An Inside Look at the Architecture of NodeJS

Benjamin San Souci
McGill University
845 Sherbrooke Street West
Montréal, QC, Canada
benjamin.sansouci@mail.mcgill.ca

Maude Lemaire
McGill University
845 Sherbrooke Street West
Montréal, QC, Canada
maude.lemaire@mail.mcgill.ca

ABSTRACT
This paper will identify and describe the various architectural features of NodeJS, a popular software platform used for scalable server-side networking applications on the web.

General Terms
Web Development

Keywords
NodeJS, Server-side networking, Web development, Event-driven, Asynchronous, Single-threaded

1. INTRODUCTION
Initially released in 2009, NodeJS set out to revolutionize web applications. Its creator, Ryan Dahl, sought to give web developers the opportunity to create highly interactive websites with push capabilities in order to maximize throughput and efficiency. Today, dozens of companies including LinkedIn, The New York Times, PayPal, and eBay utilize Node's evented I/O model in order to power their large network programs on the web[2].

Just a few years ago, the web was a stateless environment based on the stateless request-response paradigm. Most interactive features were encapsulated within Flash or Java Applets, as isolated units within a web environment. Node allows web applications to establish real-time, two-way connections. Its major advantage lies in using asynchronous, event-driven I/O, thus remaining lightweight and efficient when managing a data-intensive application distributed across multiple devices[1].

Node is currently sponsored by Joyent, a software company specializing in high-performance cloud computing. Since its initial release as a Linux-only software platform, Node has acquired compatibility with Mac OS X, Windows, Solaris, FreeBSD and OpenBSD operating systems[2]. Contributions to the code base are made regularly by just over a dozen developers via GitHub. With nearly one commit per day, Node's fast-paced development has led to over 230 releases in just over 4 years. Nonetheless, it is imperative to note that Node has yet to release a version 1.0[4]. The project is currently open-source under the MIT license.

With larger clients looking to integrate NodeJS into their mobile platforms, this relatively new technology is increasingly influenced by enterprise application in comparison to its initial popularity among autonomous developers[1]. This trend is certain to persist as Node gains popularity, but it is doubtful to replace solutions provided by Java and .NET with much more significant worldwide investment.

This text will give an overview of the architecture of NodeJS, with a focus on a handful of key ideas that have led to its widespread adoption.

2. TECHNOLOGY
NodeJS is divided into two main components: the core and its modules. The core is built in C and C++. It combines Google's V8 JavaScript engine with Node's Libuv library and protocol bindings including sockets and HTTP.

2.1 V8 Runtime Environment
Google's V8 engine is an open-source *Just In Time*, or JIT, compiler written in C++. In recent benchmarks, V8's performance has surpassed other JavaScript interpreters including SpiderMonkey and Nitro. It has additionally surpassed PHP, Ruby and Python performance. Due to Google's approach, it is predicted that in fact it could become as fast as C[3].

The engine compiles JavaScript directly into assembly code ready for execution by avoiding any intermediary representations like tokens and opcodes which are further interpreted. The runtime environment is itself divided into three majors components: a compiler, an optimizer and a garbage collector[3].

2.1.1 Compiler
The compiler dissects the JavaScript code provided, extracting relevant commands. A built-in profiler identifies portions requiring optimization and sends these to the optimizing module[3].

2.1.2 Optimizer

The optimizer, known as Crankshaft, constructs an *Abstract Syntax Tree*, or *AST* using targeted code. It is then translated to a *Static Single Assignment*, or *SSA* representation and optimized[3].

2.1.3 Garbage Collector

V8 divides memory into two categories: the new space and the old space. Both are located in the heap and used to keep track of JavaScript objects as referenced by pointers. Any new object is added to the new space. When the new space reaches a size threshold, the garbage collector will remove any "dead" objects from the new space and store them within the old space. Although the garbage collector disallows manual memory management and may slow the web application, it is a necessary component in order to maintain a lightweight JavaScript code base[3].

2.2 Libuv

The C++ Libuv library is responsible for Node's asynchronous I/O operations and main event loop. It is composed of a fixed-size thread pool from which a thread is allocated for each I/O operation. By delegating these time-consuming operations to the Libuv module, the V8 engine and remainder of NodeJS is free to continue executing other requests[7].

Before 2012, Node relied on two separate libraries, Libio and Libev, in order to provide asynchronous I/O and support the main event loop. However, Libev was only supported by Unix. In order to add Windows support, the Libio library was fashioned as an abstraction around Libev. As developers continued to make modifications to the Libev library and its Libio counterpart, it became clear that the performance increases sought would be more appropriately addressed by making an entirely new library. For instance, Libev's inner loop was performing tasks unnecessary to the Node project; by removing it, the developers were able to increase performance by nearly 40%. Libev and Libio were completely removed in version 0.9.0 with the introduction of Libuv.

2.3 Design Patterns

Node relies heavily on Object Pool, Facade, and Factory Design Patterns. Other, less prominent design patterns which appear throughout both Node and its V8 component include the Singleton and Visitor pattern. The majority of Node's main thread is tightly coupled to the V8 engine through direct function calls. Most of the design patterns found within the V8 component extend their behavior into Node.

2.3.1 Object Pool

As resources are limited when performing I/O operations, NodeJS heavily relies on the Object Pool design pattern in order to maintain a centralized memory management system. Object pools are implemented as a list of objects available for a specific task. When one is required, it is requested from the pool manager. This design pattern is applied to Libuv's thread pool.

2.3.2 Facade

A facade is an object that provides a simplified interface to a larger body of code, such as a class library. Within NodeJS, Libuv acts as a facade around the smaller Libev and Libio libraries, the first of which allows for asynchronous I/O and the latter Node's event loop. With this structure, the library provides support for both Windows and Linux systems.

2.3.3 Singleton

A singleton class restricts the instantiation of a class to one object in order to better coordinate actions across a platform. NodeJS uses a singleton in its `ArrayBufferAllocator` in order to keep track of allocations in a centralized location.

2.3.4 Visitor

The visitor design pattern allows for the separation of an algorithm from an object structure on which it operates. The V8 engine is responsible for keeping track of various array buffers – these are interpreted by different views according to different formats such as floats and unsigned integers. As a result, a visitor pattern was constructed surrounding the global array buffer.

2.4 NPM

The Node package manager, denoted *npm*, is the official package manager for NodeJS, written entirely in JavaScript. It comes bundled and installed automatically with the Node environment. It is responsible for managing dependencies for the Node application and allows users to install applications already available on the npm registry. Npm was initially developed in 2010 with the immediate goal for quick integration into the NodeJS platform. As of Node version 0.6.3 [8].

Although it can function as a separate entity, npm is essential in keeping the small, easily maintainable nature of Node's main core. By managing dependencies outside of Node, the platform focuses on its true objectives.

2.5 Influences

The NodeJS platform is heavily influenced by the architecture of the Unix operating system. The Unix OS is composed of a small kernel; it is complemented by layers of system calls, library routines and other modules. One of its main components consists of a concurrency-managing subsystem. This schema is reproduced in Node's main thread, complemented by the Libuv component, all of which wrap around the V8 runtime environment.

Node was further influenced by the Ruby Mongrel web server, an open-source platform available on all major operating systems developed in 2008. Mongrel offered both an HTTP library and simple web server written entirely in Ruby; it was the first web server utilized by social networking service Twitter.

3. INSTALLATION OVERVIEW

NodeJS is composed of a large main code base and an additional 6 libraries; these include C-Ares, HTTP Parse, OpenSSL, Libuv, V8 and Zlib. The code is well organized and well commented, allowing any developer to easily understand Node's individual components and better assess how these interact to provide a complete server-side platform.

3.1 Installation

Installing Node is a simple process – by accessing the platform's website, a user can install the version corresponding

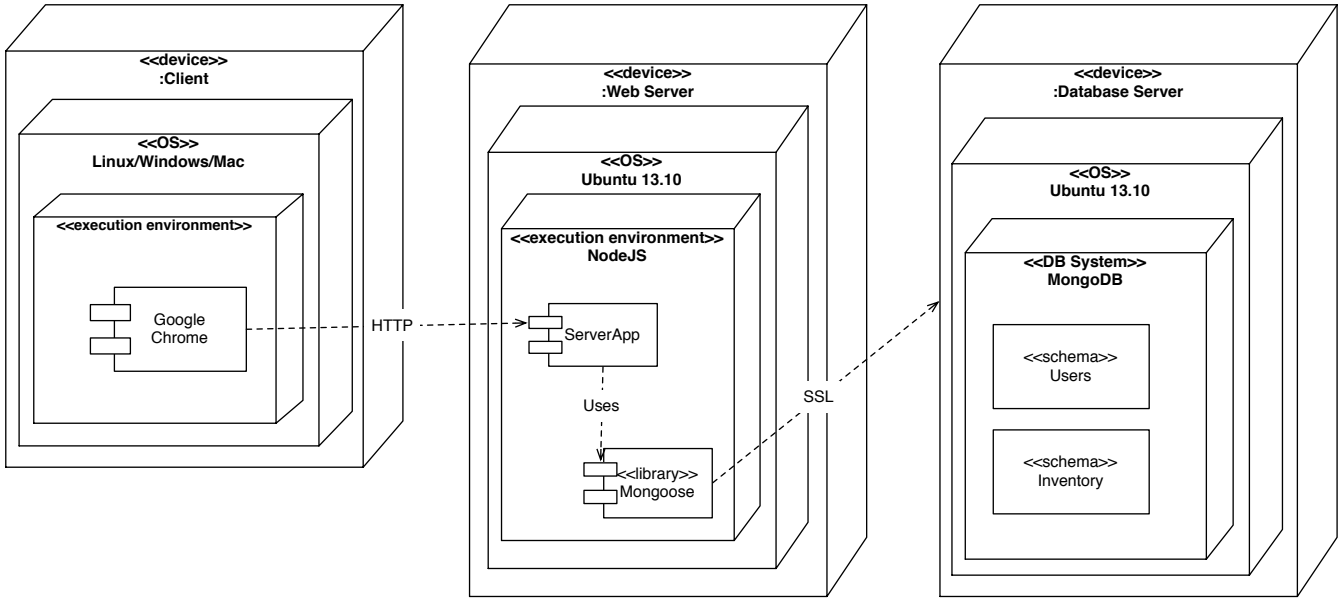

Figure 1: A sample deployment diagram of a NodeJS system. It depicts a web application running within the Chrome web browser and communicating with a NodeJS server application running on an Ubuntu server. Requests are made to a MongoDB database through the Mongoose library, which is available for download via npm, Node's package manager.

Table 1: Lines of Code

Measure	Lines of Code
Number of non-empty line	30,006
Number of non-empty and non-commented lines	15,446
Number of commented lines	2164

to his or her operating system. It is important to note that there are three major prerequisites in order to successfully install and run Node: GCC v4.2, GNU make 3.81 or newer, and Python 2.6 and 2.7. Running the platform is as easy as typing `node` into the terminal from within any directory. The command will initialize a new instance of Node and open the JavaScript interpreter. Simple examples of how to initialize a web server are provided on the NodeJS website, most of which are no longer than six lines. Figure 1 demonstrates a sample installation set-up for a NodeJS application.

3.2 Metrics

An analysis of Node's code was performed in order to better understand its composition and overall architecture. Table 1 displays results obtained from the C and C++ Code Counter's analysis of the NodeJS code base. Table 2 displays metrics relating to each of the six libraries[6].

The code base follows the following structure: a `src` folder contains all source code; an `out` folder contains compiled portions of the source code; a `deps` folder contains dependencies; and a `lib` folder contains standard libraries for the NodeJS API. CCCC identified a total of 155 classes within Node, excluding the V8 engine and other supporting libraries.

Table 2: Library Sizes

Library	Size
C-Ares	1.1MB
HTTP Parse	0.2MB
OpenSSL	24.2MB
Libuv	2.1MB
V8	58.3MB
Zlib	0.6MB
Total	86.5MB

The metrics additionally revealed that although Node appeared to be composed of nearly 30% JavaScript code, excluding its package manager, these modules were entirely dedicated to testing. Nearly 70% of the code base visible on GitHub is designated for testing[4].

4. DISCOVERY PROCESS SUMMARY

In order to better understand Node's composition and functioning, an in-depth study of the code was performed. As the entire Node code base is well organized and solely consists of around 30,000 lines, this process proved to be best for discovering subtleties in the platform.

4.1 Libraries

First, the main libraries and dependencies were identified. For each of these, we determined their main functionality within Node. This was achieved by tracing through the code and finding references to these libraries. Tight coupling between the main core and additional Node components was uncovered as a result on this analysis.

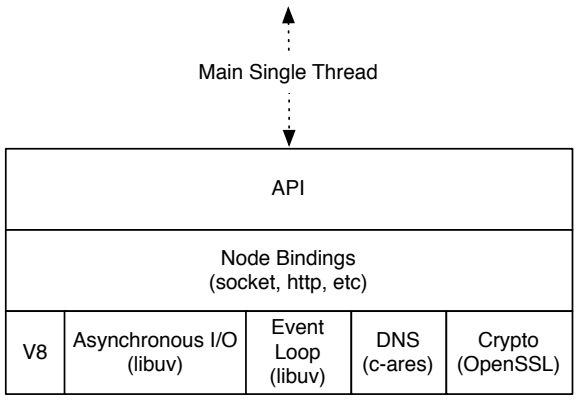

Figure 2: The internal structure of NodeJS. The main, single thread handles all incoming requests concurrently. All components are single-threaded, promote asynchronous behavior, and are able to interface with one-another. All connections are high-speed and enabled through streams of data.

4.2 Examples

Second, we used some examples provided in the NodeJS documentation to trace behavior throughout the platform. A simple web server was constructed and various requests were made in order to test predicted performance and determine if the proper results were returned. By running the simple server with code highlighting, the functions used for processing specific kinds of requests were established. For each request, the memory used and processing power consumed were monitored. This led to better understand how Node's main event loop allocated different threads for asynchronous I/O.

4.3 Blogs and Official Documentation

Third, the evolution of Node and its architecture was ascertained by reading an assortment of blogs authored by contributors to the project, including creator Ryan Dahl. Each of these detail new features released with new versions of the platform and any bugs that have been addressed. Most of these accounts, with the exception of discussing modifications to libraries, rarely discuss Node's architecture.

Node developers have produced detailed documentation for programmers wishing to use the platform, but there is little information in these sources about the actual architecture of the project. As such, there is little information released from those working on the platform concerning its structure.

5. ARCHITECTURAL DESCRIPTION

NodeJS incorporates a pipeline architecture in order to efficiently interpret input data and produce the corresponding output. It follows a very modular structure, but remains tightly coupled to its libraries, particularly the V8 engine[5]. As the platform must coordinate data input and output over a variety of distributed systems, it must be flexible and reliable. NodeJS is able to achieve this using a single thread, event loop and non-blocking I/O [1]. Figure 2 demonstrates the modular structure of the Node platform. Figure 3 demonstrates the order in which each of these components interact within a single Node instance.

5.1 Data Flow Style

The data flow architecture style equips developers with solutions for constructing a program that will process data through a series of transformations, each of which are independent of the other.

Node can be used for accessing and/or modifying data. As such, the platform incorporates a pipe and filter architecture. The system consists of a data sources, numerous filters for processing data, pipes between each of these and data sinks where data is finally dispatched.

Using streams allows Node to process data immediately in a non-blocking manner. For instance, it is able to process a file and modify it as it is being uploaded. Streams are passively processed; any data that Node receives is sent per a request from the client, and responds only to such requests.

This architecture grants Node a high level of concurrency and thus higher overall throughput.

On an internal level, Node contains a pipeline interpreter for JavaScript commands: Google's V8 runtime engine. This interpreter converts JavaScript into C++ byte by byte. The resulting code is used to perform corresponding I/O operations. The data flow style is thus an integral part of the NodeJS structure at both a macro and micro perspective.

5.2 Distributed Style

The distributed architecture style addresses important design decisions associated with a collection of computational and storage devices which communicate over a network. Generally, this consists of a remote computer independently executing an application, and communicating with a server computer equally independent over a network.

Node facilitates a client-server architecture with corresponding clients. Whether these clients are thin or fat [1], Node communicates with these clients over a network.

JavaScript is used to interface with Node from both the client and server perspectives[2].

Node uses a single thread to manage all requests and connections over the network. When Node is initialized, it informs the operating system that it should be notified when a new connection is made, and goes to sleep. With each new request from a client, a small heap allocation is generated to keep track of the connection and a callback is executed. Node handles each request sequentially using its Libuv library [1].

5.3 Implicit Asynchronous Style

Asynchronous behavior has been a major component of Node since its initial release. There are two internal components within the platform that must communicate constantly: V8

[1]This relates to the amount of business logic present within the client; thin denoting very little and fat denoting most, if not all, of the business logic is contained within the client.

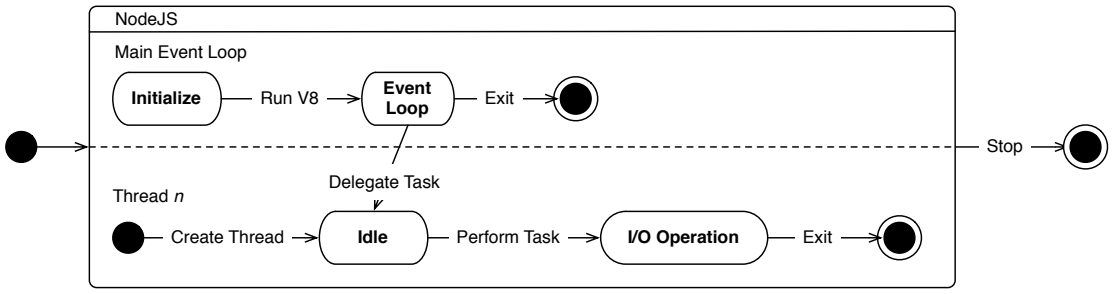

Figure 3: The various interactions between NodeJS platform components during a single Node instance. This demonstrates where how the event loop delegates tasks to a thread allocated by the Libuv library and allows for an asynchronous solution.

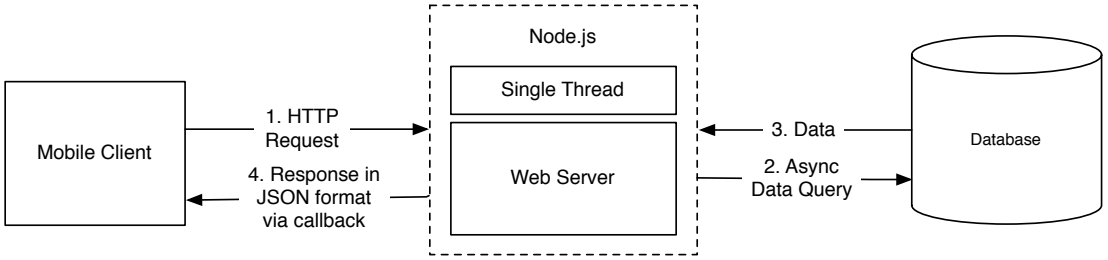

Figure 4: This example depicts a simple example of a Node JS request to access data from a database. Here, the platform acknowledges the request immediately before writing or reading any data.

and libuv. V8 and Libuv are responsible for JavaScript interpretation and I/O operations (local or networked) respectively.

For instance, suppose there exists a Node web server listening for HTTP requests. The compiled application immediately begins to listen to requests from the network. When a request is detected by the Libuv library, an event is generated and the request is linked to a callback. The callback contains a response sent to the client performing the HTTP request. When responding, the application creates another event. This event is sent to Libuv which decides proper formatting and recipient. This scenario demonstrates the constant interaction between a Node application and the Libuv library.

A similar behavior can be observed when performing a local I/O operation such as writing to a file. Events are used to communicate with Libuv and enable asynchronous I/O. The Implicit Asynchronous Style allows for scalability and modifiability, two aspects clearly demonstrated in the Node platform. Though it has a single execution thread, Node dispatched I/O operations in parallel, thus significantly increasing execution speed (as I/O tends to be a bottleneck). This portion of the application must coordinate with the operating system of the machine on which it is deployed.

Joyent fostered the development of a communication protocol between Node and its I/O operations component, enabling them to inject the proper OS-dependent library without reprogramming the Node core. This approach provides high cohesion with loose coupling between the core and main subroutines.

Node utilizes a single thread to process all incoming requests. The simplicity with which you can build an application complete with concurrent I/O is directly due to Node's early adoption of the single-threaded, event-based scheme.

5.4 Hierarchical Style

Both V8 and Libuv occupy important roles in the execution of a Node web application. The first listens for incoming events, interprets them and dispatches the task to Libuv. Next, Libuv will assign the task to one of its threads and handle the request. This behavior divides I/O operations from any business logic.

V8 currently serves as the entry point for requests within a Node application; this is indicative of the main-subroutine architecture. An important drawback to this architecture is the tight coupling between the main portion of code and corresponding subroutines. In developing Node, Joyent and its developers were conscious of this limitation, yet sought to provide a platform with a condensed core upon which additional modules could be layered. NPM was constructed, to complement the core as Node's primary package manager. It enables developers to write and share modules for Node.

NodeJS provides a JavaScript interpreter, used to compile Node modules. As a result, Node behaves similarly to a virtual machine, allowing any NPM module to execute regardless of the operating system.

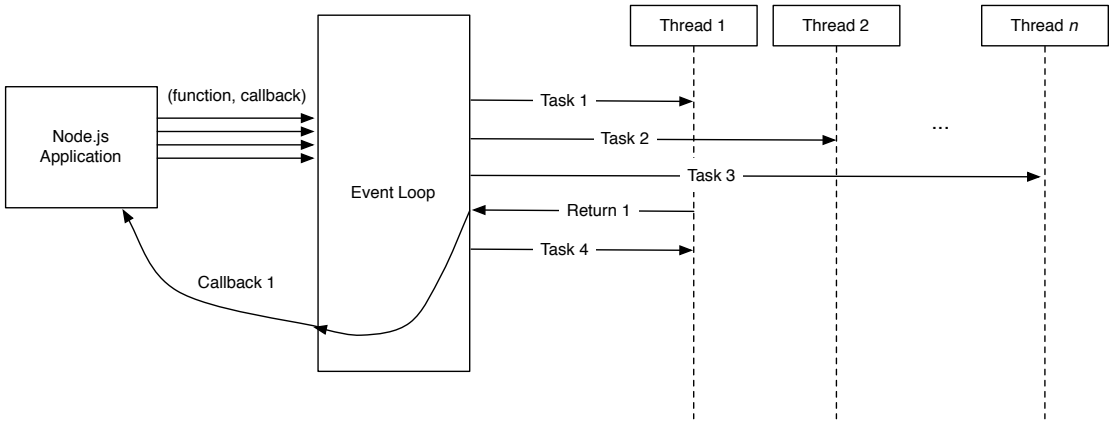

Figure 5: A closer look at the inner workings of the Event Loop.

6. CRITIQUE OF EXISTING CONTENT

There is a large assortment of tutorials dedicated to learning how to use NodeJS. Due to its popularity among autonomous developers and large enterprises alike, it is imperative to have readily available learning tools for creating a Node-based web application. Multiple developer blogs discuss Node development in addition to resources available at `NodeJs.org`. This is particularly impressive for a platform that has yet to reach a stable v1.0 [2].

Unfortunately, much of the information regarding Node is strictly directed towards learning how to use the platform and provides little insight into its actual structure and behavior. On a few occasions, Ryan Dahl spoke about the internal composition of Node and how these components interacted, but his main focus lies in functionality when speaking about the platform.

The few articles that shed light on Node's architecture lend a shallow overview and offer little to no justifications. Most of these articles are very factual in nature, and lack a critical analysis.

As more large enterprises seek to incorporate Node into their web applications, there should be a greater availability of reliable information regarding its structure and behavior. It is predicted that within the next few months, as the release of v1.0 approaches, more critical texts regarding the platform will be available to the public, clearly identifying Node's benefits and limitations.

7. CRITICAL ANALYSIS

Node has brought about a massive change in web applications since 2009; it has replaced the web's stateless environment, littered with Flash and Java Applets, into an real-time environment with two-way connections. Using asynchronous, event-driven I/O allows Node to remain lightweight and efficient, but important improvements must be made before a stable v1.0 is released.

7.1 Benefits

Because of its single-threaded, non-blocking scheme, Node can support nearly 1 million concurrent connections as opposed to typical web servers which spawn a new thread for every request. For example, a typical thread requires about 2MB of memory. If the server is running on 8GB of RAM, there is a limit of 4,000 concurrent connections. If considering the additional cost of context switching between threads, Node is considerably faster than its competitors[1].

Node's asynchronous, event-based scheme allows for scalability, and lower memory usage and CPU overhead. As previously discussed, instead of using threads for every request, Node simply allocates a heap element for each request. This is not only much faster, but utilizes less memory. As for CPU overhead, because

Requests are acknowledged quickly due to Node's asynchronous nature, allowing the web application to continue execution without waiting for Node's response. When the request has been handled, a callback is made and the application is notified.

Node facilitates native JSON handling which is very practical in today's very JavaScript-oriented web environment. it also provides easy RESTful services for establishing efficient connections between client and server[5].

Node can be installed on most major modern operating systems. It interacts with the machine using speedy native bindings in C[2].

Using its JavaScript API, Node can be used to build a web application with both a front-end and back-end entirely in JavaScript. Using the same language for client and server development can greatly decrease overall development time and lead to fewer bugs related to interfacing between different programming languages[5].

Finally, due to its real-time nature, Node makes it possible to process files while they are being uploaded. As a result, Node-based web applications dealing with heavy data flows particularly efficient[2].

7.2 Applications

Node is best suited for quick prototyping, rapidly evolving applications such as media sites, chat applications and computing and orchestration tasks divided using worker processes.

7.3 Limitations

As with any platform, there are drawbacks to Node's architecture: the first being that the main Node core and the V8 runtime engine are tightly-coupled. Any modification in the V8 engine could potentially lead to serious conflicts within the Node code base. Developers would need to reprogram a large portion of the core component in order to ensure that there were no compatibility issues.

Due to its single-threaded nature, the platform has a single point of failure for all requests, and thus low fault-tolerance. A large influx of requests within a short period of time could slow down the web server considerably. Decreases in speed are reflected throughout the platform and any faulty requests could crash Node's single thread[1].

JavaScript is known for its unique approach to exception handling. It is particularly important to developers to be careful about exception-handling when using Node. Exceptions could bubble up to the core, or top-most portion of the event-loop and cause the instance to terminate, thus crashing the web application. Alternatively, this can be avoided using callback parameters instead of throwing exceptions.

Although it is possible to monitor a Node process and recover a crashed instance, any user sessions will be unrecoverable[4].

Node is one of the most popular projects on GitHub today – hundreds of developers are contributing to its code base regularly, many of which come from different programming backgrounds. Although most of the code tends to be neat and some standards have been enforced by the project's lead developers, the platform as a whole is lacking code quality standards. Reasoning through a file which has been edited by different developers can be difficult[4].

Without a complete v1.0, Node developers are enforcing backwards-compatibility between all stable releases. Any additional code required in order to ensure this tends to bloat the code base and can lead to decreases in performance[2].

8. ALTERNATIVES

Considering the platform's main goal is to foster a large number of speedy, concurrent connections, the developers have opted for a very appropriate architecture. Asynchronous, non-blocking I/O promote this attribute, in addition to using a single thread to manage all incoming requests. It is important to note that competing web server platforms distinguish themselves from Node by adding a focus on greater reliability, such as user session recovery if corrupted or interrupted; implementing such features oftentimes decreases performance and opposes one of Node's main objectives.

In order to overcome issues stemming from tightly-coupled components, it is suggested that Node adopt a more component-based architecture, while maintaining its asynchronous, non-blocking scheme. By building more generic APIs to communicate between components such as the main core and the V8 runtime engine, developers will less likely need to alter large portions of code to promote compatibility with newer versions of the JavaScript interpreter. A greater abstraction could perhaps allow users to link a JavaScript interpreter of their choice to the Node platform, encouraging further customization of web applications with a plug-and-play approach.

9. CONCLUSION

Node provides an elegant solution for modern web developers. As a complete software platform for scalable server-side and networking applications, in just under 14 full version releases, it has fully achieved the behavior Dahl set out to deliver.

The highly anticipated release of v1.0 should provide users with a stable, fast web server solution by the end of 2014. As larger clients are seeking to integrate Node into their mobile platforms, the maturity of NodeJS is quickly increasing and so is the need for greater stability and backwards-compatibility. This increasing enterprise influence will definitely continue to play a role in its development, but it is predicted that the popularity of Node among autonomous developers will persist beyond v1.0.

10. REFERENCES

[1] T. Capan. Why the hell would i use node.js? a case-by-case introduction, 2013.
[2] R. Dahl. About node, Dec. 2013.
[3] Google. Introduction to the v8 engine, 2012.
[4] Joyent. Nodejs code base, Mar. 2014.
[5] N. Letaifa. A full javascript architecture, part one – nodejs, May 2011.
[6] T. Littlefair. Cccc-c and c++ code counter, 2010.
[7] N. Marathe. Introduction to libuv, Jan. 2014.
[8] I. Schlueter. Nom blog, Mar. 2014.

APPENDIX

A. METRICS

The metrics provided in the text were produced using CCCC, the C and C++ Code Counter.

B. SCRIPTS

No specific scripts were used to generate the metrics as the necessary data was provided by running CCCC, the C and C++ Code Counter, on the entire NodeJS code base.

C. CONTRIBUTIONS

C.1 Benjamin's Contributions

Benjamin was responsible for downloading NodeJS from the platform's website using a Linux installer. In addition, he downloaded the source code from Joyent's GitHub repository and compiled the code on his Linux (Ubuntu) machine. Benjamin then identified all major libraries, dependencies, and all major programming languages.

Ben determined the structure of the main Libuv library and Google V8 runtime engine by studying the corresponding

code. He further determined where the Object Pool and Visitor design patterns were used.

After developing a better sense of how the code functioned, he ran metrics on NodeJS using CCCC.

Ben wrote about the various aspects of the V8 runtime engine and Libuv library. He researched interactions occurring within a single Node instance in order to generate an activity diagram. He shared these findings with Maude.

Furthermore, Ben was responsible for identifying architectural styles used throughout Node and any benefits and limitations associated with each.

C.2 Maude's Contributions

Maude was responsible for downloading NodeJS from the platform's website using a Mac OS X installer. In addition, she downloaded the source code from Joyent's Github repository and compiled the code on her MacBook. Maude then identified all major libraries, dependencies, and all major programming languages.

By studying the code, Maude was able to determine the structure of the overall platform and determine where the Singleton and Facade design patterns were used. She further identified how many versions of Node had been released since its initial release date and the frequency with which the code is updated. Her research included background information on the platform, documentation to currently asses the architecture of Node and the API used to interface with the software. From this research, she created an outline of the preliminary architectural features.

Maude wrote the introduction, wrote about important technological influences and preliminary architectural features. She generated a static diagram depicting the various components of the Node platform. She edited and formated the entire text and generated the bibliography.

In addition, Maude was responsible for identifying architectural styles used throughout Node and any benefits and limitations associated with each. She researched the best uses for a Node application and found an alternative design to ameliorate the current platform's structure.