University of Padova

Department of Mathematics

F1.js

https://github.com/filnik/F1.js

Filippo De Pretto - 1057576

Padua, Italy, 2012.

Informations about the document

Author	Filippo De Pretto
Supervisor	Tullio Vardanega

Contents

List of Figures

Abstract

In this document is described how the F1.js program has been designed and implemented. It is a project developed for the “Concurrent and distributed systems” exam, matching the requirements that can be found at: http://www.math.unipd.it/~tullio/SCD/2008/Progetto.html

The programming language chosen for this project is Javascript/node.js, for its particular properties that makes this task a lot easier than in other languages.

This is accomplished by the fact that in node.js there is no difference between concurrency and distribution, as it will be explained.

1 The language

In order to understand how we solved the problem, is important to give an explanation of how the language, its architecture and its tools are designed. Indeed, this is not a common architecture, although there are similar examples such as Twisted for Python or Event Machine for Ruby.

1.1 node.js features

As said, the project has been written using node.js. Quoting the official site's main page: *Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.*[1]

In other words, it is expressive, in particular with the Express.js framework[2], fast and scalable.

An important aspect is the use of the V8 JavaScript Engine to interpret the JavaScript code. Written by Google, it increases performance by compiling JavaScript to native machine code (x86, ARM, or MIPS CPUs)[?], before executing it, versus executing bytecode or interpreting it.

Performances aren't great only for the use of V8, but also for the programming style that node.js implies.

1.1.1 Asynchronous I/O

node.js real difference is the asynchronous I/O and evented support. Citing "cloudfoundry.com" [?]: *In order to write a fast and scalable server application, we typically end up writing it in a multi-threaded fashion. While you can build great multi-threaded apps in many languages, it usually requires a lot of expertise to build them correctly. On the other hand, these libraries (along with Chrome's V8 engine) provide a different architecture that hides the complexities of multi-threaded apps while getting the same or better benefits.*

Let's compare classic multi-threaded server with an evented, non-blocking I/O server:

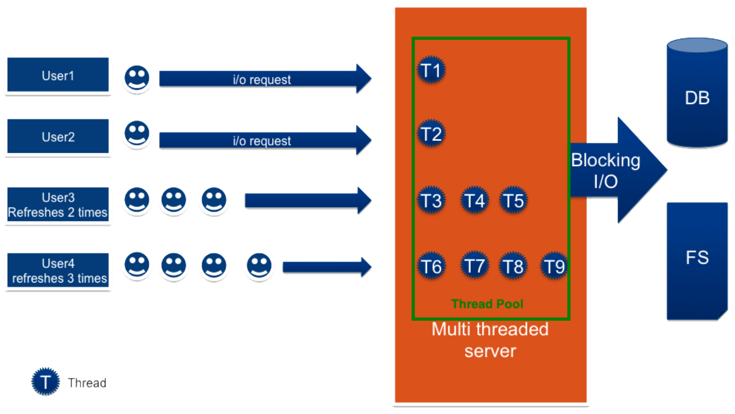

Figure 1: An example multi-threaded HTTP server using blocking I/O

[1]Official node.js website: http://www.nodejs.org
[2]Official Express.js framework website: http://expressjs.com/

The diagram in Figure 1 depicts a simplified multi-threaded server. There are four users logging into the multi-threaded server. A couple of the users are hitting refresh buttons causing it to use lot of threads. When a request comes in, one of the threads in the thread pool performs that operation, say, a blocking I/O operation. This triggers the OS to perform context switching and run other threads in the thread pool. And after some time, when the I/O is finished, the OS context switches back to the earlier thread to return the result.

Architecture Summary: *Multi-threaded servers supporting a synchronous, blocking I/O model provide a simpler way of performing I/O. But to handle a heavy load, multi-threaded servers end up using more threads because of the direct association to connections. Supporting more threads causes more memory and higher CPU usage due to more context switching among threads.*

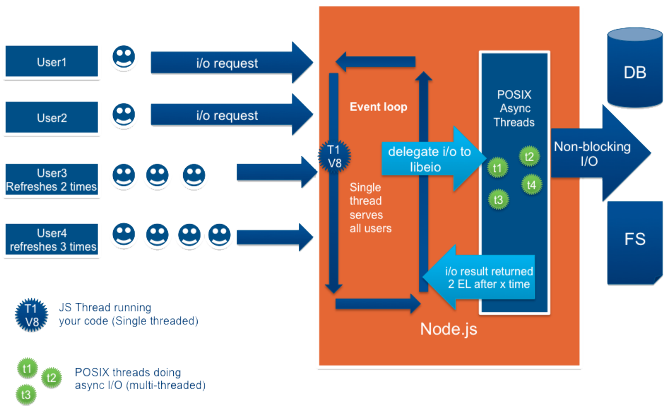

Figure 2: Event-driven, non-blocking I/O (Node.js server)

The diagram in Figure 2 depicts how Node.js server works. At a high level, Node.js server has two parts to it:

- *At the front, you have Chrome V8 engine (single threaded), event loop and other C/C++ libraries that run your JS code and listen to HTTP/TCP requests;*
- *And at the back of the server, you have libuv (includes libio) and other C/C++ libraries that provide asynchronous I/O.*

Whenever a request is made from a browser, mobile device, etc., the main thread running in the V8 engine checks if it is an I/O. if it is an I/O then it immediately delegates that to the backside (kernel level) of the server where one of the threads in the POSIX thread pool actually makes async I/O. Because the main thread is now free, it starts accepting new requests/events.

And at some point when the response comes back from a database or file system, the backend piece generates an event indicating that we have a result from I/O. And when V8 becomes free from what it is currently doing (remember it is single-threaded), it takes the result and returns it to the client.

Architecture Summary: *This architecture utilizes an event loop (main thread) at the front and performs asynchronous I/O at the kernel level. By not directly associating connections and threads, this model needs only a main event loop thread and many fewer (kernel) threads to perform I/O. Because there are fewer threads and consequently less context-switching, it uses less memory and also less CPU.*

1.2 Concurrency and distribution in node.js

We have seen why an asynchronous webserver is a lot faster than a synchronous one. The fact is that performances might not be so interesting, as in this situation, but might be more interesting the easiness of designing and implementing the solution.

As said concurrency and distribution in node.js are the exact same thing. The reality indeed is that node.js is (mainly) single-threaded as seen before, handling everything with events. For this reason there isn't the concept of "lock" in node.js or of threads. It's possible to use webworkers that are processes that you can create forking the single-thread but are heavy and usually not needed, like in this case. They are needed only in cases of CPU-intensive tasks that might block the main thread from answering the events that it gets.

So, when you have the application that waits for an event, it's of no importance to know if the event is generated in the client, in the server or in another server. The important thing is to be subscribed to the right event emitter, and this can be easily done and changed, even at run-time.

In this way, we can run the program in a single instance on a server, in multiple processes in a single server or even split the processes in different servers without affecting the logic of the application at all. This means that we can partition vertically every single event emitter, if we want. This is a lot more scalable than rewriting the whole application or to adapt it in both cases.

1.3 node.js versus Twisted and Event Machine

Basically the great advantage of node.js is the use of the JavaScript language. The reason is that all the libraries already written for the JavaScript language are asynchronous, since it's how JavaScript has been designed, while Python and Ruby have a lot of synchronous libraries that you cannot use inside these two asynchronous environments.

Another important fact is that now JavaScript is an isomorphic language.*By isomorphic we mean that any given line of code (with notable exceptions) can execute both on the client and the server.*[?]

This seems trivial but it's not. Indeed we can easily communicate between client and server in a single language using events and, if needed, RPC. We can indeed write the same library for the client and the server and validate the data in each step to prevent code changes. We have no mind-switch from one language to another and the application can avoid to decouple in a strictly way view from controller from model, since the client-server limit is not so strict as in other contexts.

1.4 socket.io, ascoltatori and redis

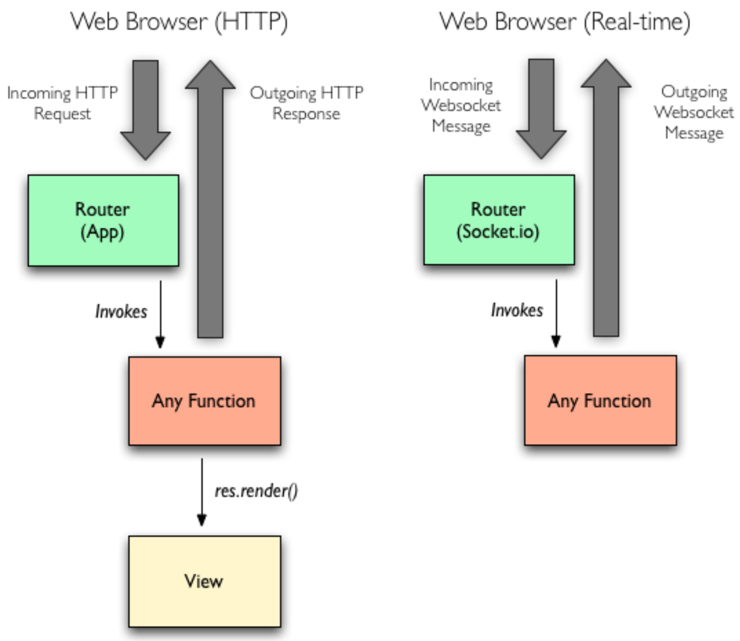

Figure 3: Express and socket.io model

In Figure 3 you can see how the framework used and socket.io work. They are really simple and neat. Express has only the job to answer the web requests and render the HTML pages. Socket.io is a wrapper over the WebSocket technology.

Socket.io solves a lot of problems that occurs if you use the WebSocket technology as it is. What socket.io handles is:

- the connection. WebSocket is a modern technology that is not available in old browsers. This library overcomes this problem using different technologies using the fastest possible. In this way it degradates gracefully from WebSocket to Flash or even the AJAX long-polling technology;
- the re-connections without creating different instances but reusing the same instance that there were before the disconnection;
- the send of JSON objects instead of plain text;
- the namespaces. In this way you can have a single WebSocket connection but using different namespaces to divide different parts of the logic.

In this way, we are totally unaware of the underlining problems of the network and we can handle the problem in a lot easier way.

It allows you also to pass pointers to functions that are executed in the context of definition. In this way you can simulate a Remote Procedure Call, if needed, without loosing transparency at all since it's all asynchronous. In this way, you don't need to handle problems regarding the proxy/skeleton pattern.

Ascoltatori is another really useful wrapper library. It is really interesting since it makes a lot easier and transparent the communication server to server. Indeed it wraps a lot of ways to communicate, in particular: Redis, AMPQ (RabbitMQ), ZeroMQ, MQTT (Mosquitto) or just plain node.js.

In this way, we can have for example a node of the network written in C++ with ZeroMQ and use this library to make everything work without changing a line of code.

We choose **Redis** because it "is a database, but it would be more accurately described as a datastructure server" [**?**]. In this way we can use only one technology both for communication and storage. Instead of setup a Redis instance, we choose to use redistogo, that is an online storage service (it would work in local too, anyway).

2 The problem

To analyse the problem we have chosen to describe it through UML-schemas in order to have a clear path to follow.

2.1 F1.js

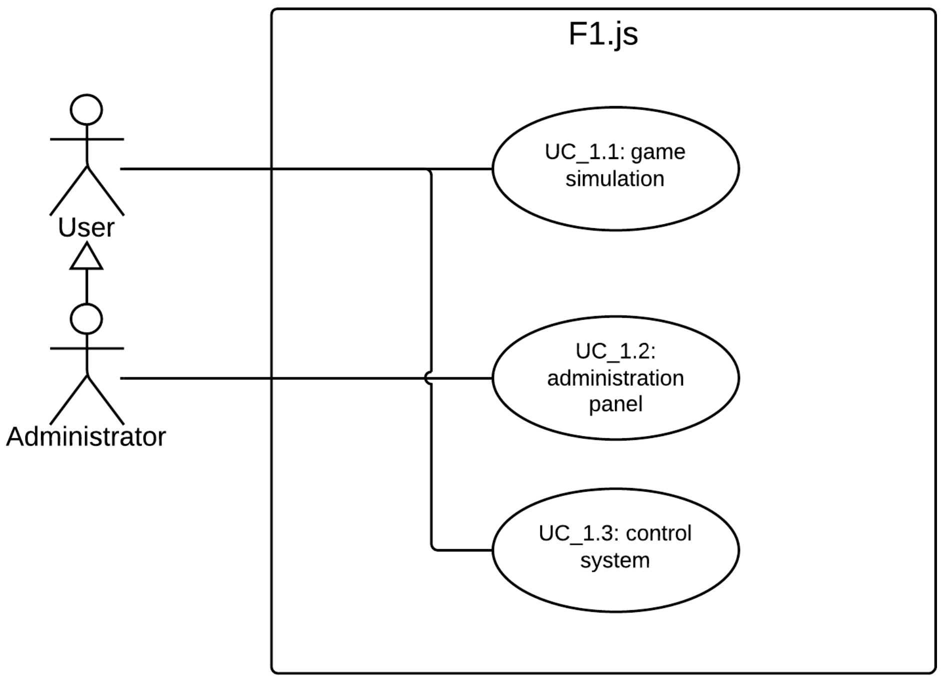

Figure 4: F1.js - the system

In Figure 4 is described the problem divided in its 3 main parts: the game simulation, the administration panel and the control system.

The normal user can enter the game simulation and view the control system with the status of the simulation. Only the administrator can change some of the game settings.

2.2 The Game Simulation

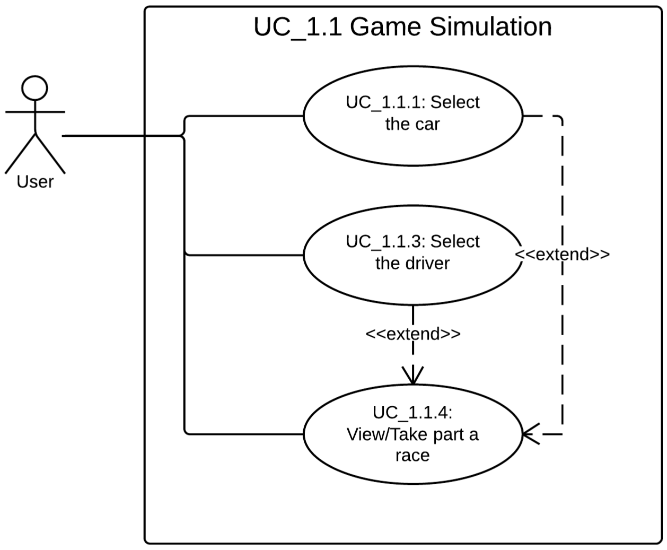

Figure 5: The Game Simulation

When a new user connects to the system, she can decide what car and which player to play with. If she doesn't want, she can let the default parameters. Finally, she can decide to view or to take part to the current race. If there are already enough player, the race begins and every new player can only watch the competition.

2.3 The Administration Panel

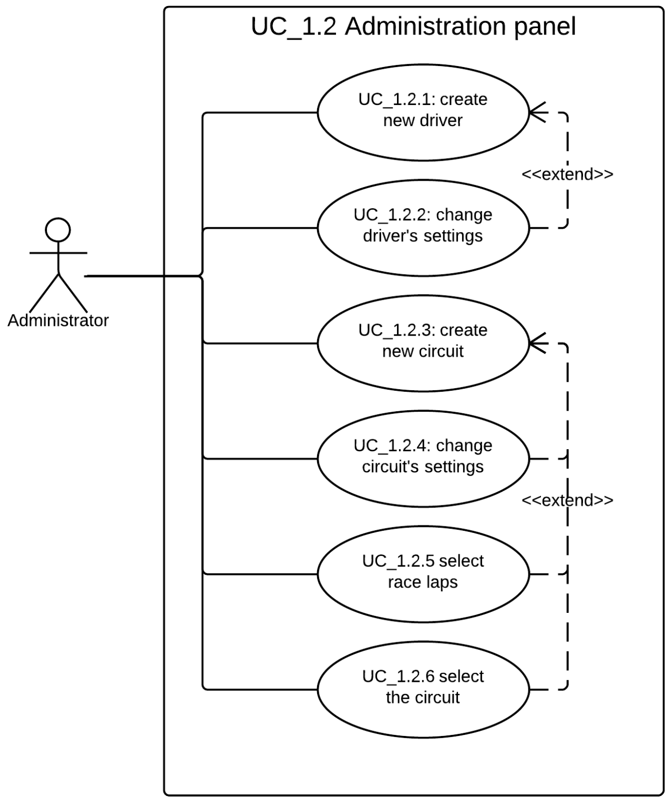
UC_1.2 Administration panel
Administrator
UC_1.2.1: create new driver
UC_1.2.2: change driver's settings
<<extend>>
UC_1.2.3: create new circuit
UC_1.2.4: change circuit's settings
<<extend>>
UC_1.2.5 select race laps
UC_1.2.6 select the circuit

Figure 6: The Administration Panel

In Figure 6 there is the panel where the administrator can change some parameters of the simulation. The administrator can:

- create a new driver or change the parameters of an old one;
- create a new circuit or change the parameters of a circuit already inserted;
- select the race laps for the next race;
- select the circuit of the next race.

2.4 The Control System

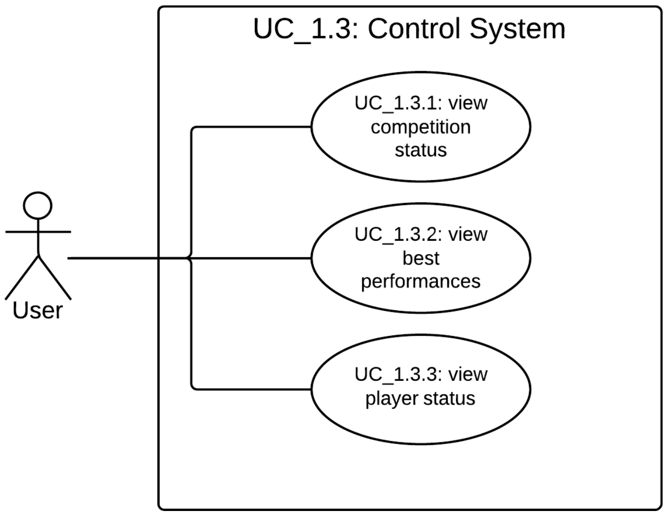
UC_1.3: Control System
UC_1.3.1: view competition status
UC_1.3.2: view best performances
UC_1.3.3: view player status
User

Figure 7: The Control System

In Figure 7 is described the control system. Every user can view:

- the competition's status;
- the best performance until now;
- the player's status.

3 System Analysis

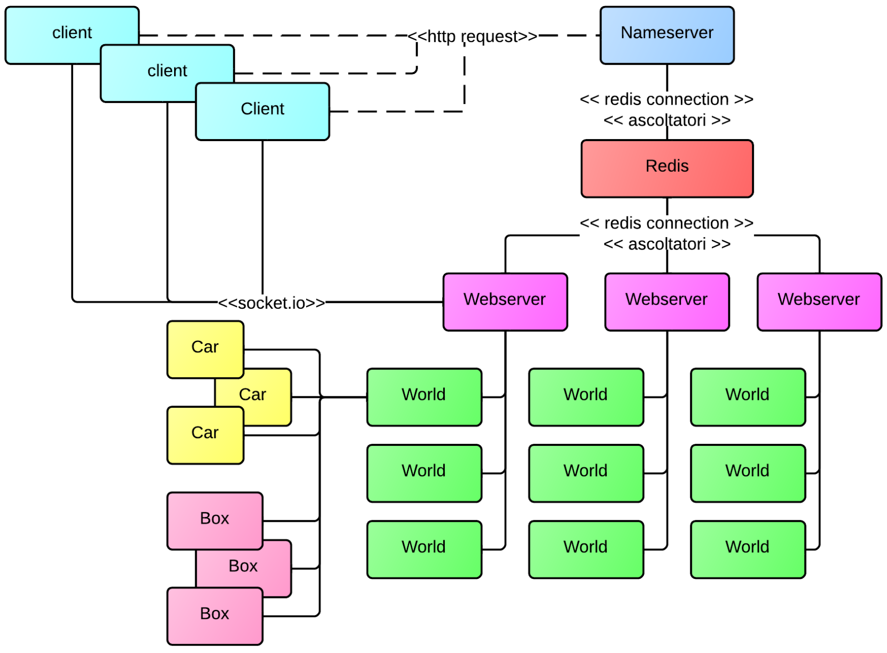

Figure 8: Main system's entities

In Figure 8 is described the main system architecture proposed. There are some clients that connects through a web browser to the website and one, or more, servers that handle the requests.

3.1 Client

In our system, the clients are monitors and can do the action described in the use cases in Section 2. In the solution proposed they does nothing, they are only interfaces. They can indeed not communicate directly with other clients, due to the limitations of the JavaScript language.

However, since the language is isomorphic, a whole simulation can be moved to one client and run there with all the parameters set previously. Obviously, that client will be the only monitor if there is not a server that works as a bridge for all the other clients. This situation is shown in Figure 9, where a client simulates the whole system and returns the results to the other client.

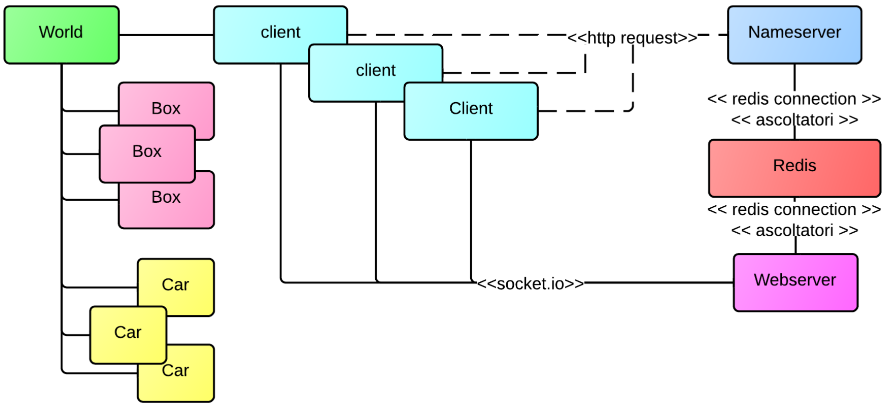

Figure 9: Alternative vision of the system client-focused

This solution, although may seem “strange” at first sight, is similar to the server-client architecture designed in some multiplayer games where one client acts as a server too. However, since the security issues and the increased overhead caused by the client-server-client communication, this choice hasn't been chosen. Although it would be really easy to switch, since you have only to move the code and redirect the calls.

3.2 Nameserver

In Figure 8 you can see the we have chosen to have a nameserver too. The reason is because it servers as a name solver and a balancer too. We need to have an active figure that update the redis database with the entities that are connected to the system.

In this way, we can redirect every client call to the less loaded webserver and to let it join to the selected simulation .

Having only the redis instance operative, would lead to ghosts every time an entity disconnects itself, while in this situation the nameserver will keep the situation monitored.

3.3 Redis

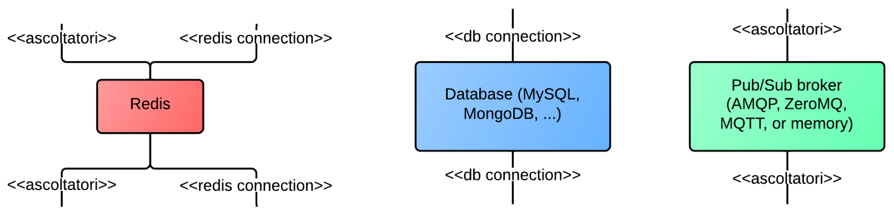

Figure 10: Redis VS other solutions

For the reasons explained above Redis comes very handy for this use case. It serves indeed as a database and as a pub/sub broker too, that allows our server-to-server communications. The current solution is show in the left part of Figure 10. However, if we want, we can split the Redis entity in two (database+pub/sub broker) and handle the two jobs in a separate way, as you can see in the right part of Figure 10.

3.4 Webserver

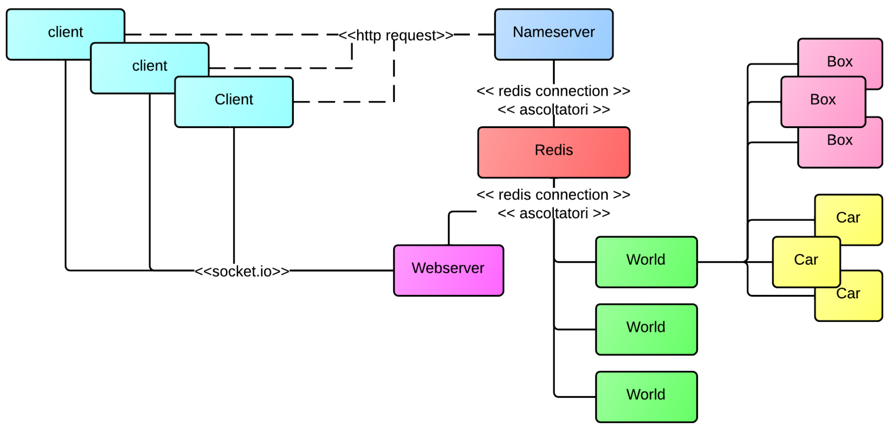

Figure 11: Simulation separated from the Webserver

In the architecture chosen as default (shown in Figure 8) there is only Client, Webserver, Redis and the Nameserver as active entities. This is because of the fact that there is no need of a further grade of distribution. As we know, more distribution will lead to possible lags, fragilities and problems. The system is also horizontally distributed, so if a webserver dies, the nameserver can redirect all the simulations to another one.

However, World, Car and Box can be distributed if needed, as you can see in Figure 11 (the other webservers are omitted in the Figure). The reason is pretty basic. Since they are event emitter, there is no need for them to be local, but they can be set everywhere they can be reached. Obviously the simulation will start with a bigger delay, there can be problems such a fault of the communication (or a disconnection of a box, for example), but you can choose this solution if you want it.

The webservers are not connected to each other since there is no need of communication. Every simulation is indeed atomic. We could however connect every webserver if we want to add the ability (not required) to move a live simulation from a webserver to another one. This could be easily done, since the whole simulation is a JSON object and the IA logic is shared in the webservers. So we would only have to add the ability for the clients to change the webserver with whom it is connected, that is pretty basic (simply send the address to the client and create a new socket.io connection, disconnecting from the previous one).

This can be easily done settings a flag, since the communication instead of being in the local memory, it occurs through Ascoltatori.

3.5 World, Car, Box

The World represents the circuit, the boxes, the cars and the players. As we have said, the world could be both on the server (Figure 8) or on the client (Figure 9). It can be also distributed as a separate process on the same server or in another server at all. Since every parts of it are connected through Ascoltatori, every parts of it can be distributed easily to other servers.

We can decide to distribute also some part of the World in one or in every client. For example, we can choose to move every car simulation to its client, the same for the boxes. This however, will increase the fragility of the system, the security, the lags of the system and the communication overhead between the clients and the server for syncing the entities.

For this reason, the proposed solution has every entity in the server, although as said this configuration can be easily changed.

4 Loading and Monitor

Every simulation needs a loading time. In this time, the simulator (that in the default situation is the server) needs to retrieve the data, start the simulation and then send the output to every monitor that has then the duty to reproduce it.

According to the configuration selected, the time needed may vary and may occurs problems, if some nodes go down (in particular if we choose to distribute the cars too or using a client as host).

This process can be done in real-time too, however this will lead to lags or errors in the simulation, reducing its quality. For this reason, since the user's input is not provided, we have decided to proceed in this way.

5 Artificial Intelligence

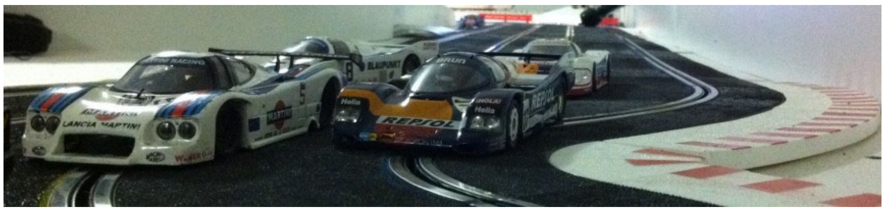

Figure 12: A slot car overtaking

In Figure 12 we can see a step car race. Slot cars are machines that runes in a circuit, like the one that we have to simulate, that can run only in a defined slot with the circuit itself. It is interesting to notate that, to overtake an opponent, the two cars have to be in different slots, otherwise they will crash.

In some competitions, every car is in a different slot, so the overtake is just natural when a car runs faster than the other. However, since this will make the simulation not so effective, we have decided to simulate the race with a different type of track.

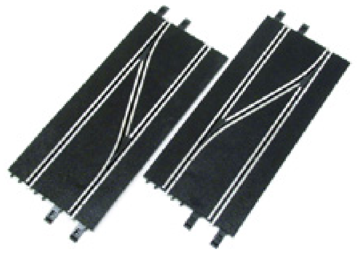

Figure 13: A switcher track

In Figure 13 we can see a *switcher*. With this term we mean a particular track that allows machines to change their slot. In this way, cars can't simply change slot when they want, but they have to wait for this sort of *interrupt*.

As for embedded real-time systems, this will lead to a big simplification of the logic and of the connections. Indeed, we haven't to check in every instance where the other cars are (that may be a costs in terms of lags, if the cars are distributed) but only when we receive this sort of interrupt. This is also much more interesting, since it is easier that crashes occurs and the cars has to handle it correctly.

To not complicate things too much, every switcher allows every car to move to every narrow slot (i.e. if the car is in slot 1, it can move to slot 2 but not to slot 3 since it is too far).

5.1 Events to handle

Every car has its parameters and doesn't care about where the other cars are until it is close to:

- a switcher. In this case, the car has to decide whether to change slot to overtake an opponent or not;
- another car. In this case, the car has to decide its speed according to the car that is reaching, trying to change slot as fast as possible to overtake it;
- its box. In this case the car has to decide if it's the case to enter the box or not;
- two or more of these cases.

In this way, the problem reduces to few decision and communication between the cars and the other part of the world. There is the need of a check only in these situation and not in every instance.

A Installation

It is really basic to install the program. The first thing to do, is to download the source code. It can be easily downloaded from the F1.js website[3] as .zip package or using git.

Once the code is downloaded, you have to install node.js and npm (that usually comes with node.js). We relate to the official site for further details.[4]

The first thing to do, is to setup the redis database. It can be locally available or on a redistogo instance. Once you have chosen what to use, you have to create the "settings.js" file with your settings.

After that, you have to open the shell in the source folder and type "npm install". This would install every node package needed. To run the server, just type "node app.js" and go to "localhost:3000" with your browser to see the program running. You can open multiple tabs on the same site to simulate multiple clients.

[3]F1.js website: https://github.com/filnik/F1.js
[4]node.js official site: http://nodejs.org/