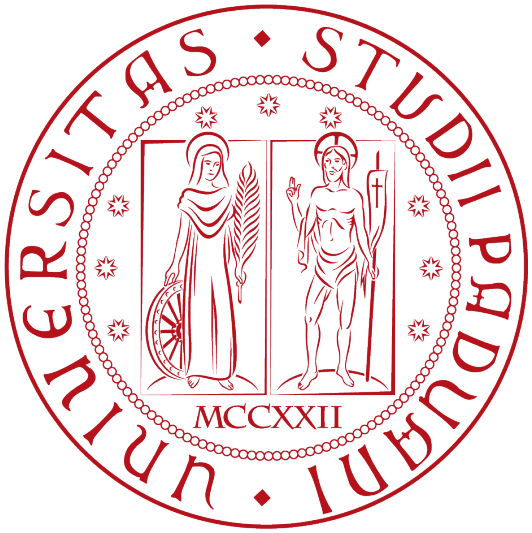

University of Padova

Department of Mathematics

F1.js

Formula 1 in a concurrent yet distributed way

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 it's described how the F1.js program has been designed and implemented. The programming language chosen for this project is node.js/Javascript, for its particular properties that makes this task a lot easier than in other languages.

With this new technology is possible to do something that requires a great amount of work in an easy way. This could be done also 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 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.

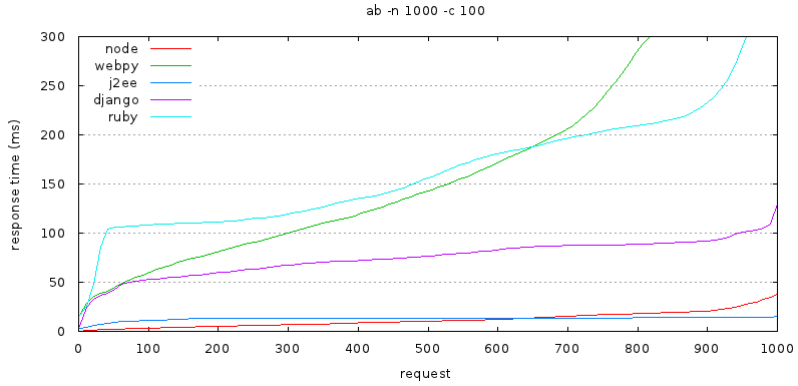

Figure 1: node.js benchmark versus other popular languages/platforms/frameworks

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:

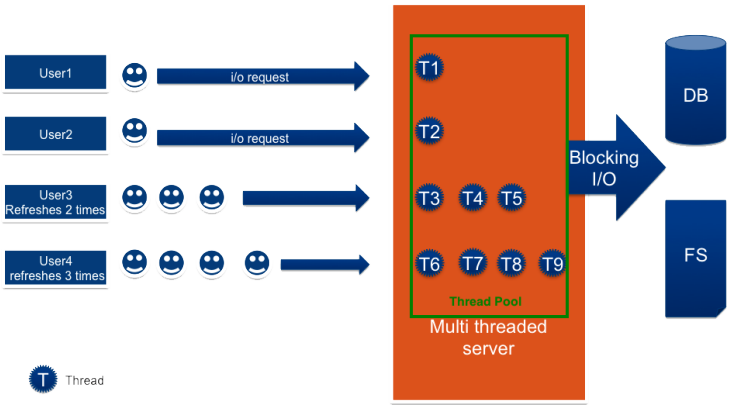

Figure 2: An example multi-threaded HTTP server using blocking I/O

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.*

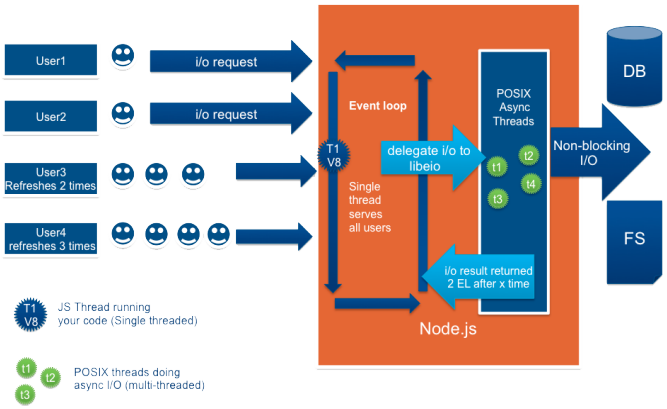

Figure 3: Event-driven, non-blocking I/O (Node.js server)

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 Synchronous versus Asynchronous

We have seen why an asynchronous webserver is a lot faster than a synchronous one. The fact is that performances might not be interesting, such 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.

1.3 node.js versus Twisted and Event Machine

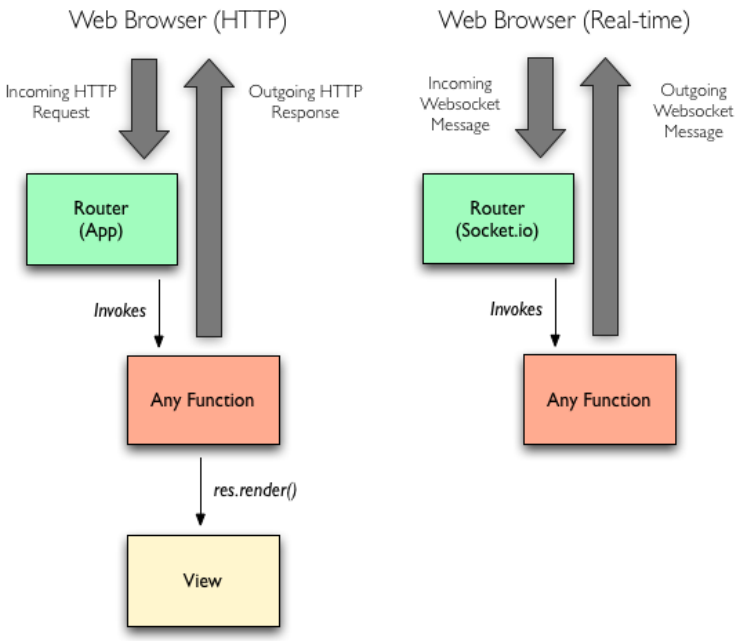

Figure 4: Express and socket.io model

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 always be, while Python and Ruby has 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 not RMI or 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 Database

MongoDB (from "humongous") is a scalable, high-performance, open source NoSQL database written in C++.

The particularity of MongoDB is that it has a document-oriented storage, in particular it stores BSON documents that are the binary representation of JSON documents.

JSON, acronym for JavaScript Object Notation, has been designed to be as much similar as possible to JavaScript objects. It is now a standard and it is commonly and widely used. So much, that Facebook uses JSON too. in Figure 5 you can see how everything in the structure designed simply used JSON Objects. They are managed, changed, stored and got without any conversion from XML, to class-objects, to SQL data or anything similar.

This has a big impact in performace, since MongoDB was designed to be as fast as possible, and to stability, since without any conversion, it is much easier not to have some error in the process.

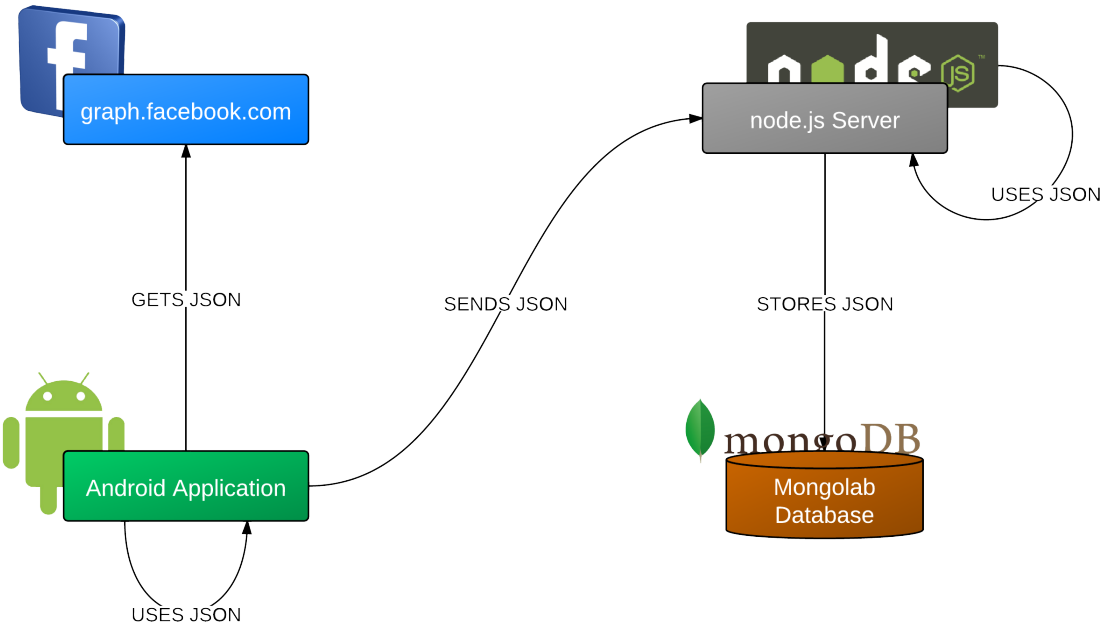

Figure 5: How the applications interact with each others, all in JSON

1.5 Difficulties with MongoDB & Mongoose

I have also dedicated some hours at the study of MongoDB and how to create correctly a database of this type. I was lucky to find the page of the last NoSQL day[3] with all the speeches given that day. I had a basic knowledge of MongoDB's API but I had no idea of how to model it.

These videos have helped me to create an efficient solution in a quicly way. Solution then I formalized also with the use of Mongoose.

Initially, we used the native driver for MongoDB, not knowing the existence of Mongoose. This was easy to use, but it didn't allows to check if data given are correctly modelled or not. With Mongoose you have this check and you can also set the "strict" mode, in order not to insert objects that match the required fields but have also something more than expected.

Furthermore, with MongoDB, the basic rules of E-R modeling, are completely reversed. To cite some meaning considerations about it:

Data duplication and denormalization are first-class citizens.[**?**]

This means that, instead of what happends to E-R databases, the focus is not to avoid duplication or denormalization. So, they are allowed and, in some cases, needed and encouraged. There are some cases where E-R databases aren't the best choice. This is because the E-R system is user and answers oriented, as Ilya Katsov says [**?**]:

- *The end user is often interested in aggregated reporting information, not in separate data items, and SQL pays a lot of attention to this aspect;*
- *No one can expect human users to explicitly control concurrency, integrity, consistency, or data type validity. That's why SQL pays a lot of attention to transactional guaranties, schemas, and referential integrity.*

[...]

NoSQL data modeling often starts from the application-specific queries as opposed to relational modeling:

- *Relational modeling is typically driven by the structure of available data. The main design theme is* ***"What answers do I have?"***
- *NoSQL data modeling is typically driven by application-specific access patterns, i.e. the types of queries to be supported. The main design theme is* ***"What questions do I have?"***

So, as you can see, the focus is not to create a database easily consulted by an human being, but to create a model that can scale easily answering as fast as possible to the questions that the program needs. Although the database structure stays clear, updates and transaction should not be done by hand. For example, in this project the login for a player and an uploader is duplicated. This is because every time you need one of them, you shouldn't download two tables but only one.

As Gabriele Lana says [**?**]:

The best design is the one where needed data can be easily extracted

[3] NoSQL day website: http://nosqlday.it/

The way you need to query your data should influence your design

In Figure you can see how you can tune your queries with MongoDB. With this, you can ask the engine to explain how it processes the query, if indexes are being used and how much time is needed to answer the request. You can also set the engine so to log every request that is slower than a given time, to find the queries bad formed, fix them and speed up your application.

2 The problem

3 Subproblems analysis

4 Solution: the architecture

5 How the solution resolves the problem

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.