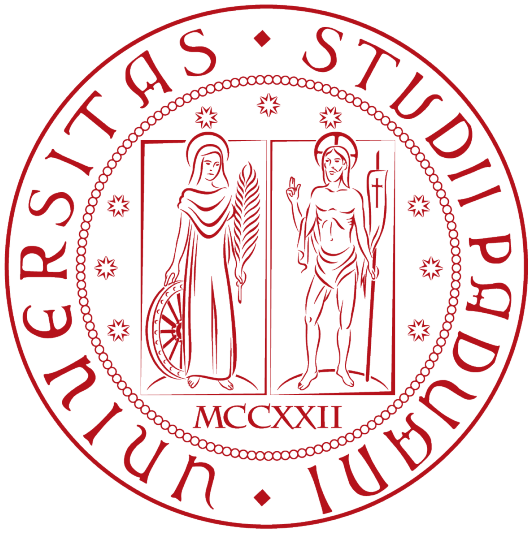

University of Padova

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)[3], before executing it, versus executing bytecode or interpreting it.

As you can see in Figure 1, this raises a lot the performances, reaching almost Java's speed.

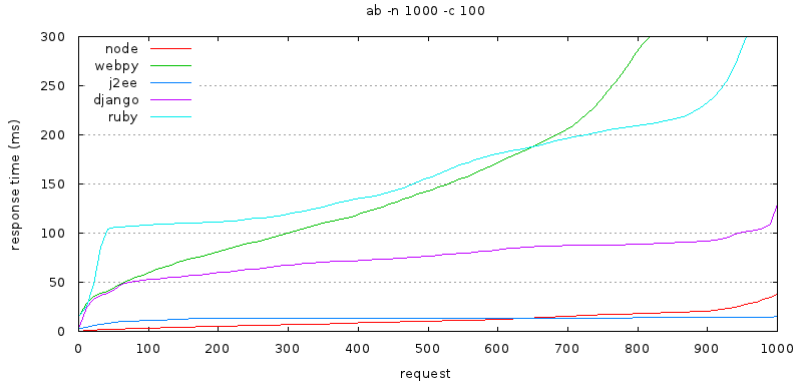

Figure 1: node.js benchmark versus other popular languages/platforms/frameworks

However, the performance 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" [1]: *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*

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

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:

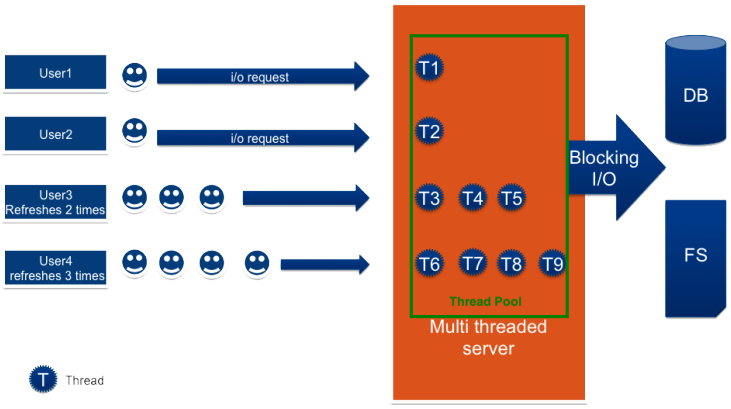

Figure 2: An example multi-threaded HTTP server using blocking I/O

The diagram in Figure 2 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.*

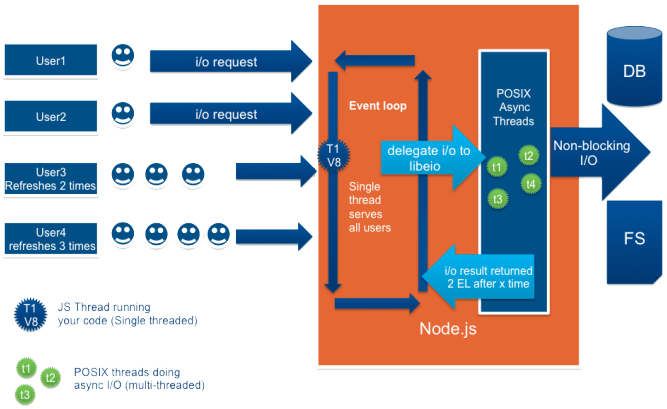

Figure 3: Event-driven, non-blocking I/O (Node.js server)

The diagram in Figure 3 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.*[2]

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

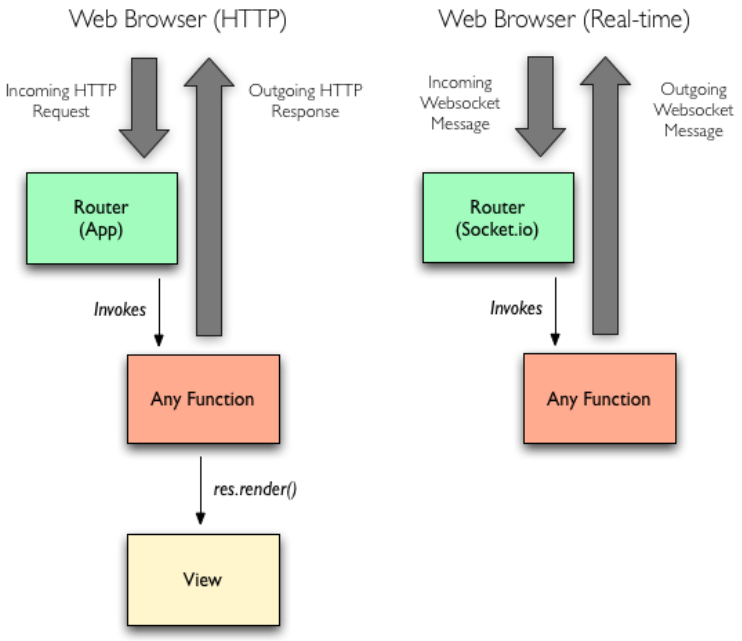

Figure 4: Express and socket.io model

In Figure 4 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" [4]. 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

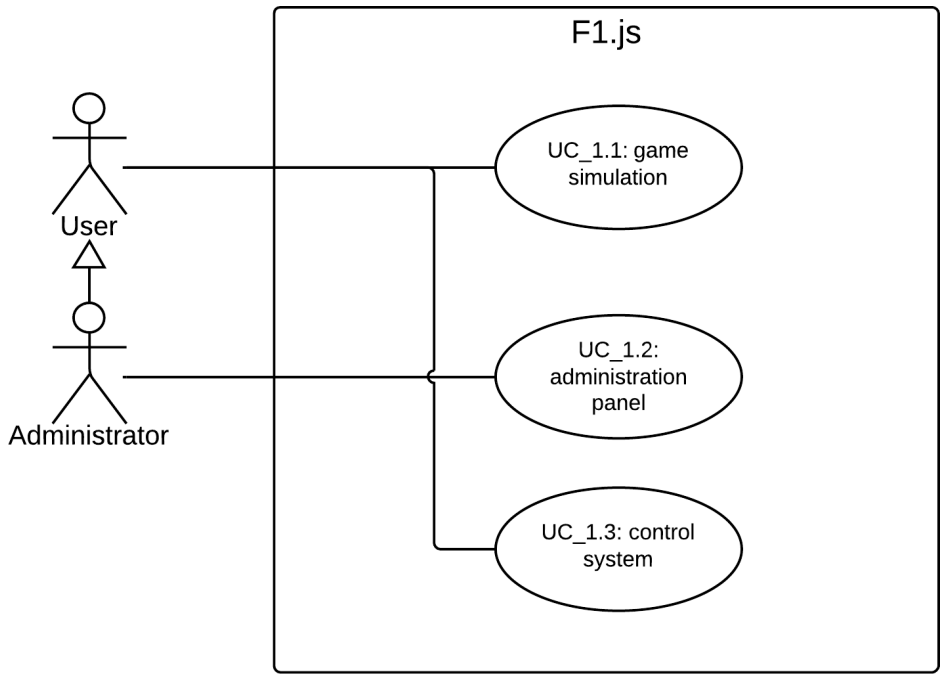

Figure 5: F1.js - the system

In Figure 5 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

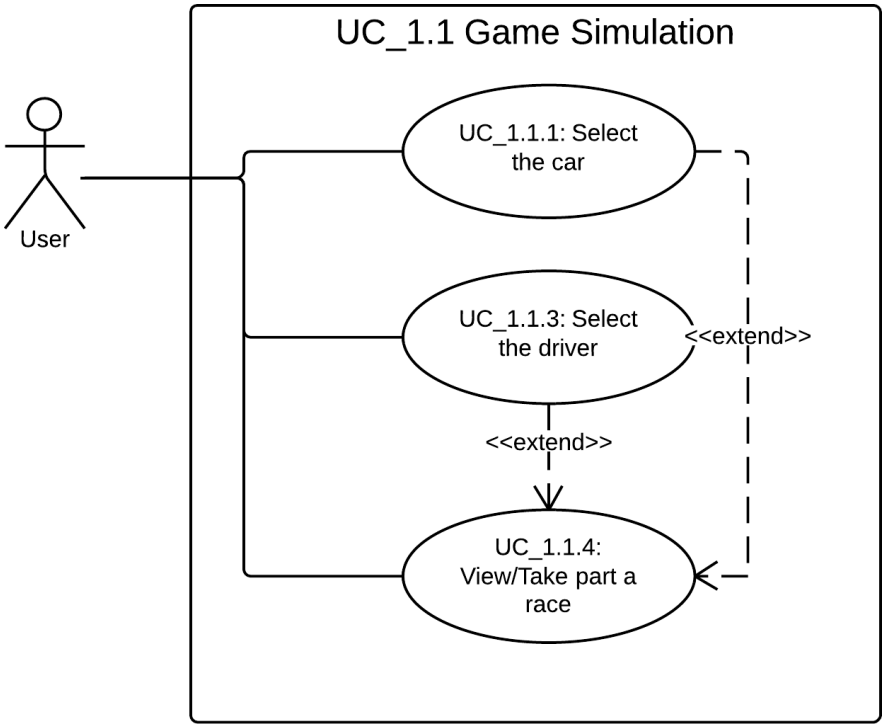

Figure 6: The Game Simulation

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

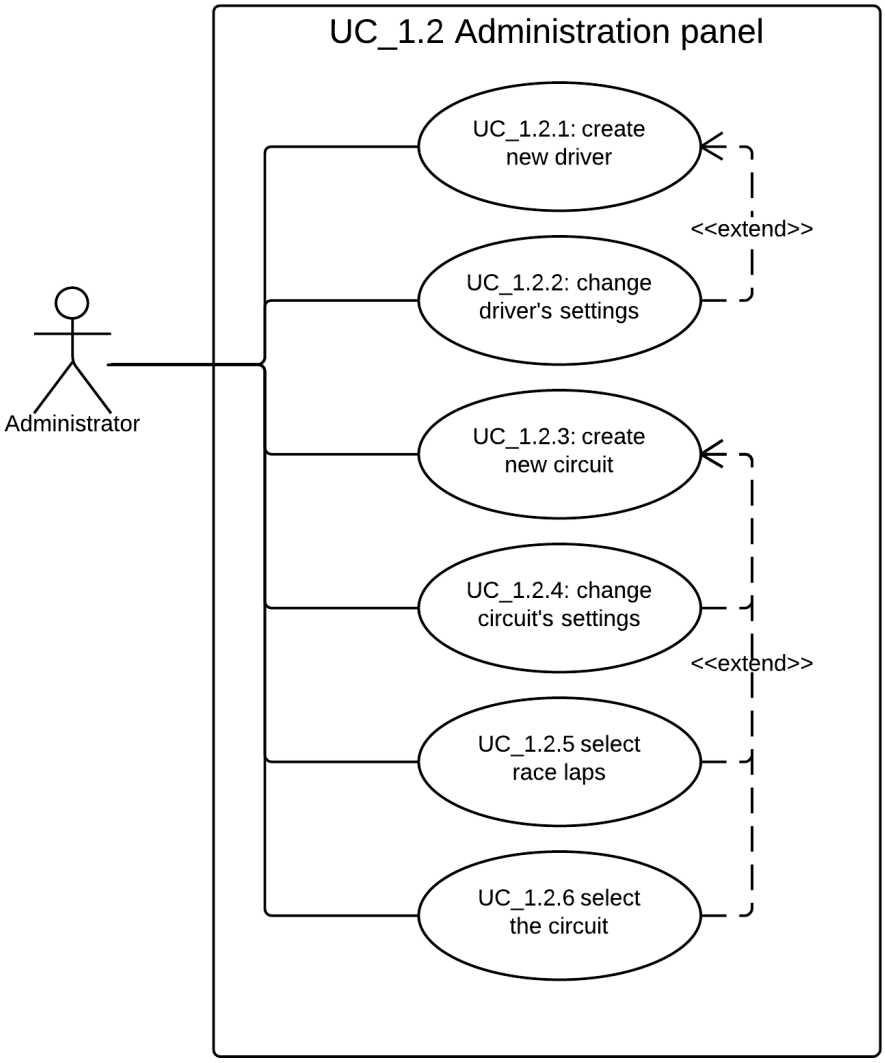

Figure 7: The Administration Panel

In Figure 7 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

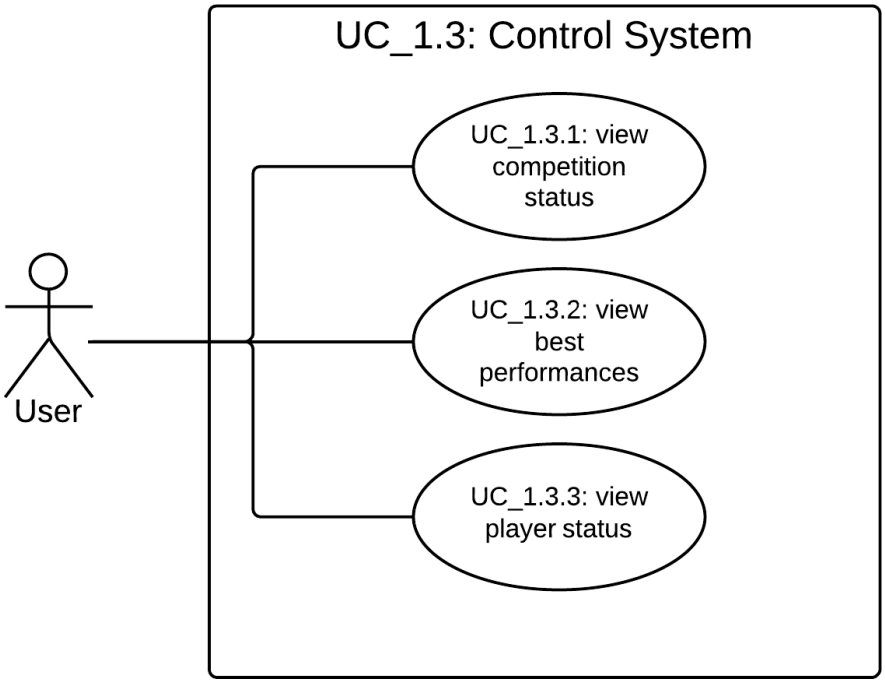

Figure 8: The Control System

In Figure 8 is described the control system. Every user can view:

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

References

[1] Future-proofing your apps: Cloud foundry and node.js. http://blog.cloudfoundry.com/tag/polyglot/.

[2] Scaling isomorphic javascript code. http://blog.nodejitsu.com/scaling-isomorphic-javascript-code.

[3] V8 introduction page by google. https://developers.google.com/v8/intro.

[4] G. Rauch. *Smashing Node.js: JavaScript Everywhere*. Smashing Magazine Book Series. Wiley, 2012.