival estimate
- path between stations

1.2 Purpose

1.2.1 Business Perspective The app offers clients a way to consult train timetables from their home devices like PCs and laptops and to find out the best possible route to take between two cities.

1.2.2 Motivation The motivation comes from the possibility to develop my coding skills and distributed algorithms thinking and also from a real need of an application to make the process of staying in touch with train timetables and finding to optimal route effortlessly.

1.2.3 Considerations Although the application has read usage in the real world, there are many applications similar and they support a wider range of devices, proving that this is not the perfect solution for all the problems.

1.3 Conventions and Abbreviations

1.4 Coding style

The coding style used for the C++ source code is the Google C++ Style. For more details, visit the address specified in the references section.

1.5 Abbreviations

1.5.1 UML The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering, that is intended to provide a standard way to visualize the design of a system.

1.5.2 TCP

The Transmission Control Protocol (TCP) is one of the main protocols of the Internet protocol suite. It originated in the initial network implementation in which it complemented the Internet Protocol (IP). Therefore, the entire suite is commonly referred to as TCP/IP.

2 Technologies and Programming Languages

2.1 Programming Languages

2.2 C++14

The main programming language used for the client, server and UI implementation is C++14. The motivation for using this programming language is that it's fast, reliable and comes with a good documentation and support for multithreading and low-level implementations.

2.3 Cypher

This is the language used for retrieving and storing the time tables data. This language comes with the nod4j database.

2.4 Networking

2.4.1 TCP Sockets

For the communication between clients and the server is provided by sockets. The TCP protocol is preferred over UDP because the connection must be kept alive and there shouldn't be any data loss.

2.5 UI

The user will interact with the application through a graphical user interface (GUI).

2.5.1 QT

QT is a graphical user interface library that is fast, reliable and comes with a C++ driver.

2.6 Databases

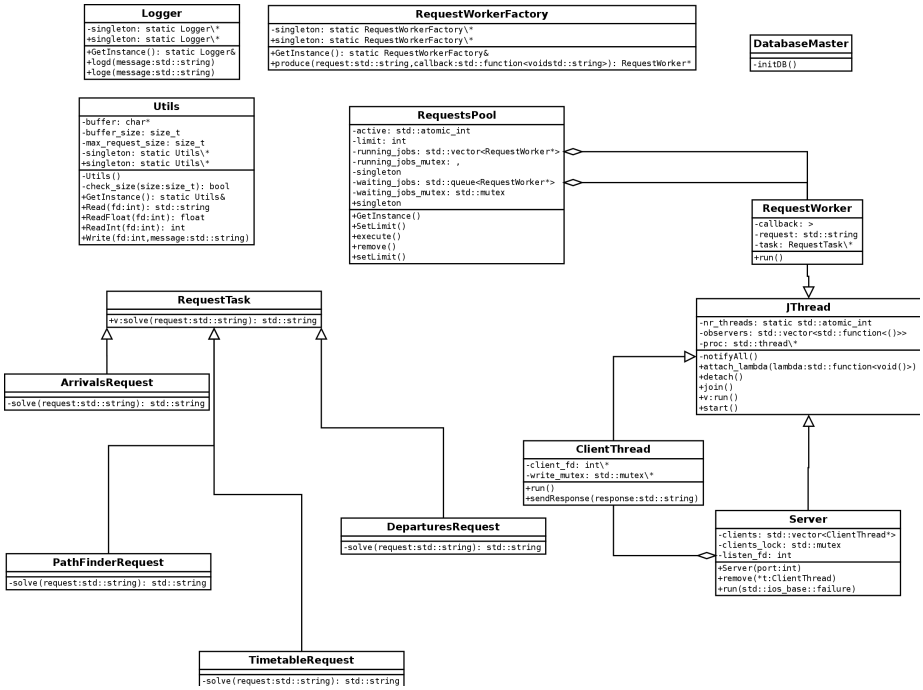
The data will be kept in a graph structure with stations being the nodes and routes being the edges. This allows to find the shortest path faster than relational models and the data is stored in a natural way for the problem at hand.

2.6.1 Nod4j

Nod4j is a highly scalable, native graph database purpose-build to leverage not only data but also its relationships.

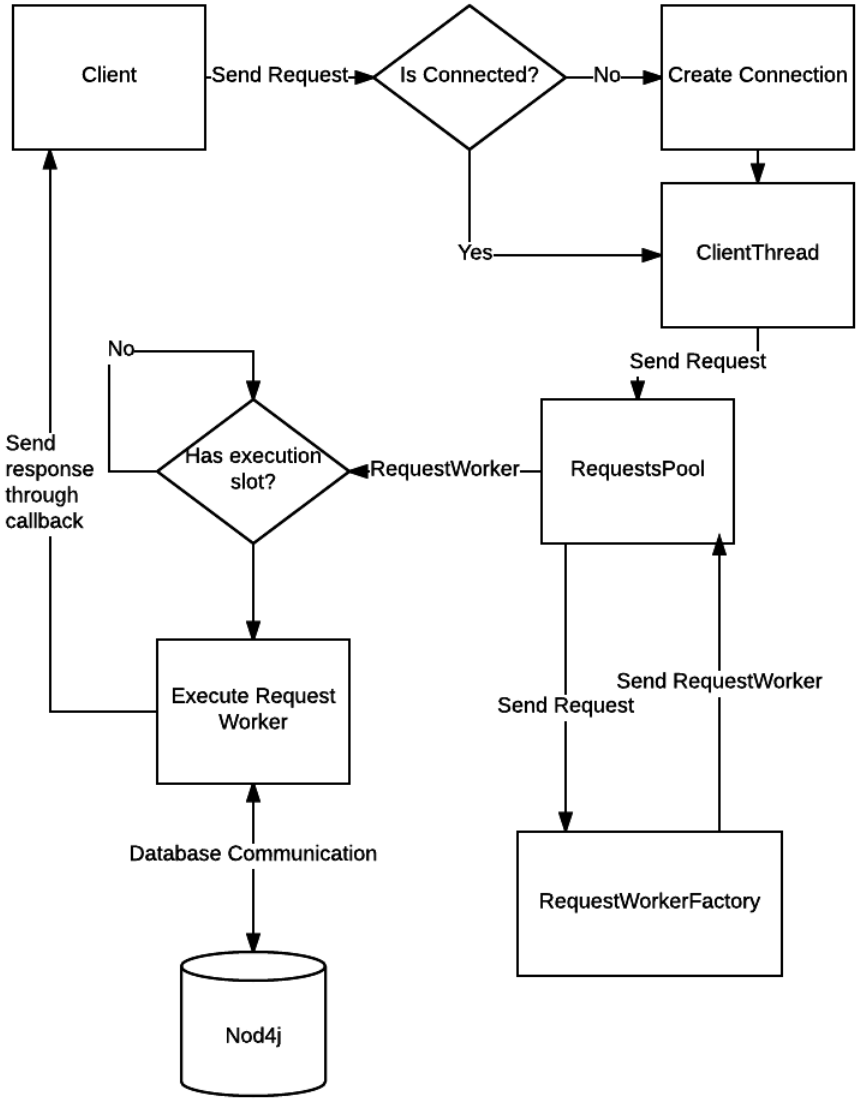
3 Application Architecture

3.1 UML



This UML contains the major classes and intefaces for the server side. Because of it's simplicity, the client side does not have an UML diagram.

3.2 Diagrama Aplicatiei



4 Implementation Details

4.1 JThread

JThread is a wrapper class for std::thread that allows for easier management and storage through polimorfism of threads. It also supports lambda functions as observers. The notifyAll is called when the run function finishes.

```
class JThread {
        static std::atomic_int nr_threads;
        std::thread *proc; // the thread object
        void notifyAll (); // notify all observers
public:
        // calls the run function on another thread
        void start ();
        void join (); // joins the thread if is joinable
        void detach (); // detach the thread

        // main thread logic - this must me implemented
        // by derivate classes
        virtual void run () = 0;

        // attach lambda observer
        void attach_lambda(std::function<void()> lambda);
};
```

Implementations for the functions above (abstract class JThread):

```
std::atomic_int JThread::nr_threads{0};

void JThread::join () {
        if (proc->joinable())
                proc->join();
}

void JThread::start () {
        nr_threads++;
        proc = new std::thread ([this](){run();notifyAll();});
}

void JThread::detach () {
        proc->detach();
}

void JThread::attach_lambda(std::function<void()> lambda) {
        observers.push_back (lambda);
}

void JThread::notifyAll () {
        for (auto i : observers) {
                i();
        }
}
```

4.2 Server and ClientThread

The Server class extends the JThread class and contains the main server logic. This includes

- socket creation
- port binding
- listening
- creating ClientThreads when a new connection arrives

```
listen(listen_fd, 5);
while (true)
{
        // accept client
        int client_fd = accept(listen_fd, (struct sockaddr *)NULL, NULL);
        Logger::GetInstance().logd("Client accepted. Reading from socket " +
             std::to_string(client_fd));

        if (client_fd < 0)
                throw std::ios_base::failure("Cannot accept client");

        // devirate class of JThread controlling the loop that
        // keeps the client connection alive and forwards the requests
        ClientThread client(client_fd);
        clients.push_back (client);
        //remove client when connection is dropped
        client->attach_lambda([this, client]{ this->remove (client);});
        client.start(); // start receiving
}
```

ClientThread is another JThread extension and represents a worker thread that manages a connection with a single client. The purpose of this class is to capture user requests and pass them to the RequestsPool to be solved.

4.3 RequestsPool, RequestsWorker and RequestTask

RequestsPool uses the command design pattern to solve incoming requests and through callbacks send back the response. This implies that we need to alocate a limited number of workers threads (RequestWorkers extends JThread) with the task at hand and remove them when the execution is finished.

Sample for Command Design Pattern job queuing:

```
worker->attach_lambda([this, worker](){this->remove(worker);});
if (active < limit) { // check for limit of workers
        worker->run ();
        running_jobs_mutex.lock ();
        running_jobs.push_back (worker);
        running_jobs_mutex.unlock ();
}
else {
        waiting_jobs_mutex.lock ();
        waiting_jobs.push (worker);
        waiting_jobs_mutex.unlock ();
}
```

The RequestTask is an interface for the request types that the user can request. Derivate classes can be found in the UML.

```
class RequestTask {
public:
        RequestTask() {};

        // each derivate class must implement this method
        // to serve the incoming request
        virtual std::string solve (std::string request) = 0;
};
```

5 Conclusions

The potential of the project is great but its limitations, the devices it can support and the lack of support for constant database updates can be solved by hiring employees and offer a website for the clients to support multiple devices.

As for the technologies used, I learned to use graph database, integrate them with C++ language and schedule and scheme multi-threaded operations and concurrent access to the database.

References

1. Neo4j - https://neo4j.com/
2. Latex Manual - https://en.wikibooks.org/wiki/LaTeX
3. C++ Docs - http://www.cplusplus.com/
4. TCP/IP https://en.wikipedia.org/wiki/Transmission_Control_Protocol