An Analysis of Node.js

By: Eric Daniels

Date: 04/08/2013

Abstract

This report is an analysis of the Node.js software framework. Node.js is a recently founded software framework that has been rapidly adopted globally. It has been seen as a direct competitor to certain web services and servers. The purpose of this paper is to give the reader an idea of what Node.js does, what it is used for, and why it would be beneficial to use it. It will cover areas such as Node.js's history, motivation for conception, core features, community, and current use in the real world. This paper assumes a working knowledge of JavaScript, threading, and peer to peer technologies.

Introduction

With the internet being the go to center for information or any kind of data in general, the amount of clients to be served increases every year. These clients can either be actual human beings or a computer itself. Regardless of the classification, the magnitude is growing quickly. As of June 30, 2012, the amount of human internet users was recorded to be 2,405,518,376 [Miniwatts Marketing Group, 2013]. That is roughly 33% of the world's population. Now if we factor in the amount of unattended computers (generally servers), this number of clients becomes even larger. In order to serve the clients over the internet, we need some kind of server. Whether it is a web server (Apache, nginx) or any kind of server, the demands on the server have increased. Disregarding bandwidth requirements, servers have either needed to become more powerful or we distribute the work of one server across multiple servers. This is where Node.js comes in.

Node.js is a single-threaded software framework that essentially lets you use JavaScript at the server side to be able to provide clients with whatever service necessary. We will discuss the history of how Node.js came about, why it was made, its core features that make it useful, the community surrounding it, and some practical applications of it. Being a fairly new technology, most of the information delivered in this paper is derived from internet resources and first-hand experience.

History

In 2006, Ryan Dahl was working on a PhD in Algebraic Topology at a university in Rochester, New York. He was growing sick of the discipline and as a result he decided to drop out of school and move to Chile, South America with the little amount of money he had left. Much to his luck, he met a guy who needed help developing PHP websites. Dahl, being a math major, had already experienced some programming and picked up the language pretty easily. This got him into the web development industry. Dahl hated the fact that Ruby on Rails was so slow and knew that it could not be fixed due to the Ruby's underlying design. After one day noticing how Flickr was able to present a progress bar for image uploads, Dahl found out they were using Mongrel (a web server written in Ruby) to be able to dynamically serve requests and send responses out to a client. Due to Ruby's poor design, Dahl set out to create software that had much better performance than Ruby that was also optimized for web experiences. [YouTube, 2011]

After a while of searching, Dahl started using JavaScript due to its lack of built in file I/O and Sockets (this let him define his own methods of performing these features). In 2009, Dahl presented what was an early version of Node.js at JSConf in Berlin. This conference proved to be a huge success and after much interest from numerous companies, Dahl chose to work for and receive backing for his project from Joyent, a high performance cloud infrastructure company . [YouTube, 2011] Node.js has been in development since then and is one the most rapidly adopted web server frameworks in the world.

Features

To start off, applications written for Node.js are executed by running the node interpreter. It provides a console for messages to be written to. It also provides a very rudimentary system for displaying information about program failure. It usually prints to the console a stack trace and some basic information about why the program failed.

An important feature of Node.js is its extension of JavaScript in the form of modules. Modules are the effective equivalent of including files for any other language. Being a server-sided framework, this extension is necessary to create modular applications. Including a module is straightforward; you use the require() function which returns an object that has members provided by the module. If one were to create a module, he/she would use the exports object to expose whatever variables or functions deemed appropriate for public consumption.

The contents of `foo.js`:

```
var circle = require('./circle.js');
console.log( 'The area of a circle of radius 4 is '
        + circle.area(4));
```

The contents of `circle.js`:

```
var PI = Math.PI;

exports.area = function (r) {
  return PI * r * r;
};

exports.circumference = function (r) {
  return 2 * PI * r;
};
```

[Joyent]

Here we have a file called circle.js that exports the area and circumference functions. Foo.js then includes circle and can then easily call on its functions as members of an object. Modules are the foundation for including any code that JavaScript doesn't already have built into itself; in this case, all of Node.js's built in features are accessible through modules. One of the most important modules that Node.js has that transforms JavaScript into a server side language is the sys module. The sys module includes system features such as arguments passed to the application, process information, and standard input/output processing.

Event Driven Programming is what drives the bulk of Node.js. Event Driven Programming (EDP) is a programming paradigm in which the flow of program execution is determined by a series of events. EDP is generally implemented by having an event "selector" that chooses and accepts existing and incoming events respectively. After selecting these events, we normally have a process that can handle these events; referred to as an event handler. [YouTube, 2011] In Node.js the idea of EDP is manifested in the form of callback functions. A callback function is a self-defining term. A callback function is a function that is potentially called at some point during program execution. These functions can be passed in as arguments that accept callback functions. A prime example of the use of callback functions is within the createServer function in the http and https libraries included in Node.js.

```
https.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}).listen(8000);
```

[Joyent]

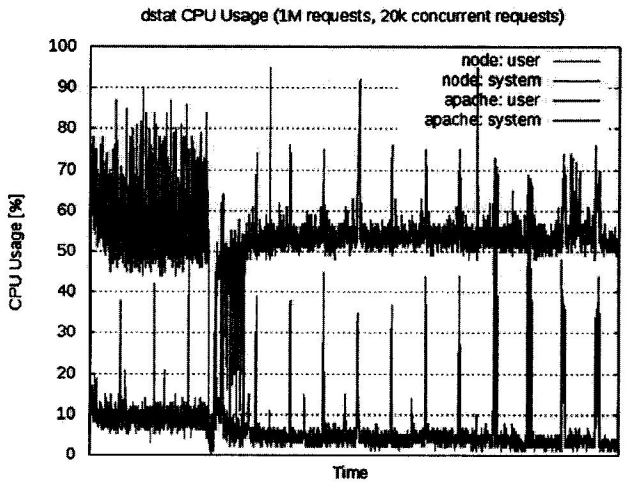
The createServer function accepts two arguments, options and requestListener. RequestListener is designated as a function that accepts two arguments, req & res (for request and response of a client). [Joyent] Any time a request is sent to the server after calling createServer, the requestListener callback function is called and it can choose to do whatever it so pleases with the request and response objects. This includes writing a response to the user. These callback functions are what allow Node.js to be singly threaded as we can build a queue of events that are processed when they need to be processed.

This directly builds into why Node.js is seen as a favorable server framework; that is, non-blocking I/O. Most programmers are taught to embrace the idea of requesting input, waiting on that input, and then processing that input. But imagine if one is to have a single threaded application that blocks every single time an I/O resource is fetched/pushed to; we run into a major problem of serving multiple

clients on webservers. It is clear that modern webservers do not have this problem as they support millions of clients. Apache, a popular webserver, solves this problem by using multiple threads. A thread is created for each connection the server. This is a perfectly valid solution for distributed computers where a mass amount of processing power is available but for single machines this becomes an issue. The issue is that we run into resource complications due to increased stack usage and the complexity of switching thread contexts. For high-demand services, failure is high on non-distributed systems using this technique.

Node.js entertains the idea of using a single thread while using EDP to process connections. When Node.js encounters an operation that would block if we waited on the I/O to process, it adds a callback to the function requesting I/O and then passes on execution to the next event in the event loop. As soon as the result of the I/O operation is ready, Node.js defers execution and heads over to the callback function to finish execution. Although all execution is on a single thread, we decrease stack usage and context switching of threads. This leads to a sharp decrease in processing time of requests so long as the processor can handle the operations as well as it would on an Apache server. As soon as no callbacks are left on the event loop, execution is considered complete. CreateServer for instance adds on a callback that never is removed from the event loop unless it is removed explicitly. It is important to note that some operations create inevitable blocking and in this case Node.js puts these operations into a thread pool which is essentially a queue of small tasks to be executed in a thread.

Maciej Zgadzaj ran a benchmark between Node.js and Apache for a series of 1 million requests where there were 20 thousands concurrent requests at a time.



His results showed that while although Node.js used more resources (memory and CPU) then Apache, Node.js was able to process all of the requests in about a quarter of the time that Apache could. [Zgadzaj, 2010] This is mainly due to not having to deal with multiple threads and the resource consumption involved in managing the threads.

From experience, Node.js appears to be best suited to applications where high performance is needed over a short interval of time for relatively small packages of data. For instance, a system that delivers statistics about a phone call in progress would be a suitable use for Node.js. This kind of system would have low I/O and buffering which allows to Node.js to perform to its full extent. If we are delivering large files and web pages then Node.js is not best suited to this task due to fallbacks of JavaScript and the underlying V8 JavaScript engine made by Google.

Community

The community surrounding Node.js can be considered one of the main proponents of its success. As of April 8th, 2013, Node.js is the second most starred repository on github. Because of the use of modules, the node package manager (npm) was made independently from Node.js. It is a package manager that installs user made modules into the system to be used within programs. There are approximately 27,000 node packages to date. Npm's website shows that thousands of modules depend on

other modules that are not built into Node.js. Some notable modules are underscore (a functional library), async, request, and express (a framework that adds more features for web development). All of these modules and Node.js itself are completely open source. This encourages the community to contribute and develop a better platform overtime.

Conclusion

In conclusion, we have seen that Node.js is still an early framework that is continuing to grow. We saw that it uses single threading in combination with event driven programming in order to provide concurrent operations. It even proved to perform better than Apache under certain conditions. This was due in part to Node.js's idea of having no blocking I/O operations unless absolutely necessary. We also saw how the extension of JavaScript to include modules gave way for community involvement in the form of module development. These modules are distributed easily by the node package manager. I definitely recommend Node.js for projects where a webserver is necessary but the delivery of files is not. Node.js is relatively simple to setup and learn due to being based off of JavaScript. As its adoption increases, more features will presumably arise and make it an overall better server platform.

Bibliography

Cunningham & Cunningham, Inc. (2013, January 11). *Event Driven Programming*. Retrieved April 8, 2013, from Portland Pattern Repository's Wiki: http://c2.com/cgi/wiki?EventDrivenProgramming

Joyent. (n.d.). *HTTPS Node.js v0.10.3 Manual & Documentation*. Retrieved April 8, 2013, from Node.js: http://nodejs.org/api/https.html#https_https_createserver_options_requestlistener

Joyent. (n.d.). *Node.js v0.10.3 Manual & Documentation*. Retrieved April 8, 2013, from Node.js: http://nodejs.org/api/modules.html

Miniwatts Marketing Group. (2013, February 17). *World Internet Users Statistics Usage and World Population*. Retrieved April 8, 2013, from Internet World Stats: http://www.internetworldstats.com/stats.htm

YouTube. (2011, October 5). *Ryan Dahl - History of Node.js - YouTube*. Retrieved April 8, 2013, from YouTube: http://www.youtube.com/watch?v=SAc0vQCC6UQ

Zgadzaj, M. (2010). *Benchmarking Node.js - basic performance tests against Apache + PHP*. Retrieved April 8, 2013, from changeblog: http://zgadzaj.com/benchmarking-nodejs-basic-performance-tests-against-apache-php