**盐城师范学院**

**毕业论文（设计）外文资料翻译**

|  |  |  |  |
| --- | --- | --- | --- |
| 学 院： | 信息工程学院 | | |
| 专业班级： | 12(4) | | |
| 学生姓名： | 余旋 | 学 号： | 12263227 |
| 指导教师： | 王林章、曹莹莹 | | |
| 外文出处： | 书名：Node.js Paradigms and Benchmarks  页码：1-6  作者：Robert Ryan McCune | | |
| 附 件： | 1.外文资料翻译译文； 2.外文原文 | | |

|  |
| --- |
| 指导教师评语：  签名：  2015年10月14日 |

1．外文资料翻译译文

Node.js的范式与基准

**摘要**

现在的web应用环境都应该具备高性能和可扩展性的环境，现有的一些web应用已经实现了线程驱动，实现了事件或两者都有。但是日益增长的网络运输量需要新的解决方法来改善并发服务。Node.js是一个新的web框架，它可以通过JavaScript运行在服务器端，是一种事件驱动的I/O型的框架。本文的研究将会比较多种框架的多个内核的服务请求响应次数。结果将会证明JavaScript作为一种服务器端语言也能表现很好，而且对非阻塞异步模型具有更高的效率。

## 1.引言

因特网的持续发展给web应用的设计者带来了大量的挑战。大量的信息交互需要服务器更好的处理并发的会话。而且Slashdot效应证明了信息量能快速以几个量级的单位改变。Web应用必须能够高效处理并发请求。

Lauer和Needham断言，一个巨大的二元计算可以通过线程或消息传递系统建模。Web应用通过给每个请求建立一个独立的线程来实现并发处理。相比之下，消息传递或基于事件的系统，利用一个单一的线程来处理事件，如队列中的输入请求。每一种方法都有其固有的优点和缺点，这取决于具体的实现上。

鉴于广泛的支持和直观的设计，线程是管理web应用并发的较好的方式。线程在许多平台都是可实现的，这自然得益于其独立的执行流。然而，由于它本来并不是为了用户级的实现，线性编码显得十分困难。许多的挑战随着线程而产生，包括同步机制，包括解决方案在成本上的复杂度也激烈增加。而许多常见的web服务器例如Apache，它的实现线程，最近已开始趋向于研究以事件为基础的模型。

一个以事件驱动的模型也许能够利用一个简单的事件池提供更好的服务，以避免包含多线程的并发范例。可以注意到，事件通过执行回调函数来避免阻塞，因此只要一个资源正在被使用，就不会停止对它的执行。事件驱动的I/O凭借着低成本同步，较低的管理费用，更好的策略和逻辑以及大大改善的流控制复杂度而获得支持。

Node.js是最近的一种通过全栈方式实现事件模型的框架。由Ryan Dahl于2009年完成开发，Node.js是用C和C++编写的单线程服务端的JavaScript开发环境。Node的出现使JavaScript在作为一种运行在服务端的表达型、功能型的语言而更加容易，也让它在开发者中越来越火。Node借助于Google的JavaScriptV8引擎，一种能快速、高效运行JavaScript的引擎，该引擎使得Node表现更加出色。

在接下来的实验中，Node将会与多线程服务器Apache，与EventMachine以及事件型Ruby语言的web服务器作比较，以此评估当前网络环境下实际web框架在面对挑战时的表现。

## 2.相关工作

自从2009发布，Node仍然只是beta测试发布版，目前也缺少同行评测的文献。开发社区虽然已经发布了几个关于能否在网络上支持事件的实验，但是Node仍然缺乏定义性的评测。

与尖端框架相比，线程与事件之间的争论由来已久。有关基于事件结构的需求在近些年已经被确定，与此同时，近年来的服务器应用中，以线程为基础的并发研究保持着持续的活跃状态。

不管是线程还是事件驱动，他们始终都在向前发展。Apache引入有数量限制的线程池，确定线程数的上限是为了限制资源的分配。另外，事件驱动的并发已经被细化，其中包括事件队列对代码的简化，和事件队列对代码的模块化。

最近，来自UC伯克利的研究人员发现了SEDA的混合算法，利用线程和事件创建一个混合的分期组件来模块化网络体系结构。这一发现在处理庞大数据方面表现很好，但也为系统带来了很多复杂的问题。同样值得注意的是，SEDA在当前试验中并不适用。

## 3.方法论

实验的目的在于检测Node.js，EventMachine和Apache服务器在并发等级和总的请求等级上的表现。

EventMachine是一个为Ruby编程语言提供的事件驱动I/O的库。Ruby是开发于90年代中期的编程语言。Ruby on Rails是流行用于一些应用的web框架，这些应用中包括Twitter，但已被证实缺乏可扩展性。实验将会测试每个框架处理不同的服务请求时的表现。

测试环境包括一台OS X10.6.8的iMac电脑，带有8G的内存和3.06GHz的Intel Core2双核处理器，虚拟机环境在VMWare Fusion4.0.2上搭建，运行Ubuntu11.10版本的系统，并分配有2G的内存。一个双核处理器可用的内核数量是可变的。除了虚拟化更多硬件，软件允许获取机器的快照，允许机器本身的能量节约。

Ubuntu的智能软件包管理工具被用于安装测试环境，来保证其简单性和一致性。Apache装在最新的Ubuntu系统中，其他所有的软件通过智能软件安装包安装，除了Node。Node使用一个流行的脚本来安装和配置。Node的安装方法也许会帮助它的表现超过一些预编译的程序。但这不会再这次研究中被考虑。主要的安装包括Node0.4.12，EventMachine0.12.10，Ruby1.8.7和Apache2.2.20。

下面是将运行在Node.js的服务器端代码的例子。

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’);

实验将每个框架的web服务器都运行在本地，服务器上运行一个简单的程序。ApacheBench命令行工具被用来记录每个web服务器信息。在这些实验中，ApacheBench一组配置信息，以及所有的服务器请求数量和并发请求数量被设置为变量。底下是一个命令样式，该例子记录了一个运行在本地80端口，总量为10000请求和1000并发量的请求代码。

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

通过以上执行，每一次并发请求变成一个打开的文件，所以以上运行结果会有10个文件同时被打开。为安全考虑，Ubuntu默认只允许同时打开1024个文件，为了测试更多数量的并发请求，这个安全限制必须提高。但需要逐步更改，为提高限制数量，需要找到/etc/security/该目录下的limits.conf文件，并添加以下代码。

[user] hard nofile 500000

这句命令为用户实行一个强制性的、不可用蛮力改变的限制。额外的配置可以到/etc/此目录下的sysctl.conf中设置，通过以下编码。

fs.file-max = 500000

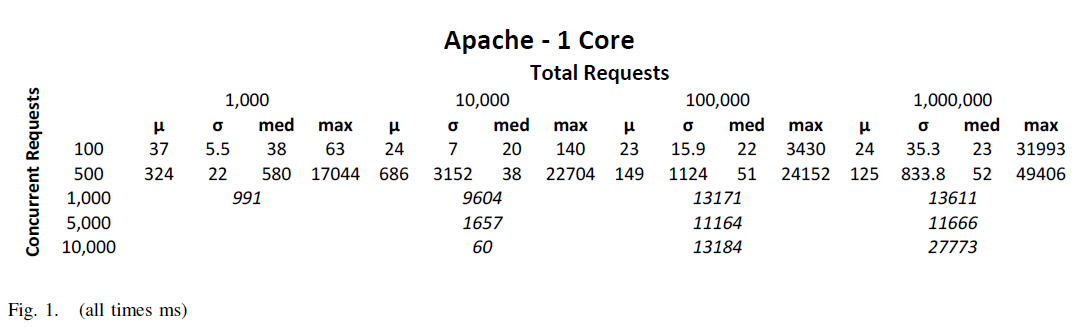
在重新启动机器登录至系统后，对限制的设定能成功的用来改变打开文件数量的最大值。

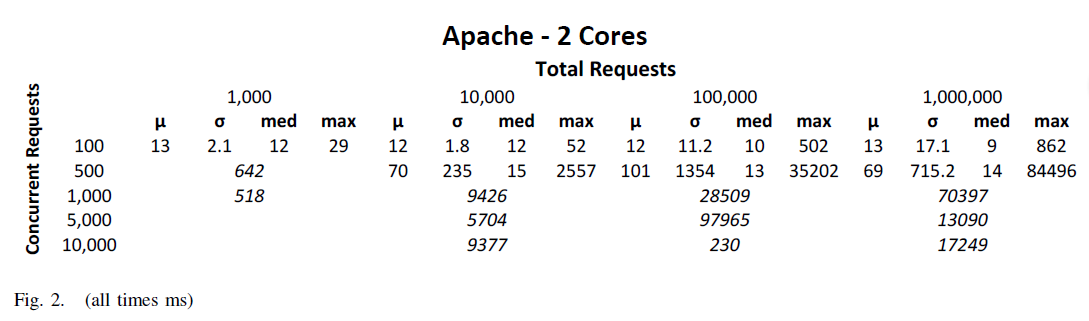
ulimit -n 120000

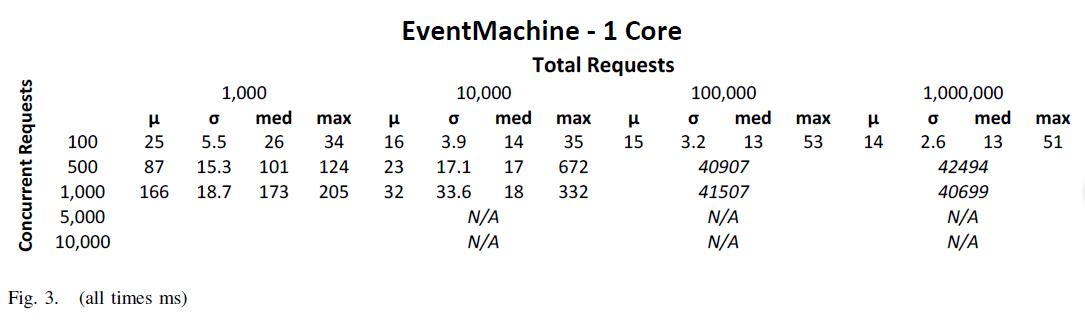
在这一系列实验中，每个框架都在不同幅度的总请求上被测试，幅度大概在1000到1000000之间。相比之下，并发请求数在模拟web环境中则显得更为重要，所以，测试的并发请求数量下降到更加精细的范围，大概为100,500,1000,5000和10000的并发请求数量。该试验机包含一个双核处理器，而每次测试运行都使用1个或者2个核，具体决定于如何适应虚拟机的配置。每个实验的单次运行都是为了提供一个准确的行为写照，即使在未来额外的运行状况会发生，实验还会提高准确度。一个成功的框架会响应所有的请求，会在当前的网络环境下强调并发请求，会为次要目标争取一个更快的请求时间，并且有更微小的最大值、更严格的标准偏差。

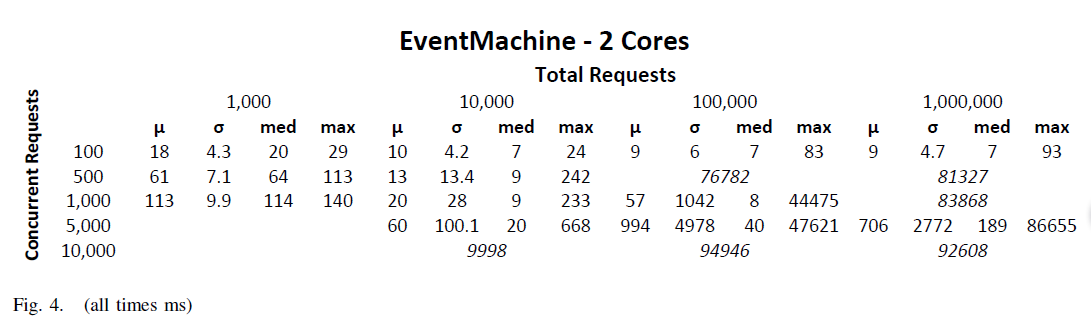
## 4.结果分析

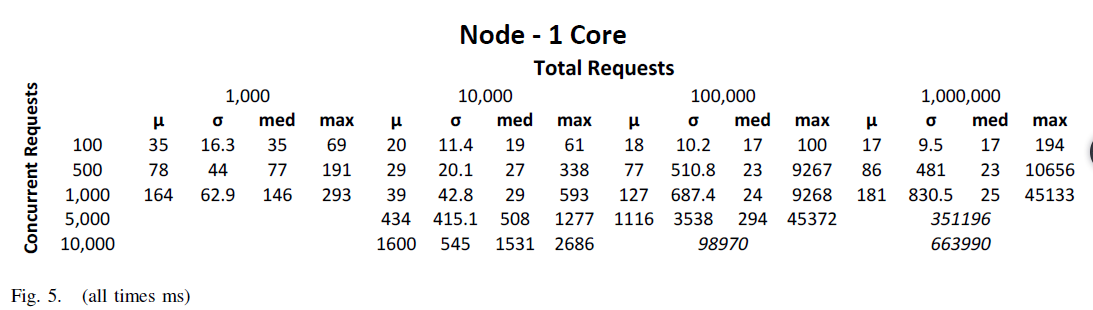
结果如图1到6所示，每个框架请求次数的标准偏差、中位数和最大值都会随着改变并发请求，总请求数和核数而改变。图7-9显示在较少请求的情况下每个框架的请求次数分布，结果所有测试的框架都完成了相似的请求次数，所以他们的曲线也是十分相似。这些分布表明大多数时候运行时间都在一个有边界的小范围内，只有大概5%的运行时间内表现得十分的频繁。考虑到一个自定义的web应用也许能更好的处理大量的请求次数，每次试验的平均运行时间也许是发挥服务器性能的精准变量。

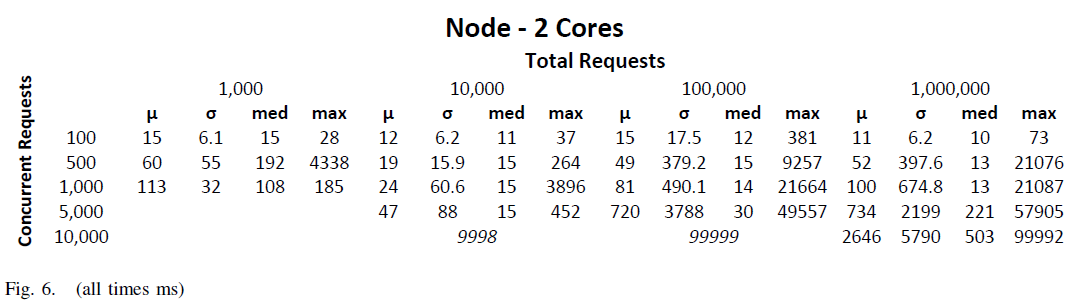












在执行完所有实验后，Node显然比其他两个框架有更好的表现，在较低水平下的并发情况下，平均的请求次数也相对的有竞争力，但最大的区别在于观察到的压力的增加。Node是唯一一个在处理高并发和所有并发两个方面都有很好表现的框架。



图7



图8



图9

1. Node

得出这样结果的原因可能是独特的环境有利于Node的事件驱动。服务器应用的简单性和传输小数据的特性，也有利于类似于Node的无状态的实现。

1. EventMachine和事件驱动模型

然而，基于Node的实验结果好于EventMachine的原因得益于高性能JavaScript V8引擎的支持，以及Node的自底向上的事件驱动。Node和所有的JavaScript库都是由事件驱动的，而EventMachine的异步I/O的实现仍然是基于线程库的。EventMachine和Node都是基于事件的并发。Node具备更深入的实现和更快速的编程语言。尽管如此，在高负荷状态下，建议所有基于事件的框架都使用事件驱动I/O来更好的适应现在的网络发展。

1. Apache

在网络传输量较低的情况下，Apache使用这两种框架的表现并不理想。线程能更好的处理数据密集型的web应用，对更为复杂的应用的标杆管理会发现Apache能提供更好的服务。随着处理器的额外增加，Apache也有了些许的改善。重新配置默认的线程限制可以提高性能，鉴于增加线程比增加相等数目事件消耗更多的内存。

## 5.结论

本文实验目的是对比另一种事件驱动框架EventMachine评估Node对事件驱动I/O模型的实现情况。同时还会将他与运行在Apache服务器上的传统的线程驱动模型作比较。在一种强调高并发和低数据吞吐量的的模拟web环境下，Node比EventMachine、Apache有着更好的表现。通过变化核数可以发现，即使到后来请求次数毫无悬念的增加了，每次实验的结论还是一致的。

在低水平的并发请求和总请求情况下，进程是相对可以比较的。只有Node可以在更高的幅度负载。EventMachine表现也很好，尽管存在一些不一致性，相比之下，Apache表现较差。随着日益改善的线程配置，在以后的测试中，Apache或许能表现的更好。

## 6.未来展望

更多更有效的任务可以用来评估Node.js。如本文中，基准实验可以重复更多次的运行来获取更准确的结果和推测。另外，设置不同的钩子并将他们加入到虚拟器中，主要目的是为了获取一个更加清楚的Node框架以及其他框架内部运行的图片。不同服务器过程中更多的数据可以阐明Node实现过程中哪一个元素对负载有缓冲作用。可以用更多的功能模块来阐述服务器应用程序，如数据库后端，甚至是一个移动设备，这些能更好地模拟当前环境。配置Apache可以更恰当地处理增加的并发请求。虽然在这些实验中，Node的表现优于竞争对手，但仍然有进一步的工作可以提供一个更清晰的结果，来表述Node的优势和劣势。

2.外文原文

**Node.js Paradigms and Benchmarks**

**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 improvedconcurrent 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.

**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.

**RELATED WORK**

Having been released in 2009, Node is still beta releaseand 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.

**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 eventdrive 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, EventMachine0.12.10 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, enforceable 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.

**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 environmentfavors 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 fasterperforming 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.

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-performancetests-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