Train Times & Timetable

Volentir Alexandra

January 2021

Abstract

This project was created with the purpose of having an easy-to-use, fast and interactive concurrent server, that will emulate the process of "train scheduling" and even train moving. The server will have access to an xml file that will give us information about the routes, number of trains, arrival times, personal code numbers, working days, route distances, arrival times and etc., so that our train dispatcher can make a comparison between the arrival predictions from the xml file and the in-time changing schedule of arriving trains, and, as a result, make its estimations about the time needed to wait for a train to arrive at a station.

1 Introduction

This report is designed to demonstrate the implementation of a concurrent server[6] of a *"Train timetable"* project on a regional level.

We will implement a concurrent server which would fetch and send back live information about ongoing routes taken by the trains. So, there would be a smart *concurrent* server and two types of clients: the *train drivers* and the *dispatcher.*

Train conductors. They will give concrete information about when they have *started* their trains with a manual push, about when they arrive to a station, so that the server can calculate its estimation on time arrivals and everything else accordingly.

Train dispatcher. It fetches from the server information about ongoing delays, leavings and arrivals. It has a protocol through which he communicates with the server.

The server. The server would receive information from the train's movement and calculate if the train arrives differently than it's prefixed schedule depending on the calculations of the server based on the *train driver's* received information. It will make and send its estimations about *departures status, arrivals status, delays and arrival estimates* to the *train dispatcher.* All the *logic* and actual *reasoning* will take place in the server. It will fetch information about the routes, stations, train particular information, planned arrival time for each machine/client from an XML file. It will parse and compare *pre-scheduled* data to the reality and send it to the *dispatcher* at the time intervals the client will ask for it.

2 Implemented Technologies

2.1 TCP - transmission control protocol

The TCP protocol was used in this project to ensure a reliable form of transmitting data between the client and server.

TCP is reliable, ***connection-oriented, full-duplex*** *and* ***safe*** *as it transports streams of bytes, using a complex transmission mechanism*[5]. The three-way-handshake[1] creates an established connection before sending data and takes care of proper **closing**, data **sequencing**, **re-transmission** in case of failure and eventual **error checking**. TCP is pretty simple to use, as you don't need to think how to handle dropped data-grams, or about the fact that your data-gram will arrive to the destination in general, and your client will get the sent information(as with UDP)[2]. So with TCP you can focus more on the code because the data transmission is reliable and straightforward.

So, we've discussed before that the *three-way handshake* is a method used to make a reliable and established connection between our clients and the server prior to sending any messages. The three-way-handshaking algorithm can be resumed mainly to the work of these three primitives: *bind, listen* and *accept.*

• bind() - "attaches a local address to a socket"[5], traditionally called *assigning a name to a socket.*

• listen() – "allows a socket to accept connections", listen has a queue of incoming connections, and as soon as accept() created the socket descriptor, the queue frees and receives other clients. My listen() has a queue of 5 waiting connections.

• accept() – "blocks the caller until a connection request appears". "accept" will create a new socket descriptor for each arriving client.

```
bind(sd, (struct sockaddr *)&server, sizeof(struct sockaddr);
listen(sd, 5);
client = accept(sd, (struct sockaddr *)&from, &length));
```

2.2 XML parser - pugixml

In order to parse my XML file with the route scheduling, I'll use a lightweight and fast XML-parser from a third party library, called pugixml[4]. The code is distributed under the *MIT license.* The thought behind this parser is that it stores every piece of data in a tree so that later you can load it from the C++ I/O stream and traversed using built-in expressions. The structure is pretty flexible as you can change the value of the attributes at any time.

3 Application architecture

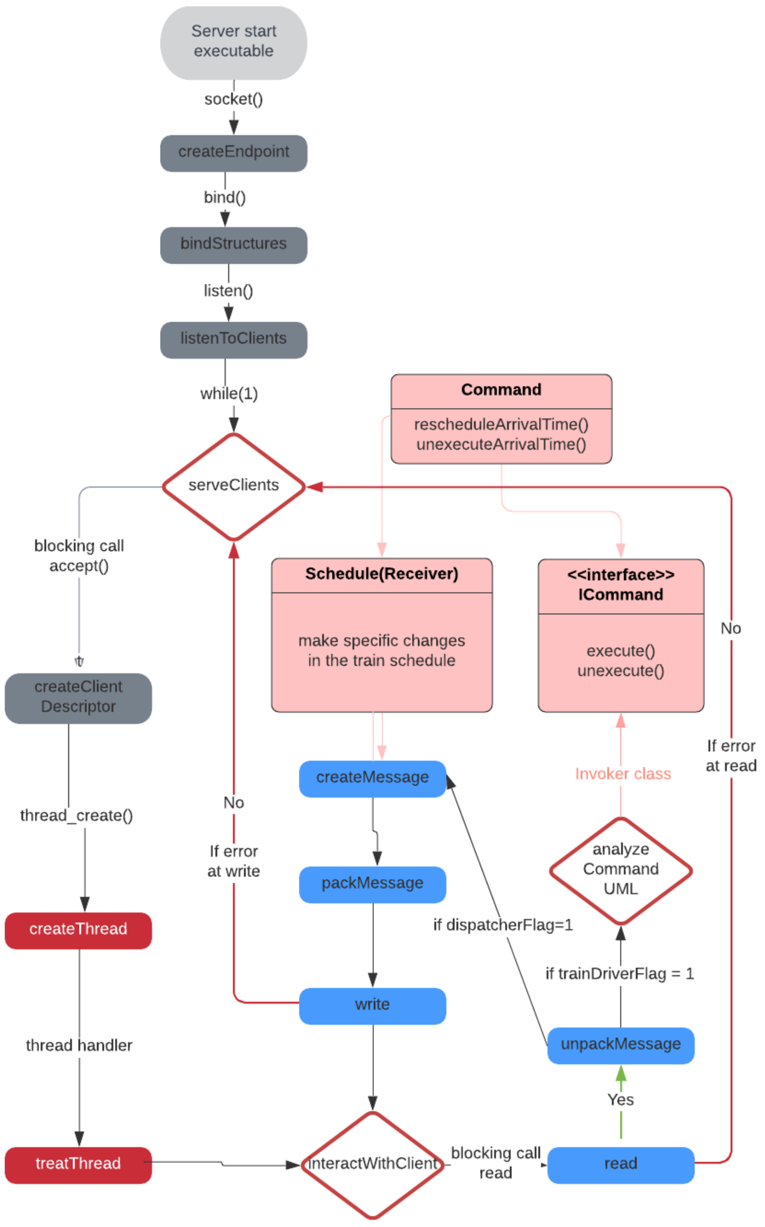

Figure 1: Architecture of the server

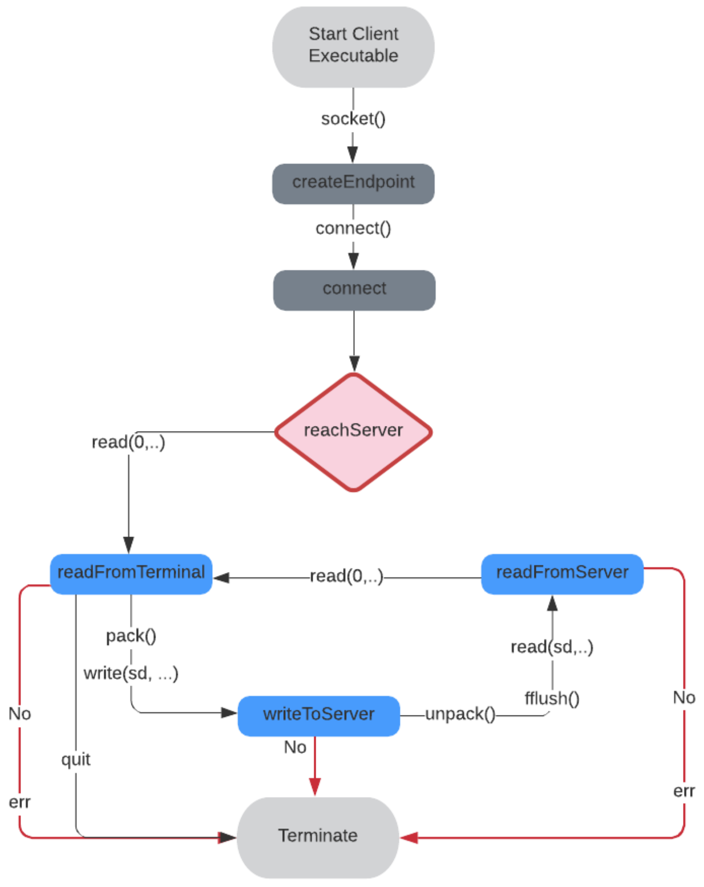

Figure 2: Architecture of the client

3.1 Server

1. The server starts by introducing the *three-way handshake*, then it enters in an infinite while loop with the function serveClients. The loop is infinite as the server is supposed to run permanently.

2. As it enters the infinite loop, the server creates the *socket descriptor* needed for the incoming client. This call is blocking, so until listen hasn't received any connection requests in it's queue, the accept won't compile.

3. Within the loop, we create a thread for each *client socket descriptor* previously instantiated:

```
pthread_create(&th[thNumber], NULL, &treat, td);
```

where *td* is a struct with information of the specific thread passed to the function *thread_create*. *treat* is a thread handler, which returns a void pointer - (NULL) and is passed a void pointer as well. In this way as clients enter the loop and are created threads[8] to analyze and make a response for their commands, we can say that the server is *concurrent*, because it handles all the clients *at the same time*.

4. In the *treat* function we have an infinite loop with in which the server communicates with the client with the blocking functions *read* and *write* and between those encapsulate the request as an object and setting it in a queue of requests. If read or write give a return value ¡= 0 than either our client was to quit itself, or it disconnected unexpectedly from the server, so we will return the value -1 and break the infinite loop exiting the function, immediately closing the client socket descriptor. So in case we return recursively -1, returning to the *serveClients* function thus doing the same thing we did before - serving the remaining clients, but having more kernel space for serving them.

5. The functions *pack* and *unpack message* prefix the message with the number of characters to be read. The client knows that it should read the first four bytes in order to know what string to read, and the client knows and well.

6. With the functions *Invoker* (called *TrainRemoteControl*), *Schedule* (or basically the *Train* class), *Delay* (or *Command*) and interface *ICommand*, we can take a bunch of commands from the client, encapsulate them and compost them into *queue*, or as in my case, *stack* of commands. Then if you need to, you uninvoke them one by one and have a sort of *batch command* - a command that executes multiple commands or this *macro* command of smaller doable actions - we can summarize this whole process as a dependency injection.

Here is a more academical description of the command pattern and what it does:

The command Pattern encapsulated a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.[3][7]

3.2 Clients

1. The client creates as well an endpoint with *socket* and connects to TCP using *connect*. Then the client enters in an infinite loop and from there he will do step by step the reading from the reading from the user, writing the *request* to the client.

2. In the first read the *dispatcher* and the *train drivers* need to authenticate themselves.

3. From the design of the client, it doesn't make sense either you're the train driver or the dispatcher, because all the logic of flags is done in the server when you authenticate. Further, is just an exchange of commands.

4. The *dispatcher* has access to all the possible commands, but it can't modify any, whereas *the train drivers* have unlimited access to change the xml attributes from the tree but because each train has its particular route and streets, arrival time and personal code etc., there will be indeed no race conditions from the *train drivers*.

5. The train drivers access the terminal in the same way as dispatcher and give a manual push from the terminal to start the train, as well as a push for going to the next station. If you push the train and don't conform to the times in the schedule, the time of your given push will automatically have a ripple effect on all the other times and schedulings connected with the train.

6. Bear in mind that there could be a lot of train drivers but only one dispatcher. Having only one main dispatcher frees us of the need to use a lot of *locks* and *mutexes*, because if there is only one dispatcher there will be no race condition on the xml data and attributes, and thus, it will make our code a lot easier and faster, because locks take a lot of time[9].

7. As well as in server, there are the *pack* and *unpack* functions, which help send the data in a more elegant and compact way.

4 Implementation details

4.1 Protocol

First and foremost, the **protocol** between the server and client includes the following characteristics:

- The messages are sent with the prefixed length before the actual message on a space of 4 chars
- the messages themselves are strings with a maximum buffer of 6000 characters
- in client and server, before analyzing the message sent through TCP, they unpack the string using a special *unpack* function, which first reads the dimension of the message from the first 4 characters, and only then begins to read the message, allocating for reading the exact number of bytes that it had read from the prefixing.
- after analyzing and deciding what answer to send back, both server and client pack their message on a maximum grid of 6000 bytes. First the program calculates the length of the message, prefixes it and attaches the rest of the message.

The actual protocol contains the following commands:

Train specific commands:

• "login : train<nr>" – logins the train. You need to enter the number of the train you want to login at the end. The number of trains ranges from 0 to 43. Obs: you can't effectuate any *manipulation* with train time or moving until you've logged in. So loggin in should be your first step.

• "start" – starts the train with which you logged in. Pressing "start", you give the train a manual push. Obs: you cannot effectuate "delay", "undo", "reset" operations if you haven't started the train yet. Once you started, your status will be seen as *active* or *Running* and the dispatcher can see that you started the train. Your arrival, leaving time and other data for each station will be automatically recalculated by the server.

• "delay+<minutes>" or "delay-<minutes>" – at each stop you should enter the delay in order to give the train the push to the next station. It can be related with technical issues, actual delay or passenger waiting. A positive delay is the actual delay. The negative delay means that the train arrived earlier at the station. If you have no delay you can simply enter "delay0".If you enter multiple delays different from 0, their sum will build up over time. After each delay, the arrival and leaving time for each station will be automatically recalculated. The dispatcher can see what are the expected times you arrive at each route, your current location and practically any estimations done after you change the predefined arrival time. The dispatcher can do it with the commands "get -all" or "get agenda train<nr>", "get -today -all", "get -hour -arrivals" and "get -hour leavings". Still, the train doesn't have access to these commands, it can only change the time data for itself with "delay+/− <minutes>", "undo" and "reset".

• "undo" – the undo operation undos the delays that you entered. If you did 3 delays with the values +4, +5, +12, an undo operation will substract 12 from the total delay time. The second undo will substract 5 and so forth. Basically, one undo means one step back. The arrival and leaving time will be recalculated after each undo.

• "reset" – resets the total delay value to 0, and, respectively, recalculates the arrival and leaving time accordingly. It practically cancels all the delays you did before. The arrival and leaving time will be recalculated accordingly and the total delay time of that particular train will be set to 0.

• "logout" - logs out the user (the login flag is set to 0). All the data on that particular route will be lost. He can no longer access train-specific commands. The train will be stopped and all the data regarding the train arrivals and delays will be reset to normal.

• "quit" or "q" – the thread of the client stops it's execution and you quit the application. All the data on the route will be reset to default values.

Dispatcher specific commands:

• ”login : dispatcher” – logins the dispatcher, giving the login flag the number of the thread on which the dispatcher is connected. There can be only one dispatcher, so you can’t connect from another thread until the dispatcher is active on a thread.

Fetch information on trains:

• ”get agenda train<nr>” – gets the table for the current situation of the train (nr of the stations, stations name, the number of minutes until the next station, actual delay, time until reaching terminus, and the time of leaving)/

• ”get info train<nr>” – outputs information about the nr assigned to the train, the origin and destination stations, the personal code, the passed kilometres, the length of the train, the code of the owner, the rank and tonnage.

• ”get line train<nr>” – fetches the basic route of the train with the stations numerotated in the order of their traversal.

• ”get -all” – fetches ALL the information on the current schedule. Precisely, the origin and destination station names, the date and time of leaving, the name of the station, the nr of the station in the route, delay in minutes, and state (”Sleeping” when the train haven’t been started yet and ”Running” when the train was started and is active).

• ”get -today -all” – similar to ”get -all” but it only fetches information about trains that run from this hour on till midnight.

• ”get -hour -leavings” – fetches the information on the leavings that will take place THIS hour. Precisely, the information fetched will include: the origin and destination station names, the date and time of leaving, the name of the station, the nr of the station in the route, delay in minutes, and state (”Sleeping” when the train haven’t been started yet and ”Running” when the train was started and is active).

• ”get -hour -arrivals” – fetches the information on the arrivals that will take place THIS hour. Precisely, the information fetched will include: the origin and destination station names, the date and time of arrival, the name of the station, the nr of the station in the route, delay in minutes, and state (”Sleeping” when the train haven’t been started yet and ”Running” when the train was started and is active).

• ”get info trains” – fetches information about ALL the trains, including: the order number assigned to the train, the origin and destination stations, the personal code, the passed kilometres, the length of the train, the code of the owner, the rank and tonnage.

Quit:

• ”logout” – logs out the dispatcher, he will no longer be able to fetch data about the trains until he logs in again.

• ”quit” pr ”q” – quit the server. It automatically logs you out.

General commands:

• ”m” - lists the available commands on train & dispatcher.

• ”man” - gives a detailed explanation of each command with details on how to use it.

4.2 Use cases

1. The *dispatcher/train driver* leaves the server unexpectedly $\Rightarrow$ the server doesn't die, because in case the client disappeared unexpectedly, the functions will return recursively -1, and the server will finally close the socket descriptor of the client, in that way killing the thread and returning to the serveClientsConcurently function and have now more kernel resources to serve other existing clients.

2. The user introduces an inexistent command - in that case the server will return the message "inexistent command" and with the immediate next loop iteration the client will be able to give another (maybe correct) response.

3. The server sends a partially correct command - the server helps the client giving him options for computation

4. If a train leaves the server unexpectedly (it had a crash) or an accident, all the data on the train moving on that day will be erased or reset to default.

5 Conclusions

As an optimization, my clients could have a graphical interface and more control over the data from the train. Also, I would like to implement in the future the Dijkstra's algorithm to find optimal routes for clients' route search.

References

[1] Sunny Classroom. *TCP - Three-way handshake in details.* `https://www.youtube.com/watch?v=xMtP5ZB3wSk&ab_channel=SunnyClassroom`, 2019.

[2] Matt Cook. *TCP vs. UDP: What's the Difference?.* `https://www.lifesize.com/en/blog/tcp-vs-udp/`, 2017.

[3] Kathy Sierra Bert Bates Eric Freeman, Elisabeth Freeman. *Head First Design Patterns.* OReilly, 2008.

[4] Arseny Kapoulkine. *Pugixml - Light-weight, simple and fast XML parser for C++.* `https://github.com/zeux/pugixml`, 2006-2020.

[5] Alboaie Lenuta. *Transport Level Cr.* pages 29–45, 2021.

[6] Alboaie Lenuta and Panu Andrei. *Computer Networks 2021.* `https://profs.info.uaic.ro/~computernetworks/ProiecteNet2021.php`, 2020-2021.

[7] Christopher Okhravi. *Command Pattern – Design Patterns (ep 7).* `https://www.youtube.com/watch?v=9qA5kw8dcSU&ab_channel=ChristopherOkhravi`, 2017.

[8] Jacob Sorber. *How to create and join threads in C (pthreads).* `https://www.youtube.com/watch?v=uA8X5zNOGw8&list=PL9IEJIKnBJjFZxuqyJ9JqVYmuFZHr7CFM&ab_channel=JacobSorber`, 2018.

[9] Jacob Sorber. *Safety and Speed Issues with Threads. pthreads, mutex, locks.* `https://www.youtube.com/watch?v=9axu8CUvOKY&list=PL9IEJIKnBJjFZxuqyJ9JqVYmuFZHr7CFM&index=3&ab_channel=JacobSorber`, 2019.