# USING NODEJS FOR DISCRETE SIMULATION IN A WEB-BASED ENVIRONMENT

## Thomas Wiedemann<sup>(a)</sup>

(a) University of Applied Science Dresden, Germany

(a) wiedem@informatik-htw-dresden.de,

#### **ABSTRACT**

There is a quiet revolution in the web application area. Instead of using hundreds of heavy web-servers in expansive server farms, only one powerful server is used for the black Friday super-business at companies like Walmart. The secret behind it is the base software that is being used: a JavaScript engine called NodeJS for server applications. In contrast to heavy weight multitasking procedures of the operating systems with a maximum of about 200 processes at one time, NodeJS uses its own light weight multitasking in combination with Google's performance optimized JavaScript-V8 engine with up to 1 Million processes on the same machine.

This paper analyses the options for using this multitasking power for discrete simulation. The additional benefit of using JavaScript on a webserver is the easy integration in web-oriented environments.

Keywords: NodeJS, JavaScript (JS), light weight multitasking, web-based JavaScript environments

## 1. INTRODUCTION

The web community is seeing a dramatic change: nearly all web companies focus on the new JavaScript (JS) options on the server and develop new JS-frameworks for all needs like visualization, data handling and 3D-design. The first time in web history there is no programming break between the browser client and the server. By using the JS-framework NodeJS JS-code is implemented both on the browser and the client. So there is just one code base for both sides.

An additional benefit is the high performance of the whole system. In contrast to heavy weight multitasking procedures of the operating systems with a maximum of about 200 processes at one time, NodeJS uses its own light weight multitasking in combination with Google's performance optimized JS-V8 engine with up to 1 Million processes on the same machine. It was reported that Walmart's extreme black-Friday business was done by only one powerful server machine instead of a large computer farm (Pasqual 2015). Because this saves some millions of dollars, this technology will quickly emerge on all other web servers and big web solutions.

The power behind the NodeJS is the new developed V8-engine from Google. When this engine was put to work for the first time in 2010, it impressed the community with the speed of the JavaScript inside the bowser.

Instead of drawing the menus of some big web application like TYPO3 step by step, all the menus and backend options were shown at the same time; it was a visible performance jump of about 1000%.

The secret behind this performance jump is the optimized V8-JS-compiler from Google (see (Daniel Clifford, 2012)).

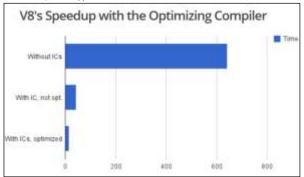


Figure 1: The performance Jump of Googles V8-engine

There is a long list of details, how to use the optimizing compiler in the correct way. In the best case with all optimizations active, the speed of the interpreted JavaScript program is nearly the same (only -17%) as a compiled C++-program. This was never achieved before and is the cause for the quiet revolution in computer science, especially in web-based apps.

### 2. NODEJS

The JavaScript based Node.js (here NodeJS) engine for server applications was built since 2010. NodeJS uses an event-driven, non-blocking I/O model that makes it lightweight and efficient (see (NodeJS 2016) for details). The further development of NodeJS has been managed since 2015 by the NodeJS Foundation which involves all big and important computer companies. So the future and full acknowledgement and support of NodeJS in the whole computer science community is nearly 100% guaranteed.

## 2.1. Code efficiency of NodeJS

NodeJS is not only powerful in the time dimension. The coding efficiency is also very high. The following code example form the Node.js homepage is all that is needed for a high-performance HTTP-webserver.

```
var http = require('http');
var static = require('node-static');

var file = new static.Server();

http.createServer(function (req, res) {
file.serve(req, res);
}).listen(1337, '127.0.0.1');

console.log('Server running ... ');
```

Figure 2: A simple HTPP-Server under Node.js

In addition, there is a very powerful package manager for Node.js, called npm. It helps to install additional Node.js modules from the global npm registry and manages their life-cycle with updates and solving dependencies from other packages.

#### 3. IMPLEMENTING THE NODE-SIMULATOR

#### 3.1. Basic architecture

The main task of the Node.js-engine and framework is the implementation of the light-weight multitasking environment for a large number of concurrent processes – the same task that a discrete simulator has.

In NodeJS the task is completed by a non-blocking layer below the main programming level. While an IO-device-access will take some time, NodeJS switches automatically to the next pending process. As a result of this **event-driven programming**, which is very different from the well-known simulation procedural sequential programming, the structure of a JavaScript NodeJS-based software is very different. The main architecture of NodeJS is shown in fig. 1. (NodeJS 2016)

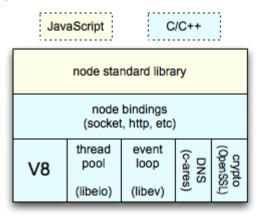


Figure 3: The NodeJS-Architecture (NodeJS 2016)

NodeJS is a combination of a number of C++-programmed basic modules, which could also be extended by new, user specific C++-modules and the JavaScript-application code above.

## 3.2. Typical code structures in NodeJS

As a result of the event-driven programming paradigm the structure of a simulation oriented program is different from the programming in procedural oriented environments like C++ or SLX.

According to the new event based programming style the code must be divided in two groups;

Blocking code will delay the further execution of the NodeJS-engine and should not be used in NodeJS. In PHP or C such code is used for IO-operations or longer OS-operations (e.g. waiting for user input). In a simulation language like SLX, the corresponding blocking code parts use the unconditional time delay advance( time) or the conditional time delay waituntil( condition). Like in NodeJS, these SLX-code-parts switch the execution sequence of the simulation processes. This is not seen in the code and unexperienced SLX-programmers will not understand the main underlying function principle of SLX (this is often seen in SLX-beginners courses)

Both the typical PHP-web program and the code of a SLX-program must be converted by the following conversion rules (see details in (Howard 2013)):

- 1. All non-blocking code can be written in the sequence shown before. If the source language is C-style compatible, this is done in a very similar manner in JavaScript.
- 2. Blocking code must be converted to a combination of calling the desired blocking function with a link to a callback to a function, which is called after completion of the time consuming task. The call back function is often provided below the function call, but can be defined also at some other place before.

By following these conversion rules a standard PHP program with some calls to blocking IO-function is converted to a chain of NodeJS callbacks (fig. 4). Please note that the NodeJS is not a sequential program, but the code exits the program after calling each fs.writeffunction and comes back a long time later (in processor time) to the callback function below the function call.

```
// PHP program writing 3 lines of text
$fp = fopen('fp.txt', 'w');
fwrite($fp, 'line1');
fwrite($fp, 'line2');
fwrite($fp, 'line3');

// corresponding NodeJS-program
fs.open('fp.txt', 'w', 0666, function(error, fp)
{ fs.write(fp, 'line1', null, 'utf-8', function() {
    fs.write(fp, 'line2', null, 'utf-8', function() {
    fs.write(fp, 'line3',null,'utf-8',function(){});
    });
});
// as a sequence of nested callbacks
```

Figure 4: A PHP to NodeJS conversion example

#### 3.3. The simulation code structure

According to the typical code structure of non-blocking NodeJS code the resulting code for a simulation with delays must be structured in the same way. Every time a simulation process must advance in time – like the SLX and GPSS commands advance( time) or the waiting construct in SLX wait\_until( condition), a combination of a corresponding function call with a callback function, where the next process steps are defined, is necessary. The resulting program code for a simulation program for NodeJS is shown in fig. 5.

Figure 5: A typical NodeJS simulation code

The simdis.advance(..) is a function call to a simulation specific function from the new simdis-module (see also 3.4). The second parameter of the function call is a nameless call back function, where the code for the simulation process after returning from the advance-waiting time is located.

The simdis.waituntil() —function is similar. The first parameter is also a function call to a predefined function (not shown in the code example), where a condition is calculated and a Boolean is returned. If the result is true, NodeJS will continue the execution to the callback function inside the second parameter.

Actually, this building of a deep nested sequence of functions is not very easy and it is very difficult to find mistakes. It would therefore be a good idea to generate such deep nested code from graphical representations of simulation models.

## 3.4. Implementation of the simulation core

The main task of implementing a simulation scheduler under Node.js is to change the common event based switching layer to a simulation time controlled module - the already well-known discrete future event list scheduler (see (Schriber 2013) for details). Instead of returning the first finished process after a somewhat random timed IO-operation, the new switching module

will precisely select the next process in simulation time from the event list and activates this process.

All the simulation specific code is located in a new module "simdis", which is loaded in the first program line in the example shown above. This module is using the same libev-C++-library for scheduling events in order to switch from a blocking scheme to a non-blocking scheme defined by function call and a callback function inside. By using this basic library of NodeJS the main event loop can also be used for the simulation scheduling.

Although first performance tests were very promising, the actual main task is the implementation of very efficient future event list modules under JavaScript.

In the actual version a very simple future event list is used for first tests. In the future version a more sophisticated event list implementation will be used, in order to test the maximal possible runtime speed. This task will be done according to well-known scheduling strategies with callback-functions like in SLX for avoiding a slow polling of conditions in blocking conditions.

It must be noted that that the described scheduling technology is based on the main single threaded NodeJS-loop which runs only on one core. If there are more cores on the hardware, there is a second option for distributing processes over different cores.

## 3.5. Horizontal distribution of processes

According to the web orientation of NodeJS and the main focus on high-parallel HTTP-server implementations, NodeJS is also able to distribute its processes over different processor cores.

This technology is efficient if the processes do not communicate too much with each other, because the communication is relatively slow in result of the used web-interfaces with TCP/IP or also faster UDP packets. The program example is fig. 6 shows a multicore simulation, where the simproc() function contains code like in fig. 5.

Figure 6: A horizontal process scheduler

One option for avoiding any communication by the distributed process is the so called Hyper computing simulation, where each single Node-process on every core executes a full version of a simulation task. Only the results of each simulation are transferred to a main controlling module on single core, which collects all results and generates the statistics and final report for all replications.

## 3.6. General performance issues under JS

JavaScript itself was long known as a slow interpreted language inside web browsers. In general this is still true, because JavaSript is still an interpreted language, which is not compiled before runtime. But the V8engine for Google changed the situation. The V8interpreter is a highly sophisticated system, which tries to start the execution of the JS-program very quickly, but during the execution checks the code more closely and generates native code for the existing processor. This native code generation depends heavily on some race conditions. The most important is the constant usage of the same type definitions over the sub-function calls. Because JS is not strongly typed, it is possible to call a function with very different variable types. In such a case the generated native code will be invalidated and the performance falls back to an interpreted very slow level. Because the web community is also interested in a very high overall performance of a NodeJS based web server, there are a lot of monitoring and testing tools for analyzing the code and its behavior (see the book "Deploying Node.js" (Pasqual 2015)).

## 3.7. Extending and running the simulation system

The supporting functions for a simulation environment like graphical presentation of results, database import and export interfaces, connections to other TCP/IP based interfaces like web-services are already ready to use from a large number of libraries like JQuery, AngularJs and D3.js.

NodeJS can run on nearly all OS without any special requirements. The Node.js-environment is free and perfectly prepared for a new style of web-based simulation. It could be expected that large cloud service providers like Amazon and Google will provide Node.js based cloud services in the near future. The new NodeSim will perfectly fit into this cloud environment and this will open new dimensions to the simulation community for joint development and usage of a very modern and open simulation framework.

Like the actual quiet revolution in web oriented development, this could also lead to new horizons in discrete simulation.

# 4. OUTLOOK TO THE FUTURE USAGE OF JS IN SIMULATION

Until now, most universal cloud providers like Microsoft's Azure, Google and Amazon offer only execution options for well understood code for their sites in result of security limitation. The reason is that it is very critical for the cloud provider to understand native machine code delivered from unknown cloud users. They therefore allow only .NET-based code with a well-defined set of .NET commands or very similar Java-bytecode.

As a result of this security barrier no well-known complex simulation environments like SLX, Enterprise Dynamics can be executed on a common cloud system. (Of course it is possible to rent a full user-defined cloud server with full OS-access and to install this software, but this very expensive and not much better than a set of existing PC's).

JavaScript and NodeJS will change this situation. Because JS is a interpreted language, the user must deliver the source code and the cloud provider can check this against its security rules. The V8-engine will then compile and optimize this code to native code for the available hardware, so there is no disadvantage against compiled code in the future. Future work will analyze the real parameters on cloud solutions and will try to use some big existing cloud systems.

## REFERENCES

Clifford, D., 2012. Breaking the JavaScript Speed limit with V8 <a href="http://v8-io12.appspot.com/#2">http://v8-io12.appspot.com/#2</a> 2012

Howard, D.. Node.js for PHP Developers. O'Reilly Media 2013

NodeJS 2016. https://nodejs.org/en/

Pasquali, S.. Deploying Node. js. Packt Publishing, 2015

Schriber T.,Brunner D., Smith J., 2013. Inside Discrete Event Simulation Software: How It Works and Why It Matters. Proceedings of the 2013 Winter Simulation Conference, Pages 424-438

## **AUTHORS BIOGRAPHY**

**THOMAS WIEDEMANN** is a professor at the Department of Computer Science at the University of Applied Science Dresden (HTWD). He has finished a study at the Technical University Sofia and a Ph.D. study at the Humboldt-University of Berlin. His research interests include simulation methodology, tools and environments in distributed simulation and manufacturing processes, intranet solutions and database applications.

Email: wiedem@informatik.htw-dresden.de