Node.js Paradigms and Benchmarks

Robert Ryan McCune, University of Notre Dame, rmccune@nd.edu

Abstract—**The current environment of web applications demands performance and scalability. Several previous approaches have implemented threading, events, or both, but increasing traffic requires new solutions for improved concurrent service. Node.js is a new web framework that achieves both through server-side JavaScript and event-driven I/O. Tests will be performed against two comparable frameworks that compare service request times over a number of cores. The results will demonstrate the performance of JavaScript as a server-side language and the efficiency of the non-blocking asynchronous model.**

I. Introduction

The Internet is continually evolving and presents a number challenges to web application designers [1]. High traffic demands services to better handle concurrent sessions [2]. Moreover, the "Slashdot Effect" demonstrates how rapidly traffic can change by several orders of magnitude [3]. Web applications must be able to efficiently serve concurrent requests.

Lauer and Needham assert a broad duality for computing, arguing that processing may be modeled by either threading or message-passing systems [4]. Web applications serving concurrent requests implement threading by assigning each incoming request to a separate thread of execution. In contrast, message-passing, or event-based systems, utilize a single thread to process events, such as incoming requests, from a queue. Each approach has inherent advantages and drawbacks depending on the implementation.

Threading is an obvious solution for managing web application concurrency, considering wide support and intuitive design. Threading is available on many platforms and naturally abstracts to independent streams of execution. However, thread programming can be exceedingly difficult and was not originally intended for user-level implementation [5]. A number of challenges arise with threading, including synchronization, where solutions add considerable complexity at drastic cost. While many common web servers like Apache implement threading [3], recent approaches have begun utilizing the event-based model.

An event-based model may provide better performance by utilizing a single event loop, avoiding the concurrency paradigm that multithreading embraces. Notably, eventing achieves non-blocking through issuing callbacks, so execution doesn't stop when a resource is already in use. Evented-I/O has gained support because of inexpensive synchronization, lower overhead for state management, better scheduling and locality, and improved flexibility in flow control [6].

Node.js is one recent framework to implement the event model through the entire stack. Developed in 2009 by Ryan Dahl, Node.js (or just Node) is a single-threaded server-side JavaScript environment implemented in C and C++ [7]. Nodes architecture makes it easy to use as an expressive, functional language for server-side programming that's popular among developers [7]. Node utilizes the JavaScript V8 engine, developed by Google [9], a fast and powerful implementation of JavaScript[8] that helps Node achieve top performance.

In the following experiments, Node will be compared to Apache, a multithreaded web server, and EventMachine, an evented Ruby web server, to evaluate practical web frameworks for the challenges posed by the current web.

II. Related Work

Having been released in 2009, Node is still beta release and currently lacks much peer-reviewed literature. The development community has posted several experiments on the web that support the evented approach [10,11,12], but Node still lacks a definitive evaluation.

In contrast with cutting-edge frameworks, the debate between threading and events has a long, established history. The need for event-based architectures has been identified in recent years [5,13,14] while thread-based concurrency remains prevalent in recent server applications [15, 16].

Both thread and event approaches have gradually evolved. Apache utilizes bounded thread pools, where the number of open threads is capped in order to limit resource allocation [3]. Moreover, event-driven concurrency has been refined, including the event queue to simplify processing and modularize code [3].

More recently, researchers out of UC Berkeley discovered a hybrid approach in SEDA, that utilizes both threads and events to create a hybrid staging component to modularize web architectures [3]. The approach performs well for high volume but adds considerable complexity to a system. While important, SEDA is not applicable to current experiments.

III. Methodology

Experiments aim to test Node, EventMachine, and Apache servers with increasing levels of concurrent and total requests.

EventMachine is a library that provides for event-drive I/O for the Ruby programming language. Ruby is a programming language developed in the mid-90's designed for ease of use. [19] Ruby on Rails is a popular web framework utilized for several applications, including Twitter, but has been criticized for lack of scalability. [21] The experiments will measure how well each framework handles varying server loads.

The test environment was created inside a virtual machine on an iMac running OS X 10.6.8, with 8 GB of memory and a 3.06 GHz Intel Core 2 Duo processor. The virtual machine was created with VMWare Fusion 4.0.2, running a fresh Ubuntu 11.10 install with 2 GB of memory. A dual core processor available allowed for the number of cores to be variable. In addition to virtualizing more recent hardware, the software also allows for machine snapshots to be taken, enabling powerful saves of machine state.

The Aptitude package manager for Ubuntu was used for installs on both test environments for simplicity and consistency. Apache is included in a fresh Ubuntu install. All additional software was installed using Aptitude except for Node itself, which was installed and configured using a popular script [20]. The Node installation method may provide performance advantages over prebuilt binaries, but will not be considered for this research. The installations included Node 0.4.12, EventMachine 0.12.10 on Ruby 1.8.7, and Apache 2.2.20.

Below is an example of web server code that will run the Node.js server

```
var http = require('http');

http.createServer(function(req, res){
  res.writeHead(200, \
    {'Content-Type': 'text/plain'});
  res.end('Hellow World\n');}) \
    .listen(8080, "127.0.0.1");
console.log('Server localhost 8080');
```

Experiments were conducted by running each framework's web server locally, hosting a simple program. The ApacheBench command line utility was used to benchmark each web server. ApacheBench offers an array of configurations. For these experiments, total number of server requests and number of concurrent requests were set variably. Below is a sample command for benchmarking a server running locally on port 80 with 10,000 total requests and 1,000 concurrent requests:

```
ab -n 10000 -c 1000 http://localhost/
```

Upon execution, concurrent requests each become an open file, so the example above would create 1000 simultaneously open files. By default, for security reasons Ubuntu only permits users a maximum of 1024 simultaneously open files. To test larger numbers of concurrent requests, the limit had to be increased

```
$ulimit -n
1024
```

but could not be immediately altered. To raise the limit, the limits.conf file was edited in /etc/security/ by adding the line

```
[user] hard nofile 500000
```

which, for the provided user, imposed a hard, enforcable limit of the number of open files to 500,000. Additional configurations were made to the /etc/sysctl.conf file by setting:

```
fs.file-max = 500000
```

After restarting the machine and logging in, the ulimit utility could successfully be used to change the user maximum open file limit to a higher limit within the new boundary

```
ulimit -n 120000
```

For these experiments, each framework is tested over differing magnitudes of total requests, from 1,000 to 1,000,000. Concurrent requests were found more significant in simulating the current web environment, so the number of concurrent requests tested falls over a finer range, including 100, 500, 1,000, 5,000, and 10,000 concurrent requests. The test machine includes a dual core processor, so each test was run over both 1 and 2 cores by adjusting virtual machine settings. A single run of each experiment is performed and was experimentally determined to provide an accurate portrayal of behavior, though additional runs could be performed in the future to increase accuracy. A successful framework will serve all requests, with the current web emphasizing concurrent requests, with a secondary objective being a faster request time, with a smaller maximum and tighter standard deviation.

IV. Results and Discussion

Results are summarized in Figures 1-6, where the mean, standard deviation, median, and maximum request times are listed for each framework over runs of variable concurrent requests, total requests, and number of cores. Figures 7-9 display the distribution of request times for each framework at lower levels of requests, where all frameworks completed their runs with similar maximums, so to not distort the graphs. The distributions reveal that majority of runtimes fall within a tight range while about 5% of runtimes are significantly higher. The median runtime for each experiment may be a more accurate statistic of a server's ability, considering a customized web application could better handle excessive request times.

For the provided environment, Node significantly outperforms the other two frameworks. Median request times are relatively comparable at lower levels of concurrency, but the biggest difference is observed at increased stress. Node is the only framework to consistently handle high levels of concurrent and total requests.

A. Node

The result may be because the particular environment favors Node's evented approach. The server application lacks complexity and transfers little data, which would favor a stateless implementation like Node.

B. EventMachine & the Evented Model

However, Node results relative to EventMachine highlight the performance benefits of the JavaScript V8 engine, as well as Node's bottom-up evented approach. Node and all the JavaScript libraries are evented, whereas EventMachine implements asychronous I/O but still includes threaded libraries. While both EventMachine and Node approach concurrency from events, Node has the more thorough implementation as well as the faster-performing language. Nonetheless, the performance of both evented frameworks in serving higher loads suggests evented-I/O may better suit the current web.

C. Apache

While quick and efficient at serving lower traffic, Apache underperformed relative to the other two frameworks. Threading may better serve more computationally-intensive web applications, so benchmarking of more complex applications may reveal workloads better served by Apache. Still, Apache displayed little improvement with an additional processor. Reconfiguring the default thread limits could result in improved performance, minding that increasing thread count consumes more memory than a comparable number of events.

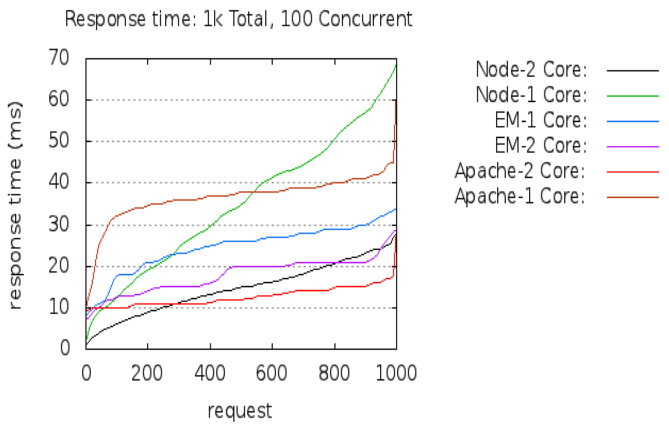

Fig. 7.

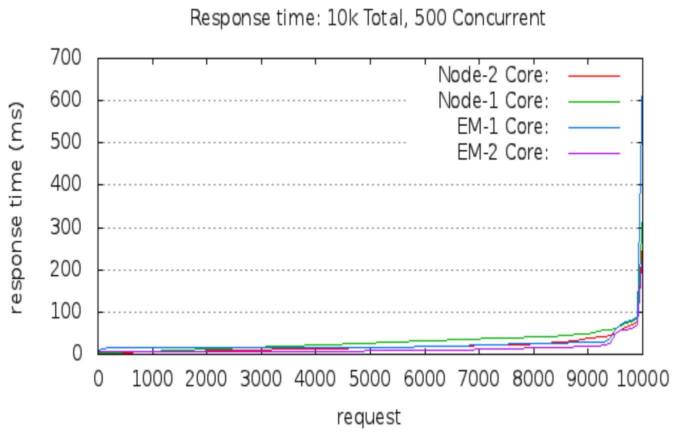

Fig. 8.

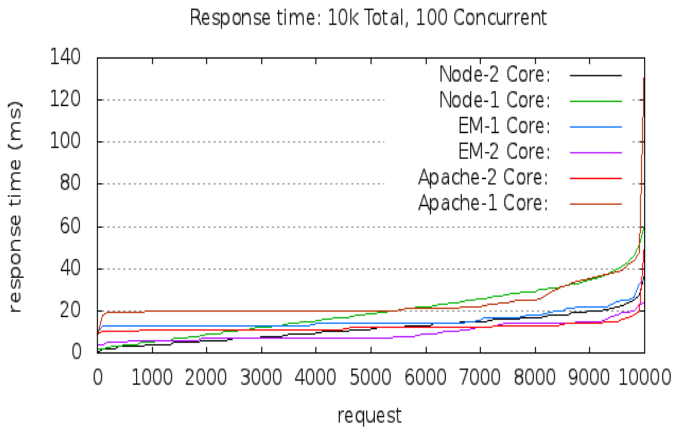

Fig. 9.

Apache - 1 Core

Total Requests

Concurrent Requests	1,000				10,000				100,000				1,000,000			
	μ	**σ**	**med**	**max**	**μ**	**σ**	**med**	**max**	**μ**	**σ**	**med**	**max**	**μ**	**σ**	**med**	**max**
100	37	5.5	38	63	24	7	20	140	23	15.9	22	3430	24	35.3	23	31993
500	324	22	580	17044	686	3152	38	22704	149	1124	51	24152	125	833.8	52	49406
1,000	*991*				*9604*				*13171*				*13611*			
5,000					*1657*				*11164*				*11666*			
10,000					*60*				*13184*				*27773*			

Fig. 1. (all times ms)

Apache - 2 Cores

Total Requests

Concurrent Requests	1,000				10,000				100,000				1,000,000			
	μ	**σ**	**med**	**max**	**μ**	**σ**	**med**	**max**	**μ**	**σ**	**med**	**max**	**μ**	**σ**	**med**	**max**
100	13	2.1	12	29	12	1.8	12	52	12	11.2	10	502	13	17.1	9	862
500	*642*				70	235	15	2557	101	1354	13	35202	69	715.2	14	84496
1,000	*518*				*9426*				*28509*				*70397*			
5,000					*5704*				*97965*				*13090*			
10,000					*9377*				*230*				*17249*			

Fig. 2. (all times ms)

EventMachine - 1 Core

Total Requests

Concurrent Requests	1,000				10,000				100,000				1,000,000			
	μ	**σ**	**med**	**max**	**μ**	**σ**	**med**	**max**	**μ**	**σ**	**med**	**max**	**μ**	**σ**	**med**	**max**
100	25	5.5	26	34	16	3.9	14	35	15	3.2	13	53	14	2.6	13	51
500	87	15.3	101	124	23	17.1	17	672	*40907*				*42494*			
1,000	166	18.7	173	205	32	33.6	18	332	*41507*				*40699*			
5,000					*N/A*				*N/A*				*N/A*			
10,000					*N/A*				*N/A*				*N/A*			

Fig. 3. (all times ms)

EventMachine - 2 Cores

Total Requests

Concurrent Requests	1,000				10,000				100,000				1,000,000			
	μ	**σ**	**med**	**max**	**μ**	**σ**	**med**	**max**	**μ**	**σ**	**med**	**max**	**μ**	**σ**	**med**	**max**
100	18	4.3	20	29	10	4.2	7	24	9	6	7	83	9	4.7	7	93
500	61	7.1	64	113	13	13.4	9	242	*76782*				*81327*			
1,000	113	9.9	114	140	20	28	9	233	57	1042	8	44475	*83868*			
5,000					60	100.1	20	668	994	4978	40	47621	706	2772	189	86655
10,000					*9998*				*94946*				*92608*			

Fig. 4. (all times ms)

Node - 1 Core

Total Requests

Concurrent Requests	1,000				10,000				100,000				1,000,000			
	µ	**σ**	**med**	**max**	**µ**	**σ**	**med**	**max**	**µ**	**σ**	**med**	**max**	**µ**	**σ**	**med**	**max**
100	35	16.3	35	69	20	11.4	19	61	18	10.2	17	100	17	9.5	17	194
500	78	44	77	191	29	20.1	27	338	77	510.8	23	9267	86	481	23	10656
1,000	164	62.9	146	293	39	42.8	29	593	127	687.4	24	9268	181	830.5	25	45133
5,000					434	415.1	508	1277	1116	3538	294	45372		*351196*		
10,000					1600	545	1531	2686		*98970*				*663990*		

Fig. 5. (all times ms)

Node - 2 Cores

Total Requests

Concurrent Requests	1,000				10,000				100,000				1,000,000			
	µ	**σ**	**med**	**max**	**µ**	**σ**	**med**	**max**	**µ**	**σ**	**med**	**max**	**µ**	**σ**	**med**	**max**
100	15	6.1	15	28	12	6.2	11	37	15	17.5	12	381	11	6.2	10	73
500	60	55	192	4338	19	15.9	15	264	49	379.2	15	9257	52	397.6	13	21076
1,000	113	32	108	185	24	60.6	15	3896	81	490.1	14	21664	100	674.8	13	21087
5,000					47	88	15	452	720	3788	30	49557	734	2199	221	57905
10,000						*9998*				*99999*			2646	5790	503	99992

Fig. 6. (all times ms)

V. Conclusions

Experiments were conducted to evaluate Node's implementation of the evented-I/O model against another evented framework EventMachine, as well as the traditional threading model represented by Apache. In an environment that simulated the current web by stressing high concurrency and low data throughput, Node outperformed both EventMachine and Apache. Results were consistent when tested over 1 and 2 cores, though request times predictably improved with the latter.

While runtimes were relatively comparable for low levels of concurrent requests and total requests. only Node was capable of serving loads at higher magnitudes. EventMachine performed well albeit inconsistently, while Apache underperformed, relative to the other two. Apache may benefit in further tests with improved threading configurations.

VI. Further Work

Much more work can be done to properly evaluate Node.js. Benchmark experiments, such as this paper, can be replicated and run many more times to gain further accuracy and precision in results. Moreover, different hooks can be added to the virtual machine to obtain a clearer picture of the inner workings of both Node as well as the other frameworks. More data on the different server processes can illuminate what elements of the Node implementation work best for given workloads. The server application can be elaborated to include more functionality, a database backend, and even a mobile device, to better simulate the current environment. Apache can be configured more appropriately to handle increased concurrent requests. While Node outperformed the competition in these experiments, further work can provide a clearer picture of Node's strengths and weaknesses.

References

[1] Labovitz, C., Iekel-Johnson, S., McPherson, D., Oberheide, J., and Jahanian, F. Internet Inter-Domain Traffic. SIGCOMM '10 (2010).

[2] L. A. Wald and S. Schwarz. The 1999 Southern California Seismic Network Bulletin. Seismological Research Letters, 71(4), July/August 2000.

[3] Matt Welsh, David Culler, and Eric Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services", ACM Symposium on Operating Systems Principles, 2001.

[4] Lauer, H.C., Needham, R.M., "On the Duality of Operating Systems Structures," in Proc. Second International Symposium on Operating Systems, IR1A, Oct. 1978, reprinted in Operating Systems Review, 13,2 April 1979, pp. 3-19.

[5] John Ousterhout, "Why Threads are a Bad Idea (for most purposes)", talk given at USENIX Annual Conference, September 1995.

[6] Rob von Behren, Jeremy Condit, and Eric Brewer, Why Events Are A Bad Idea (for high-concurrency servers), Workshop on Hot Topics in Operating Systems, 2003.

[7] Tilkov, S., Vinoski, S. Node.js: Using Javascript to Build High-Performance Network Programs. Internet Computing, IEEE, 2010

[8] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. In USENIX Conference on Web Application Development (WebApps), June 2010.
[9] Google Javascript V8, http://code.google.com/p/v8/, accessed 11/11
[10] http://zgadzaj.com/benchmarking-nodejs-basic-performance-tests-against-apache-php accessed 10/26/11
[11] http://teddziuba.com/2011/10/node-js-is-cancer.html accessed 10/26/11
[12] http://hns.github.com/2010/09/21/benchmark.html accessed 11/18/11
[13] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efcient and Portable Web Server. In Proceedings of the 1999 Annual Usenix Technical Conference, June 1999.
[14] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In Symposium on Operating Systems Principles, pages 230243, 2001.
[15] Sun Microsystems. RPC: Remote Procedure Call Protocol Specication Version 2. Internet Network Working Group RFC1057, June 1988.
[16] Sun Microsystems, Inc. Java Remote Method Invocation. http://java.sun.com/products/jdk/rmi/.
[17] http://code.google.com/apis/v8/design.html accessed 11/27/11
[18] http://rubyeventmachine.com/ accessed 11/17/11
[19] http://www.ruby-lang.org/en/ access 11/17/11
[20] http://apptob.org/ accessed 10/19/11
[21] http://blog.phusion.nl/category/ruby-on-rails/ 10/19/11