Corso di Sistemi Distribuiti, A.A. 2012/13
Anderloni Francesco - M. 1034160
Ialenti Andrea - M. 1056894

A platform for easily building fast, scalable network applications

1 Introduction to Node.js

1.1 Overview & History

Node.js is a platform developed for easily building fast, scalable network applications. It 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]. Node.js was created by **Ryan Dahl** in 2009 as a set of asynchronous libraries taking advantage of the *V8* JavaScript Engine (developed and open sourced by Google and powerful component of their *Chrome* web browser), which translate JavaScript code to native, optimized machine code [2], as well as a custom HTTP parser and the *libev*, *libeio*, *evcom* and *udns* libraries for asynchronous I/O and event loop management [3] – later replaced by the *libUV* astraction layer[4]. Before its inception, few other asynchronous libraries and frameworks existed – notable examples include the **Twisted** framework for Pyhton (2002) and **EventMachine** for Ruby (2003) [5] – and the use of JavaScript as a server-side programming language was well known since the late '90s. However, up until that point nobody even thought about a combination of the two. Dahl believed there was some potential in the development of such thing and focused his energy on the project, finally managing to showcase his progresses at *JSConf* 2009 in Berlin [6]. From there on, Node.js gained the support of thousands of independent developers, and at the end of 2010 it also obtained the financial support of the software and services company **Joyent** [7].
Node.js is currently distributed as an installer package available for download on the official website [1]. Once installed, any Node application can be launched through the use of the `node` command; the installer also make the access to the **Node Packaged Modules** [8] (third-party-developed modules that expand Node's capabilities) directory through the `npm` command.

1.2 Main features

- The main feature of node.js is its **event loop**. The event loop make **asynchronous coding** possible without requiring overhead-heavy solutions like multithreading (*Apache*'s way of dealing with concurrent requests, for example). Using **callbacks** the developer can stop worrying about what happens in the backend, and he/she is guaranteed that his/her code is never interrupted and that doing I/O will not block other requests without having to incur the costs of thread/process per request (i.e. memory overhead) [9]. The event loop handles and processes external events and converts them into callback invocations – I/O calls are the points at which Node.js can switch from one request to another. At an I/O call, the code saves the callback and returns control to the Node.js runtime environment, which proceed with the execution of the code. The callback will be executed later when the data actually is available, and the results will be handled using powerful constructs such as **first-class functions** and **anonymous functions** [9]. Figure 1 shows how the event loop works.
  The event loop is a **single thread** that handles multiple concurrent connections, which makes the overhead of Node.js grow relatively slowly as the number of requests it has to serve increases because there's no OS thread/process initialization overhead. All long-running tasks (network I/O, data access, etc…), instead, are always executed asynchronously on top of worker threads which return the results via callback to the event loop thread. JavaScript's language features (functions as objects, closures, etc…) and Node's programming model make this type of asynchronous/concurrent programming much easier to utilize – there's no thread management, no synchronization mechanisms, and no message-passing nonsense [11]. The single-thread nature of the event loop makes the execution of programs virtually **deadlock-free** [12].

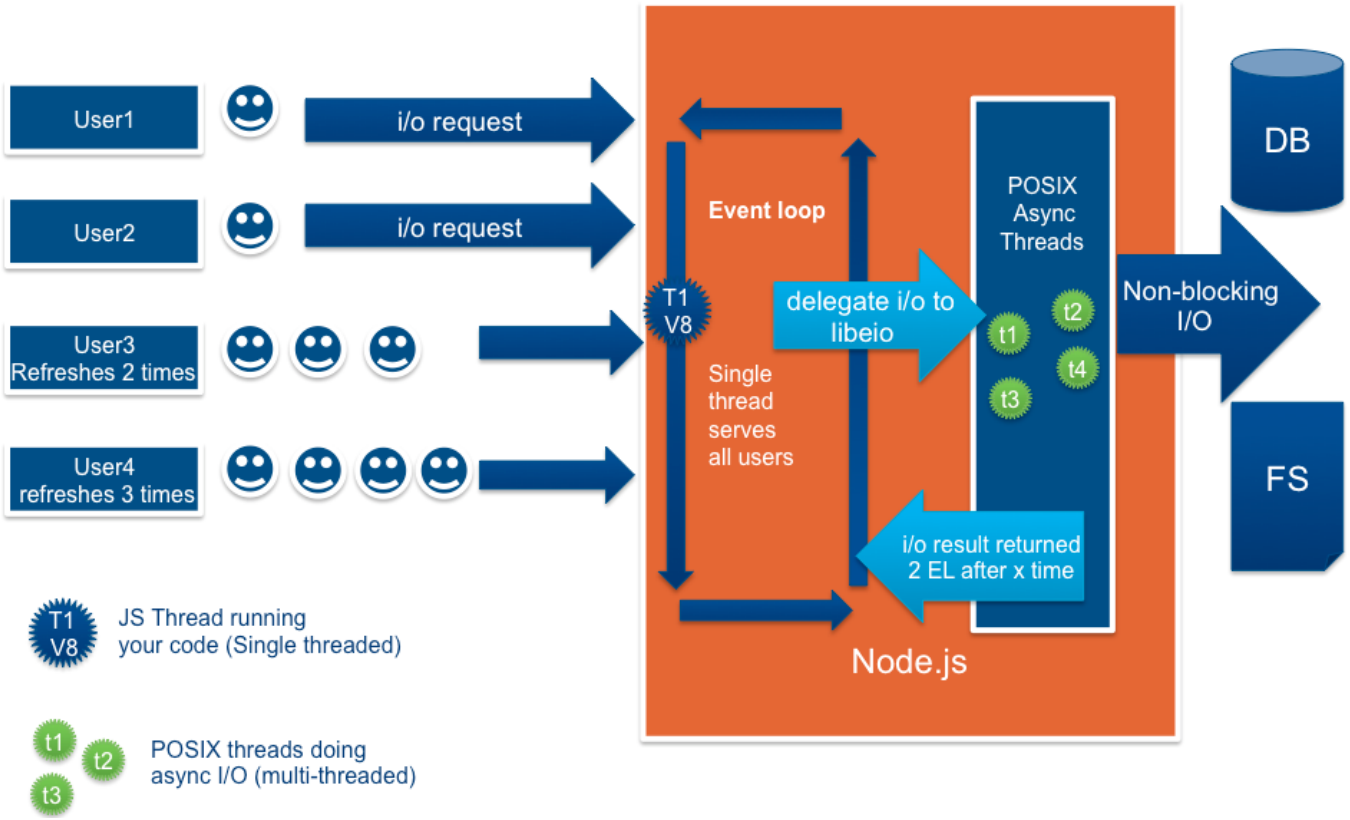

Figure 1: The Event Loop in Node.js – Source: [10]

- Node.js is a server-side software system designed for writing scalable Internet applications, notably web servers. Programs are written on the server side in JavaScript. Node.js **creates a web server by itself**, making it unnecessary to use web server software such as *Apache* or *Lighttpd* and allowing full control of how the web server actually works [4].
- Node.js is extremely **developer-friendly**. First of all, it enables web developers to create an entire web application on both server-side and client-side in **one language**, JavaScript, which is easy to learn and largely present on the Internet. Node is simple to install and configure and it just work "*out of the box*" on a variety of operative systems. It has a built-in package manager called **npm** which lets users install additional modules with a single command-line instruction. **Modules** are additional libraries which introduce support for new functionalities such as database connection, encryption and view templating. Modules are the result of the hard work of the ever-growing **developers community** surrounding node, which since its launch has collaborated in refining and expanding its capabilities [13, 14].

1.3 Pros & cons of using Node.js, and typical use cases

Nowadays Node.js is being used in a lot of application all over the internet, meeting the needs and requirements of a vast number of web application. It is not, however, the "*Holy Grail*" of webapp development: there are some use cases for which node just isn't cut for and won't bring any advantage – sometimes even producing a performance loss.
Applications that are very **heavy on CPU usage**, and very light on actual I/O, such as video encoding software, artificial intelligence or similar CPU hungry software, can be realized with node, but you'll probably get better results with C or C++ [15]. If you still need to do CPU-heavy computation and interfacing with the web a good solution lies in realizing your web services with node, coding the heavy part in C/C++ and then using one of the many C/C++ interfaces for node to realize the communication between the two ends.
Node will not bring any more benefits than PHP, Ruby or Python for **CRUD** ("Create, Read, Upload, Delete") or **simple HTML** applications. You will most likely end up sacrificing the power of tested and reliable frameworks for a little scalability (although there is an increasing number of interesting frameworks for quick and efficient webapp creation being developed for node lately) [15]. That said, there are some use cases in

which Node.js is actually recommended for the advantages it brings to the table: all of them pretty much rely on of the fact that a node web server can satisfy a great number of requests thanks to asynchronous coding. An example of this would be implementing **lightweight REST/JSON APIs**: node's non-blocking I/O model combined with JavaScript actually make it a great choice for wrapping other data sources such as databases or web services and exposing them via a JSON interface [15].
Node is also great for **soft real-time applications** such as chats/instant messaging networks, Twitter-like feed apps and sports betting interfaces, as well as **AJAX/websocket single page apps** (a good example of this is Google's *GMail*). The ability to process many requests per seconds with low response times, combined with features like sharing components such as validation code between the client and server make it a great choice for modern web applications that do lots of processing on the client [15].
Finally, **data streaming** is another good use case: in fact, traditional web stacks often treat http requests and responses as atomic events, but they actually are streams, and many cool Node.js applications can be built to take advantage of this fact. One great example is parsing file uploads in real time, or building proxies between different data layers. This was actually the use case which inspired the creation of node – in the beginning, Dahl wanted to find the best way of notifying a user, in real time, about the status of a file upload over the web [6].

1.4 Code examples

Here is the most simple example of a Node.js application - the classic *Hello World* program, with a little modification (the insertion of a `setTimeout` call) to show how asynchronous coding works.

```
//server.js
var http = require("http");
http.createServer(function (req,res) {
  res.writeHead(200, {"Content-Type": "text/plain"});
  setTimeout(function() {
    res.end("World\r\n");
  }, 2000);
  res.write("Hello\r\n");
}).listen(8000);
```

Starting the server is as easy as launching a terminal command:

```
$ node server.js
```

In another terminal window we launch `curl` (a command line tool for transferring data with URL syntax [16]) and get this:

```
$ curl localhost:8000
Hello
```

And two seconds later:

```
$ curl localhost:8000
Hello
World
$
```

showcasing how the `setTimeout` instruction doesn't block the execution of the program, which continues with the writing of Hello before outputting World and finally terminate the communication.
This works for **multiple concurrent requests** too (the following example was deployed on *Cloud9*, an online IDE with basic application hosting capability [17] – so in this case there are actual communication delays too):

```
$ curl node-hello-world.ffander.c9.io && echo "done1"
  & curl node-hello-world.ffander.c9.io && echo "done2"
  & curl node-hello-world.ffander.c9.io && echo "done3"
  & curl node-hello-world.ffander.c9.io && echo "done4"
  & curl node-hello-world.ffander.c9.io && echo "done5"

Hello
Hello
Hello
Hello
Hello
World
done5
World
World
World
done2
done1
done4
World
done3
```

Let's now analyze an example of how **asynchronous I/O** works. We start up a server containing a call to a database (we are omitting implementation details) which contains a table `hello` with a column named `world`, and only one row for which that column contains the string "World". The code for the server is

```
//server.js
http.createServer(function(req,res) {
  res.writeHead(200, {"Content-Type": "text/plain"});
  res.write("Hello");
  //...database implementation details omitted
  db.executeQuery("SELECT * FROM hello", function(err, rows, fields) {
    res.end(rows[0].world);
  });
  res.write(",\r\n");
}).listen(8000);
```

We start up the server and launch `curl` in another terminal window:

```
$ curl localhost:8000
Hello,
World!
```

This program output "Hello" then starts the execution of a query on the database. As you can see, the execution of the program **continues after the call** to the database (the comma is outputted after "*Hello*"), then, when the results of the query are ready, the callback (anonymous) function proceed by printing them out and terminating the execution of the server.

2 Developing a Node.js example application – *SerieALive*

2.1 Description

The application we developed to demonstrate Node.js's capabilities realize a simple "*live score update*" service. The purpose is to provide **real time information about current soccer results** as well as a **history of past seasons and matches results**. The application consists of two main views:

- The **main page**, which is displayed by default and shows the results of the current matches of the day, updated in real time, or the results of the latest finished matches. Older matches results can be browsed by choosing an older matchday, possibly in an older season.
- The **administrator page** let him/her update the results (automatically sending the updates to the clients) as well as managing data such as teams and seasons.

The main page is accessible by anyone, requesting the root path – `/` – at the application's address, while the administrator page requires **authentication** with username and password.

Admins manage the team lists (add new teams and select the teams competing in a new season) and create new championships. To perform this last task we used a variant of **Berger's Algorithm** [18] which, given a list of teams, returns a championship organized in matchdays, where every team plays against each others two time, one as the home team and the other one as the away team. This is the formula currently used in the italian national soccer championship, *Serie A*, from which the application name.
When a new season starts the administrator can make its first matchday visible by the general users on the main page, and start updating results in real time as the matches are played (currently, the software works under the hypothesis that all the matches in a match day will start at the same time). When all the matches in a matchday finish the main page displays their results until a new matchday start. Older results are still browsable by selecting the matching season and matchday, and the results can be edited by administrators at anytime if there are some errors.
Our application is quite simple and could have been realized with a lot of other frameworks or languages. If we forget the administrator page for a moment (which is basically a CRUD application) though what we want to realize is a real-time, single-page application, and therefore Node becomes the "*best man for the job*". Since the vast majority of the request should come from users that visit the main page and want to browse the results (causing a lot of quick read I/O operations server-side) while the number of update operations performed by administrators should be small, we see how Node, with its Event Loop, could perform better than a traditional web server engine. This approximative workload analysis is sufficient to justify the use of Node for our application.

2.2 Development of the application

On the server side the application is based on the **MVC pattern**. To accelerate the creation of the directory structure and the management of the architectural pattern, we used the `express` module. Express.js is a minimal and flexible Node.js web application framework, which provides a thin layer of features, fundamental to any web application, without obscuring node.js features [20].
Real-time client-server communications are implemented using the `socket.io` module. Socket.io realizes a **publish/subscribe messaging model** using the **WebSocket** protocol (although it can fall back on other methods, such as Adobe Flash sockets, JSONP polling and AJAX long polling, while continuing to provide the same interface) [19]. The subscriber waits for an event that will be fired by the publisher; the publisher does not program the message to be delivered to a specific receiver [21] and does not know if there are subscribers waiting for that event. Thanks to the use of the WebSocket protocol, the communication is based on a full-duplex channel, on a single TCP connection, which means that **both clients and server can provide data to each other**. Socket.io requires some configuration code in the main application file (in our case `app.js`), and the `socketio.js` must be provided to every client; event handling code is present both on the server and the client.
This technology is heavily used by our application: as an example, when one of the scores is updated on the administration page, an `updateScoresView` event is triggered and the update is sent to all the clients currently connected on the socket, which execute the corrispondent handling code. On the server, after the database update there will be a callback function containing the event emit:

```
socket.broadcast.emit("updateScoresView", updatedData);
```

while on the client there will be a script containing the event handling code:

```
socket.on("updateScoresView", function(receivedData) {
  var l = receivedData.length;
  //...other instructions
});
```

We used other npm modules: `mysql`, for interfacing with the database containing all the championships' data; `jade`, which makes the homonymous template engine available for use with Node; `bcrypt`, which provides a JavaScript implementation of the key derivation function for passwords based on the Blowfish cypher [22]. Other useful tools were **Twitter Bootstrap**, a front-end CSS/JS framework for faster and easier web development [23], and `git` with **Github** for code versioning and syncing.
The code for this application is available at our Github repository, https://github.com/aialenti/sistemidistribuiti.

2.3 Deploying in the cloud

We also experimented with **OpenShift**, Red Hat's auto-scaling Platform as a Service (PaaS) for applications. It allows the deployment of a small-scale web application for free, with pricing plans for larger applications. The basic plan allows a registered user to create an application slot which utilizes up to three cartridges. A **cartridge** is a basic component of the application which runs on the platform: in our case, **node.js 0.6.0** as application engine/web server, **MySql 5.1** as database management system and **PHP-MyAdmin 4.3** (a PHP interface for the database accessible from the browser). It is easy to deploy the code for the webapp: every time a user creates an application a new git repository is associated to it; this repository contains the code of the current version of the application. Upon creation is possible to specify an existing repository to clone, and subsequently sync for instant update-pushing. The only caveat to keep in mind is the use of specific **environmental variables** for application names, usernames, passwords, hostnames, IPs and ports. The latest version of our application is currently deployed and running on OpenShift and it is available at http://seriealive-ffander.rhcloud.com – although not functioning due to problems we encountered with Websockets (the protocol has just recently been implemented on OpenShift and therefore is still not 100% working).

References

[1] http://nodejs.org/

[2] http://en.wikipedia.org/wiki/V8_(JavaScript_engine)

[3] http://s3.amazonaws.com/four.livejournal/20091117/jsconf.pdf

[4] http://en.wikipedia.org/wiki/Nodejs

[5] http://www.jayway.com/2011/05/15/a-not-very-short-introduction-to-node-js/

[6] http://elegantcode.com/2012/02/06/solving-the-upload-progress-bar-problemthe-history-of-node-js/

[7] http://jaxenter.com/node-js-moves-to-joyent-32530.html

[8] https://npmjs.org/

[9] http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/

[10] http://blog.cloudfoundry.com/2012/06/27/future-proofing-your-apps-cloud-foundry-and-node-js/

[11] http://www.aaronstannard.com/post/2011/12/14/Intro-to-NodeJS-for-NET-Developers.aspx

[12] http://kunkle.org/nodejs-explained-pres/#/dead-lock-free

[13] http://www.quora.com/Node-js/Why-is-Node-js-becoming-so-popular

[14] http://mashable.com/2011/03/09/node-js/

[15] http://nodeguide.com/convincing_the_boss.html

[16] http://curl.haxx.se/

[17] http://c9.io

[18] http://it.wikipedia.org/wiki/Algoritmo_di_Berger

[19] http://en.wikipedia.org/wiki/Socket.io

[20] http://expressjs.com/

[21] http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

[22] http://en.wikipedia.org/wiki/Blowfish_(cipher)

[23] http://twitter.github.io/bootstrap/