Stack Wars: The Node Awakens

Steven Kitzes and Adam Kaplan
Department of Computer Science, California State University Northridge, Northridge, CA, U.S.A.

Keywords: Web Server, LAMP, Node.Js, Benchmark, Cloud Technology.

Abstract: As the versatility and popularity of cloud technology increases, with storage and compute services reaching unprecedented scale, great scrutiny is now being turned to the performance characteristics of these technologies. Prior studies of cloud system performance analysis have focused on scale-up and scale-out paradigms and the topic of database performance. However, the server-side runtime environments supporting these paradigms have largely escaped the focus of formal benchmarking efforts. This paper documents a performance study intent on benchmarking the potential of the Node.js runtime environment, a rising star among server-side platforms. We herein describe the design, execution, and results of a number of benchmark tests constructed and executed to facilitate direct comparison between Node.js and its most widely-deployed competitor: the LAMP stack. We develop an understanding of the strengths and limitations of these server technologies under concurrent load representative of the computational behaviour of a heavily utilized contemporary web service. In particular, we investigate each server's ability to handle heavy static file service, remote database interaction, and common compute-bound tasks. Analysis of our results indicates that Node.js outperforms the LAMP stack by a considerable margin in all single-application web service scenarios, and performs as well as LAMP under heterogeneous server workloads.

1 INTRODUCTION

Cloud technologies and their proliferation have revolutionized the ways in which we live our lives (Welke, Hirschheim and Schwarz, 2011). Service domains that have come to rely on the cloud include retail commerce, entertainment, and business functions across the board, including, in a circular fashion, software development itself. Many service and research domains would not be possible at current scale without the cloud. These include contemporary social networks and big-data analytical science. These domains leverage massive amounts of instantly-accessible data which can be collected, moved, and analysed upon a flexible collection of computing resources deployed across the world.

A need has arisen to assess the various popular web frameworks, and to produce an understanding of their performance characteristics. In the absence of such understanding, a web-service company may find itself under- or over-provisioning resources, and/or poorly utilizing developer effort. For a modern technology company developing a web-service product, an ideal design includes a thorough analysis of existing web-stack technologies, such that their strengths and weaknesses can be identified and their performance capabilities estimated. Such an analysis can guide selection of the most promising technology stack upon which to host a given service.

Existing research has focused on *scaling up* the hardware on which cloud technologies are hosted, by upgrading or expanding hardware resources to become more performant (Appuswamy et al., 2013), or to *scaling out* the host hardware by adding more computational resources upon which to distribute compute workloads (Ferdman et al., 2012). Investigations have also targeted database performance (Pokorny, 2013), resulting in extensive database benchmarking standards (Transaction Processing Performance Council, 2016). These standards may guide web-service architects through the database design space.

However, less focus has been paid to the comparative performance of the web application software being run on these hardware platforms. This software is often deployed on numerous load-balanced machines, and facilitates business logic for thousands of concurrent users or more (Chaniotis,

Kitzes, S. and Kaplan, A.
Stack Wars: The Node Awakens.
DOI: 10.5220/0006280702390249
In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017) - Volume 3*, pages 239-249
ISBN: 978-989-758-249-3

Kyriakou and Tselikas, 2015). In this work, we investigate the performance of these web application service engines as they compare under concurrent user load when run on the same hardware. We specifically focus attention on Node.js in conjunction with the Express.js framework, together part of an increasingly popular, recently deployed web application server stack known as MEAN (MEAN.JS, 2014). We compare Node.js to its more entrenched competitor, namely the Apache web server in conjunction with PHP, which together form part of the famous LAMP stack, still the most widely distributed web platform in production (Netcraft, 2015).

It is worth noting that Node.js functions as a complete platform that allows developers to write server-side software and web applications using JavaScript. By comparison, Apache does not contain an engine for executing PHP code, but rather integrates with the PHP interpreter and serves the resultant output of a given script to the requesting client as appropriate. Node.js is special in this regard, because an application can be written in JavaScript that handles all aspects of an application's duties, from serving client requests, to database interaction, to executing server side JavaScript code. Since Node.js is capable of managing static file hosting, handling business logic, serving dynamic content, and handling database and file I/O, this platform has proven to be an attractive solution, in some cases winning a place over other popular frameworks in the development environments of industry leaders including eBay, LinkedIn, PayPal, Uber and many others (Github, 2016). To these early adopters, Node.js provides tremendous community support and a broad selection of libraries made available both officially and unofficially via the NPM (Node Package Manager) system; as well as the ability to code entire web applications from front to back entirely in a single programming language (JavaScript).

Also attractive to Node.js users is the enticing possibility of superior web application performance (Chaniotis, Kyriakou and Tselikas, 2015). The attention Node.js has attracted from the community and from industry leaders generates the confidence that this is a framework that is worthy of further performance investigation, especially in direct comparison against the current industry leader, Apache/PHP.

The remainder of this paper is organized as follows. In Section 2, we provide a background description of the Node.js and Apache runtime architectures, and discuss the most closely related work. In Section 3, we detail our experimental methodology for comparing Node.js and Apache. Section 4 provides experimental results over multiple contemporary web-service benchmarks. Finally, in Section 5, we provide concluding remarks and a brief discussion of future work.

2 BACKGROUND AND RELATED WORK

Node.js employs an event loop, executed on a single thread, to carry out execution of all user defined program code, as shown in Figure 1. However, only the management of the event loop and the tasks placed upon it are executed by this single thread. Many other threads are involved in the process of managing a full Node.js instance. Node.js is built upon a bedrock of supporting technologies, most importantly Google's V8 Runtime and the libuv support library. V8 allows JavaScript to be run efficiently on server side hardware whereas libuv provides the asynchronous event handling structure which holds the promise of enhanced application performance (Libuv, 2016).

At a high level, libuv's architecture can be described as follows. An event is placed into the Node.js event queue by a client request, and the event loop will process this event. The client request will then be handled by user code. In cases requiring asynchronous operations, such as database or file system interaction, the user code will invoke Node.js (or third party) function calls for these tasks. These function calls spawn asynchronous processes on one of several worker threads running in a thread pool behind the scenes. Node.js, as well as third party library developers, provide this asynchronous behaviour by default, and implicitly encourage its

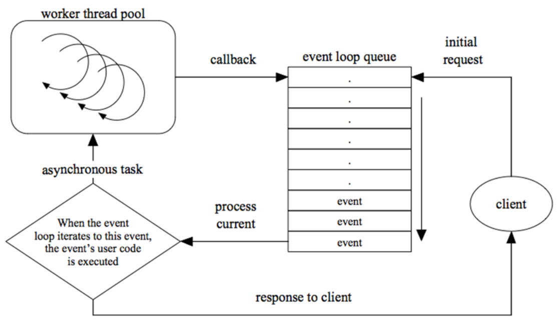

Figure 1: The Node.js Event Loop.

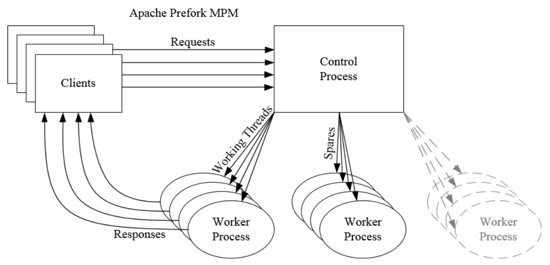

Figure 2: The Apache Prefork Multi-processing Module.

use.

Otherwise, as asynchronous tasks execute upon separate worker threads, the event loop thread continues serving other pending requests, invoking callbacks, or spawning additional asynchronous tasks upon worker threads as needed. Asynchronous tasks, once completed, return control to the application developer via callbacks, which are placed on the end of the event queue as new events, and wait their turn to be handled by the Node.js event loop as appropriate.

For performance reasons, it is critical that users take care to avoid introducing computationally heavy code, or other synchronous tasks, into either request handlers or callbacks that are executed on the event loop thread, as these block execution of the event loop. In fact, programmers must explicitly specify when and if they wish to use a synchronous version of a given operation, as any synchronous implementation will block the event loop until its task completes (Tilkov and Vinoski, 2010). However, if the asynchronous functionality that comes packaged with Node.js is employed, and the event loop remains unburdened with synchronous operations, then the load of processing each individual item on the event loop queue remains relatively light, and chiefly consists of spawning an asynchronous task on a worker thread.

As Figure 2 demonstrates, Apache employs a configurable multi-process, multi-threaded solution to concurrency. This modern Apache architecture represents a departure from the original Apache engine, which was designed at a time when concurrency was not a severely limiting factor for most use cases. The contemporary Apache maintains a thread pool (in varying patterns by version, configuration, and host operating system) for handling request service operations in parallel (Menasce, 2003). However, as mentioned above, while these processes are carried out in true parallel fashion on multiple threads – and possibly even on multiple processors – each individual thread will block in its own right while waiting on outstanding operations that might take long stretches of time, such as database or file system access. Moreover, the construction, tear-down, and context switching of these threads can be costly in and of itself. We aim to mitigate the computational overhead of Apache/PHP's management of many parallel threads, each of which may individually block as it handles a single user request. We compare against the aforementioned Node.js event queue model, where a single thread responds to multiple requests and multiple responses using the same single queue, and farms subtasks to a pool of worker threads.

The first efforts to provide web benchmarks include SPECweb and SPECjbb from the Standard Performance Evaluation Corporation. Although SPECjbb, specific to Java servers and JVMs, continues to be supported with a 2015 release (SPECjbb, 2015), SPECweb's collection of server-side applications implemented in JSP, PHP, and ASPX has retired as of 2012 (SPECweb, 2009). Although these benchmarks have been used to compare web server implementations before, including power characteristics (Economou et al., 2006) and maximum concurrent users (Nahum, Barzilai and Kandlur, 2002), the underlying architecture of these servers has changed in recent years, and the applications are no longer representative of the feature-set nor asynchronous APIs provided by modern web services. Studies using WebBench or other traffic generators to load-test Apache and other web-servers have measured server performance when accessed by tens of simultaneous clients (Haddad, 2001), rather than the hundreds or more expected on contemporary services.

Prior work has also investigated the performance of JavaScript virtual machines on different mobile platforms (Charland and Leroux, 2011), or have compared the benchmarks offered by JavaScript engines to the execution of JavaScript on the websites of famous web applications (Ratanaworabhan, 2010). Both of these studies limit performance analysis of JavaScript to client-side execution, either measured coarsely over the application duration, or by analysing fine-grain events recorded by instrumented client browsers. Although these studies compare different browsers and/or different client hardware, they do not demonstrate the scaling advantages of JavaScript when executed on the server side.

One recent study in particular measures the server-side execution of Node.js in comparison to Apache/PHP and Nginx, another open source web server competitor (Chaniotis, Kyriakou and Tselikas, 2015). The results found that for most purposes, under concurrency stress testing, Nginx performed better at scale than Apache/PHP, but also that Node.js outperformed both, except in the case of static file hosting, where Nginx was the winner. The ultimate performance solution proposed by Chaniotis et al was to develop a hybrid system in which Nginx was used to serve static files and Node.js was used for all other purposes. However, it was also acknowledged that Node.js as a singular web server environment bore other advantages, including the fact that an entire web application could be developed, front to back, all in JavaScript, a single language.

To refresh and enhance the results of prior investigation into server-side performance, we focus on the server-side benchmarking process in particular, using a number of operations commonly employed by contemporary web services. Beyond the results reported by the most recent web-server measurements (Chaniotis, Kyriakou and Tselikas, 2015), other contemporary studies focus on database benchmarking (Pokorny, 2013), and cloud scaling techniques (Ferdman et al., 2012). Moreover, the web workloads employed by Chaniotis et al are limited to static file retrieval, hashing operations, and basic client/server I/O. Thus, investigation into web application framework performance remains relatively quiet at the time of this writing. In this work, we extend the efforts of prior art by developing more extensive benchmarks that allow us to directly exercise, measure, and compare the performance of Node.js and Apache/PHP as they fare in typical client-server interactions. Furthermore, we exercise these servers with heterogeneous workloads comprised of multiple simultaneous combinations of different operations. These workloads are more representative of the diverse set of functions executed on-demand by modern web service APIs.

3 METHODOLOGY

Three physical machines were employed as nodes for this study. A Lenovo X230 ThinkPad with 8GB of RAM and a 64-bit 2.60GHz Intel i5 processor hosts a pair of virtual servers. The guest virtual machines thereon run atop Oracle VM VirtualBox, and feature a dual-core configuration with 4GB of RAM powering an installation of 32-bit Ubuntu 14.04. The server versions of Apache and Node.js are 2.4.18 and v5.6.0 respectively. Apache Bench was executed upon a client desktop machine boasting a six-core 64-bit AMD Phenom II clocked at 3.0 gigahertz and running Windows 7 on 4GB of RAM. Finally, our MySQL server was executed upon a desktop machine featuring a six-core 64-bit AMD FX clocked at 3.3GHz and running Windows 10 on 8GB of RAM.

Using Apache Bench (Apache Software Foundation, 2016), we execute several tests against both Apache/PHP and Node.js. These tests vary load, in terms of total requests made; as well as concurrency, in terms of requests sent to the server concurrently by Apache Bench. In this work, we generate up to 8192 concurrent requests for our simplest benchmark in Section 4.1.

To provide a direct “apples-to-apples” comparison, each benchmark is implemented both in PHP and Node.js using the APIs available in each framework. In Section 4.5, we employ child processes in both frameworks to measure the impact of this programming practice on workload performance.

We address performance variability in our measurements by testing each reported level of concurrency 10 times, and averaging these results into a single value represented by each bar (measurement) in the figures of Section 4. Put differently, for each benchmark, we executed each web server configuration 10 times at each concurrency level, and average these 10 executions to provide the single reported result per concurrency level per benchmark.

The metric *concurrent mean* is defined by Apache Benchmark as the mean time per request “across all concurrent requests.” This refers to the amount of time spent on each individual request, as a calculated mean value (Apache Software Foundation, 2016). If a given test executes *N* requests, at some constant concurrency level, for a total execution time of *T*, the concurrent mean is defined in Equation 1 as follows.

$$\textit{concurrent mean} = T/N \tag{1}$$

In this work, we employ the concurrent mean metric to represent how much time a server spends working on each single request, on average.

4 EXPERIMENTAL RESULTS

4.1 Web Service Baseline Measurement

In Figure 3 we compare the bandwidth in requests handled per second of Node.js and Apache under the "no-op" scenario: a simple call-and-answer test employing a small, constant response string. This test effectively reports the maximum possible performance (measured as response time) of which our server frameworks are capable, as they are performing the smallest possible amount of work per request. This will serve as a baseline against which we consider other bandwidth results from a variety of scenarios. The results of this simple test, measuring the number of requests served per unit time at exponentially increasing levels of concurrency, demonstrate a clear advantage in favor of Node.js for the "no-op" scenario. For this test, both executions of Apache Benchmark make 10000 total requests. In this test, we begin at concurrency level 2, meaning 2 requests outstanding from clients at any given moment, and ramp concurrency up by powers of 2 through concurrency level 8192.

At low concurrency levels, and as concurrency begins to ramp up slowly, Apache/PHP bandwidth wavers near 300 requests/second, before collapsing to a plateau level, so named as performance levels-off beyond (to the right of) this point. Node.js, on the other hand, performs similarly to Apache under testing conditions of up to concurrency level 256, but only wavers slightly near the plateau level of Apache/PHP. Positive results for Node.js persist longer than expected, and far beyond the concurrency level at which performance for Apache/PHP drops off.

In Figure 4, mapping mean request time against longest request service time for both Node.js and Apache/PHP at exponentially increasing levels of concurrency, we begin to see the effects of high concurrency as it taxes both of our server stacks. We see a number of interesting results surfacing from this particular manifestation of the data. First, we see that mean time and longest time do not deviate far from each other for either technology set. Next, we observe that Node.js appears to enjoy a

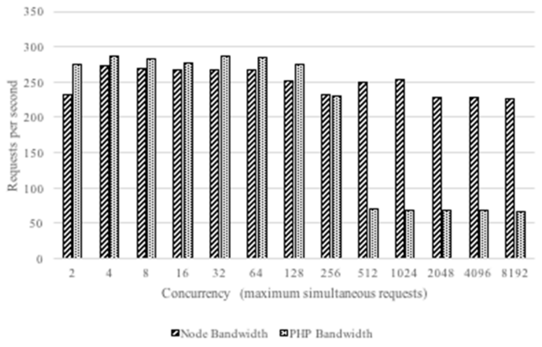

Figure 3: Bandwidth for the Baseline "No-Op" (Requests per Second).

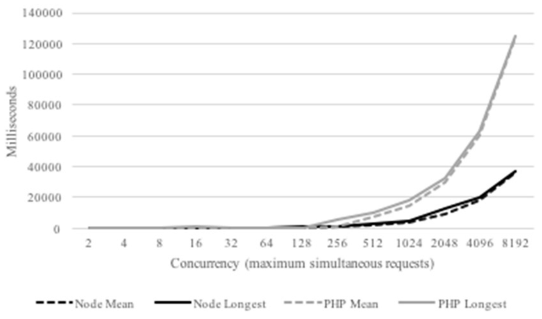

Figure 4: Mean vs Longest Request Time for Baseline "No-Op" (in ms).

tremendous advantage over Apache/PHP, although this is partially an illusion created by our charting methodology. If the same chart were presented with a logarithmically scaled Y-axis (not pictured here due to space constraints), we would see that the performance for our two technologies, with respect to mean and longest times, actually tracks linearly with respect to concurrency, and that the performance difference between the two technologies likewise tracks linearly. This result suggests that across all levels of concurrency, Node.js can be expected to maintain a considerable lead over Apache/PHP.

In Figure 5 we observe the concurrent mean across all concurrent requests. As might be predicted based on the results shown above in Figure 3, we see the average time per request rising precipitously from approx 4 to 15 milliseconds for Apache/PHP at the plateau level beyond 256 concurrent requests. The average time per request for Node.js remains relatively flat, showing no significant rise as concurrency crosses the plateau level for Apache/PHP.

4.2 Search over Large Strings

Our next test is designed to stress the computational

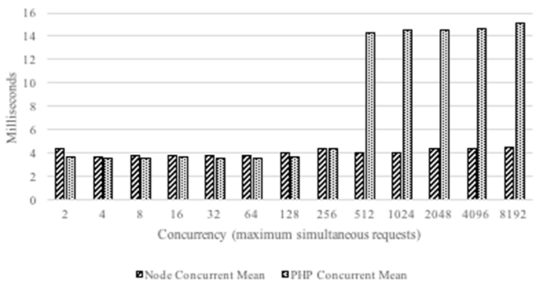

Figure 5: Concurrent Mean for Baseline "No-Op" (in ms).

efficiency of each technology. To that end, we devise a test in which a very large body of text is searched using regular expressions for target strings of various length and frequency. Again, this test was run at exponentially increasing levels of concurrency, beginning with 2 and ramping up 64. Due to time constraints and the computational cost of large string searches, we are herein prevented from testing at concurrency levels beyond 64.

In Figure 6 we see that concurrency for this particular test had no noticeable impact on performance overall, but we do see Node.js outperforming Apache/PHP in dramatic fashion, again, by a constant factor. Monitoring system performance during our tests at various concurrencies, we have observed that over all of our test runs, Node.js used 100% of all available processing resources on a single core, and Apache/PHP used 100% of all computational resources on both of the cores available to our virtual machine. In the case of Node.js, this took the form of one, single process, occupying 100% of the processor to which it had access, whereas Apache/PHP spawned many processes, each of which divided all available computational resources among themselves.

In other words, when Apache/PHP received two concurrent requests, our performance monitors report two Apache/PHP processes, each occupying 99% of computational resources (effectively 100% of all resources not demanded by the operating system). Note that this totals 198% of resources, this being due to the fact that our performance monitor reports the percentage of a given processor core's utilization that is being used by a process, not the percentage of all system-wide processor availability. (This is why we know that Node.js only occupied a single core, but did occupy the entirety of that core's computational resources, based on the 99% utilization drawn by a single process.)

When Apache/PHP received four concurrent requests, we saw reported four Apache/PHP processes, each utilizing approximately 49.5% of computational resources (again, totaling 198%, meaning 99% of each of two cores). When Apache/PHP received eight concurrent requests, we observed eight Apache/PHP processes, each utilizing approximately 24.7% of computational resources, and so on. This reinforces our understanding of the Apache/PHP model of handling multiple concurrent client requests by dedicating one process to each. Despite that, and the verification at the system level that Apache/PHP was successfully processing requests in parallel, Node.js still managed to outperform Apache/PHP by a near constant factor of more than 5 over tests at all concurrency levels.

We attempt to explain this surprise in our results as follows. First, it is possible that Node.js's string search and regular expression engine is far more efficient for this particular type of task than Apache/PHP's. It is also likely that the cost to Apache/PHP of building up and tearing down threads and processes to carry out this parallelization is damaging to its overall performance. The constant and costly context switching among Apache processes is potentially a contributing factor as well. Regardless, the evidence suggests that, for this particular type of computationally intensive task, Node.js performance is superior to Apache.

We note that in this test, performance is not impacted by increasing concurrency. We observe that the performance of this test is not bound by the efficiency of concurrency management, but by the computation of the task itself. Even if we reduce our concurrent load to a single outstanding request, the server would still require approximately the same time to service that single request, since it is the string search itself that produces a response delay.

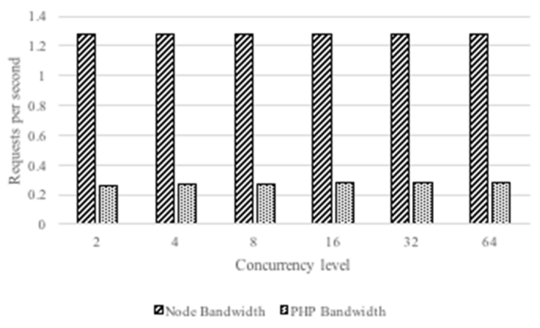

Figure 6: Bandwidth for the String Search Service (Requests per Second).

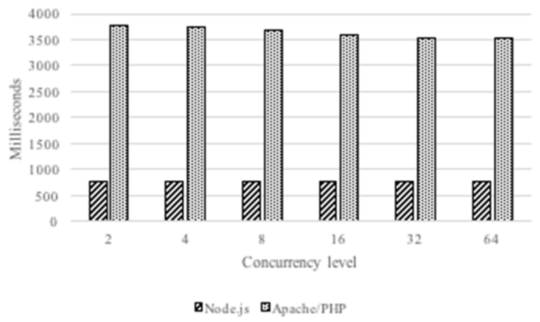

Figure 7: Concurrent Mean for the String Search Service (in ms).

Figure 7 shows the concurrent means for the large body string search tests, with results closely tracking those in Figure 6. Differences in our Node.js results are nearly indiscernible across the board in this test. The same is true for Apache/PHP, which takes a near-constant factor of 5 times longer than Node.js per request.

4.3 Serving Large Static Files

Static file service is another application domain that we investigate with respect to the performance of our two web application engines under heavy concurrent load. The goal in testing this particular scenario was to evaluate whether Apache/PHP would continue to be outperformed by Node.js in this service domain, as in prior work (Chaniotis, Kyriakou and Tselikas, 2015). Our strategy in designing this test is to stress the servers by making requests for large static files. For this test, we selected a JPEG file of approximately 3.2MB in size, representing the common transfer of profile pictures and other photos in social media and similar web service scenarios.

In Figure 8, we observe a more pronounced effect on performance with increased concurrent load for this test, as opposed to the results of the string search results in Figure 6. However, the results for both of our servers tracked more closely to each other than in any other test. This close tracking indicates that the server platform hosting our application has less impact on performance in this scenario than limitations of the host hardware. Figure 9 shows that Node.js has a small performance advantage at a concurrency level of 32 simultaneous requests and beyond, beating Apache/PHP by more than 100 ms per request (on average) at a concurrency level of 128. In this test, we were forced to limit the number of requests both in total, and under heavy concurrency, due to the amount of time required to serve static files of this size. We believe that a similar test with much smaller static files would yield results that highlight the performance benefits and limits of our servers themselves, rather than the hardware hosting them, and we consider this a prime candidate for future work.

4.4 Database Operations

The final benchmark used to compare Apache/PHP to Node.js measures their performance in fielding client requests for database operations. These requests demand little computation from the web-server itself, but require a tremendous amount of database communication by the servers to satisfy client requests. We anticipated this test would stress our server memory utilization, since state would need to be maintained on outstanding database requests, and in very large quantities as our concurrency level ramped up into the thousands. We intentionally designed the database queries to perform relatively high-latency work, such as complex join functions, in order to force our servers to hold onto request state longer.

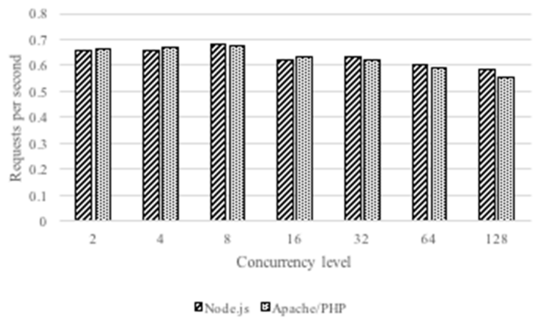

Figure 8: Bandwidth for Static Files (Requests per Second).

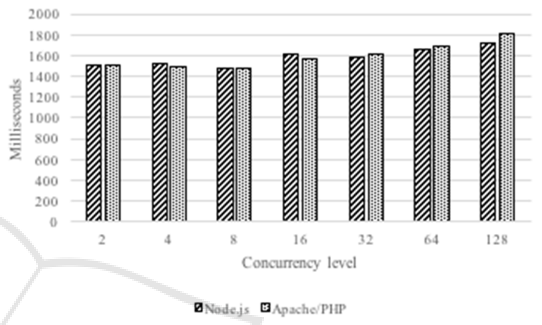

Figure 9: Concurrent Mean for Static Files (ms).

Figure 10 displays the bandwidth results for this database scenario, and shows a pattern very similar to that observed during our "no-op" testing in Section 4.1. Performance for both platforms improves early on as concurrency begins to ramp up, then levels off. Later, when concurrency ramps beyond 256 simultaneous client requests, we see Apache/PHP take a precipitous dive in performance, falling down to a plateau level very much like the one we saw during "no-op" testing. Node.js continues to perform admirably by comparison, despite some heavy turbulence in results near the same concurrency level where Apache/PHP performance drops. The reason for this plateau level cropping up in multiple test scenarios for Apache/PHP, and the reason for turbulence in

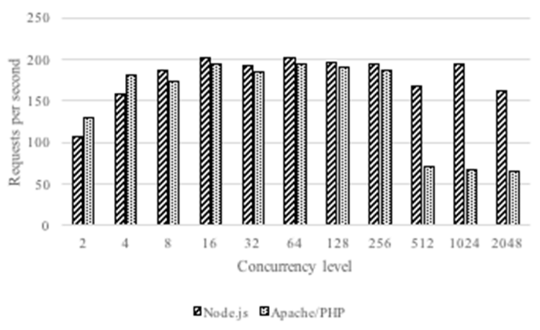

Figure 10: Bandwidth for Database Ops (Requests per Second).

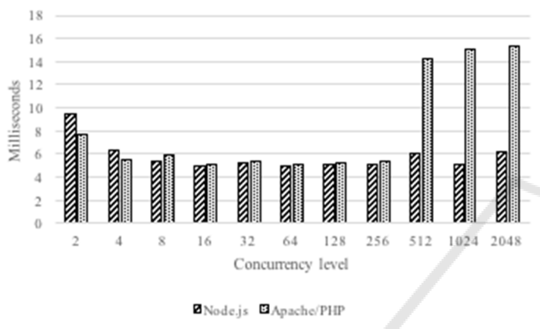

Figure 11: Concurrent Mean for Database Ops (ms).

Node.js's results pattern at the same point in its curve on our graphs, is a chief target for future work, as discussed in our conclusion below.

In Figure 11 we display the concurrent means for the same database query carried out by our two database benchmark applications on their respective platforms. As observed in Figure 10, we see the concurrent mean dropping (improving) with early concurrency ramp-up, with some light turbulence.

Later, once we reach our plateau level of 512 concurrent requests, the concurrent means for Apache/PHP skyrocket beyond 14 ms per request. Meanwhile, Node.js performance results demonstrate some turbulence, but hover around 6 ms per response as seen at lower concurrency, even as Apache/PHP experiences a sudden performance drop.

4.5 Heterogeneous Workloads

Our final test employs a heterogeneous mix of concurrent requests, generated by three simultaneously instantiations of Apache Bench – one to make requests that require heavy computation, one to make static file requests, and one to make requests necessitating heavy database interaction with the server. Instead of performing a single test multiple times at varying levels of concurrency, we ran three tests simultaneously at constant concurrency levels. These include (for both Node.js and Apache/PHP) 10,000 database interaction requests at a constant concurrency level of 64, 30 computationally heavy requests at a constant concurrency level of only 1, and 30 large static file requests at a constant concurrency level of only 1.

Since it is known to be poor practice to make the event loop responsible for time consuming computation, we employ better (if more complex) practices by moving that computation off of the Node.js event loop. This allows us to explore the impact of good programming practice on Node.js performance under a heterogeneous request load.

We accomplish this by taking advantage of child processes, which are supported by first-party libraries in both Node.js and Apache/PHP. To begin, we implemented a standalone version of our time consuming, computationally intensive large string search that could be executed from the command line. We implemented this as a standalone string-search script in PHP. Next, we adjusted our server scripts (both Node.js and Apache/PHP) so that incoming requests for this heavy computation would be handled by child processes spawned by our servers, rather than in code executed by the web servers themselves. Also, by implementing the string-search script in PHP, we remove any potential advantage that Node.js may realize via better regular expression implementation.

Of particular note is that in Node.js, the function that kicks off a child process is implicitly asynchronous, meaning that Node.js spawns the process, allowing it to run in parallel on the operating system, then receives the result packaged as a new callback event on the event loop. Thus, we anticipated that Node.js would handle our computationally heavy task in a dedicated process, freeing the event loop to handle lighter requests quickly in parallel, just as Apache/PHP does by default at scale (for better or worse).

In Figure 12, we compare bandwidth results for heterogeneous workloads with and without child processes, for both Apache/PHP and Node.js. In this heterogeneous workload, Node.js trails behind Apache/PHP with respect to database operations per second, but with the use of child processes, the margin of difference can be reduced from a gap of over 15% (without child processes) to less than 2% (with child processes). In this workload, Node.js with child processes takes the lead over Apache/PHP for static file requests, with a narrow advantage of less than 0.1 requests per second. However, without employing child processes, Node.js falls behind Apache/PHP by approximately 0.2 requests per second. Interestingly, though both Node.js and Apache/PHP are spawning identical PHP child processes to perform the string search task, Node.js performs slightly worse than Apache on this workload, losing the entire advantage it had when executing the regular expression in its own engine (without child processes).

In Figure 13 we observe that the mean time for concurrent database interactions is nearly indistinguishable between Node.js and Apache/PHP with child processes, though Apache/PHP does have a slight advantage (barely lower time per request). The mean time per static file request tracks very closely between our two applications with child processes, with Node.js eking out a narrow lead here. Comparing the mean time per string search request, however, shows a more dramatic advantage of nearly 10% for Apache/PHP with child process. This margin of victory is particularly large as the performance bottleneck for this task lies in a separate child process executing in parallel to our web applications. Moreover, this 10% advantage amounts to nearly a full second of time per request, which is significant.

We identify two potential reasons for this result. First, it is possible that the libraries used by Node.js to spawn child processes, and pass data between the Node.js context and the context of a child process itself, is itself slow, resulting in the lag time of nearly a full second when compared to Apache/PHP with child processes in Figure 13. Second, this may be an artefact of implementing the child processes in PHP, as the Apache server already has the PHP engine loaded to handle other service requests when the child process is spawned. Thus, the ready availability of the PHP engine may save time creating the child process context, as caches and other system resources are warmed-up and ready for the child code to execute. By comparison, Node.js must spin up this PHP engine from scratch when the child process is executed. This may account for the nearly full second of lag time between the two engines.

5 CONCLUSION AND FUTURE WORK

In this study, we design and execute a number of benchmark tests against Node.js and its most popular competitor and predecessor, Apache/PHP. These benchmarks are designed to determine which of these web application engines is best suited to a variety of common server-side tasks. We identify three major types of server tasks for examination, these being database interaction, heavy computation, and static file service. We compare these against a baseline “no-op” benchmark, wherein the web server always responds with a short constant string literal.

We have found that database interactions considerably favor Node.js, as do computationally heavy tasks (as represented by pattern matches on a long string). No distinguishable difference in performance was experienced between Apache and Node.js engines when serving large static files, although Node.js enjoys a modest lead at higher levels of concurrency.

We further benchmark our server engines by

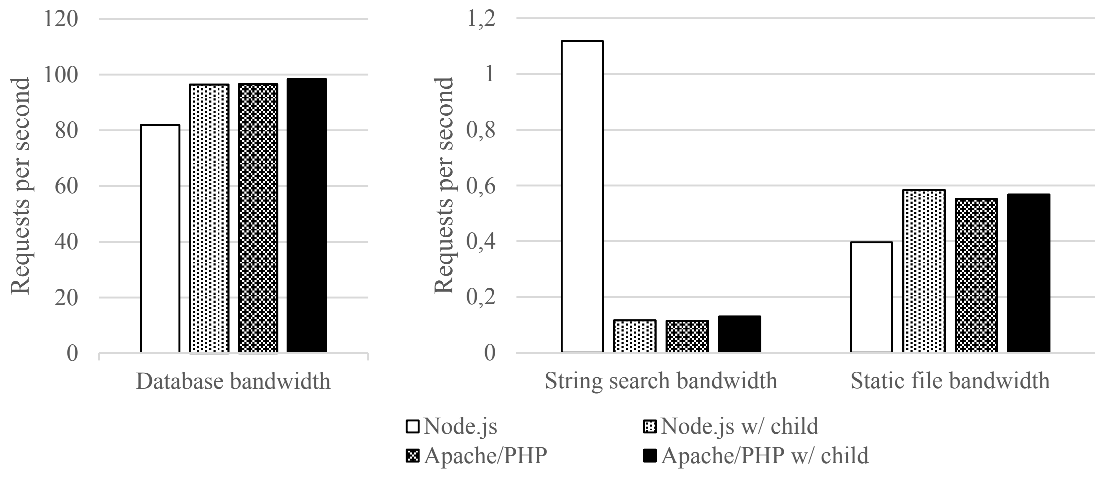

Figure 12: Bandwidth for Heterogenous Workloads with and without Child Processes (Requests per Second).

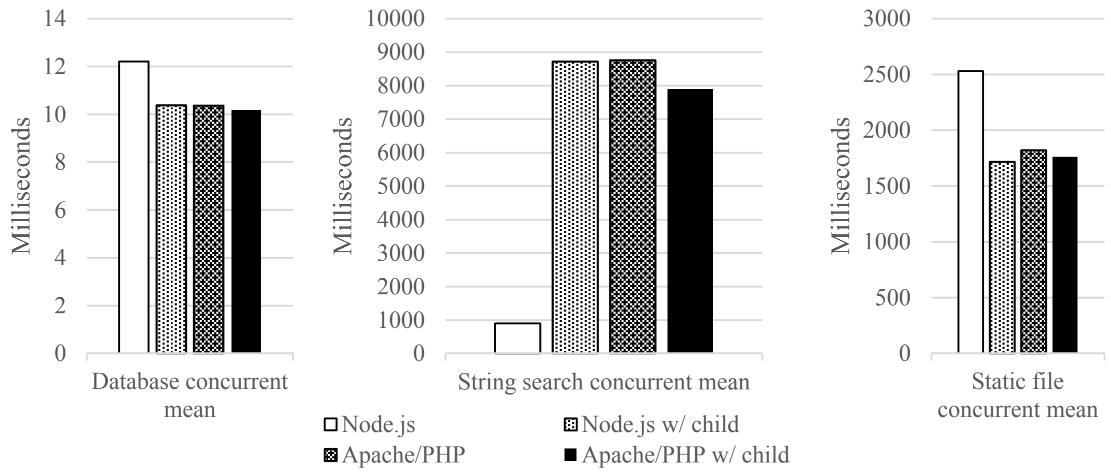

Figure 13: Concurrent Mean for Heterogenous Workloads Workloads with and without Child Processes (ms).

employing a heterogeneous mixture of request types – database, static file, and heavy computation– all being executed concurrently by each of our server stacks. In this scenario, Node.js demonstrates sensitivity to good coding practices, as computationally heavy tasks will block the synchronous event loop unless child worker processes are employed.

With the Node.js event loop freed from burden of heavy computation, Node.js falls behind Apache/PHP in performance with respect to the management of string search tasks in child processes, but manages to nearly match the performance of Apache/PHP on database interactions, and maintains a modest performance edge in static file service.

Our conclusion from the generated test data is that Node.js is the clear winner in homogeneous workload cases, i.e. wherein the web-server performs the same type of work repeatedly for numerous clients. In some tests, Apache/PHP outperforms Node.js at very low levels of concurrency, though in such cases the difference in performance between the two server stacks was insignificant. At higher concurrency levels, the advantages enjoyed by Node.js become apparent.

Although Node.js and its associated libraries enjoy growing community support and a rich repository of ready source code, PHP has a long history and a breadth of existing libraries and open templates that can be leveraged by industry adopters. We note that if abundant legacy code is an issue, or a team consists of developers with a strong background in one language or framework and no experience with another, then performance may not be the sole factor on which to base the selection of a server technology. However, all other things being equal, we can report that Node.js is the more performant server when compared with Apache/PHP. We have shown that Apache/PHP is capable of very slightly outpacing Node.js under certain types of light load, but we have also shown that Node.js is capable of keeping pace with – and sometimes dramatically outshining – its most popular competitor.

We hope to strengthen our claims in future work by experimenting with the fine-tuning of server configurations – for example, experimenting with multi-processing modules other than prefork in Apache, or tinkering with server caching settings. Moreover, exploration of this configuration space may yield custom tuning of these server engines to best suit each application type. In future work we also aim to characterize the "plateau level" exhibited by Apache/PHP, at which point performance drops and levels off with increasing concurrency. By exploring the shape of the performance curve in the neighbourhood of this plateau level, we aim to better understand Apache's behaviour at this level of concurrency, and gain further insight about its performance when hosting contemporary web services.

ACKNOWLEDGEMENTS

The authors wish to thank Professor Steven Fitzgerald of CSUN, who suggested the "no-op" scenario as a starting point, and who identified a set

of representative web-service operations that informed our benchmark selection.

REFERENCES

Apache Software Foundation, 2016. *ab – Apache HTTP server benchmarking tool.* Available at: http://httpd.apache.org/docs/2.2/programs/ab.html.

Appuswamy, R., Gkantsidis, C., Narayanan, D., Hodson, O., Rowstron, A., 2013. Scale-up vs scale-out for Hadoop: time to rethink? In *4th Annual Symposium on Cloud Computing.* ACM.

Chaniotis, I., Kyriakou, K., Tselikas, N., 2015. Is Node.js a viable option for building modern web applications? A performance study. In *Computing* (Vol. 97, No. 10m pp. 1023-1044). Springer.

Charland, A., Leroux, B., 2011. Mobile application development: web vs. native. *Communications of the ACM* (Vol. 54, No. 5). ACM.

Economou, D., Rivoire, S., Kozyrakis, C., Ranganathan, P., 2006. Full-system power analysis and modeling for server environments. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. IEEE.

Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A.D., Ailamaki, A., Falsafi, B., 2012, March. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices* (Vol. 47, No. 4, pp. 37-38). ACM.

Github, 2016. *Projects, Applications, and Companies Using Node*. Available at: https://github.com/nodejs/node/wiki/Projects,-Applications,-and-Companies-Using-Node.

Haddad, I., 2001. Open-Source Web Servers: Performance on a Carrier-Class Linux Platform. *Linux Journal* (Issue No. 91). Belltown Media.

Libuv, 2016. *Design overview – libuv API documentation.* Available at: http://docs.libuv.org/en/v1.x/design.html.

MEAN.JS, 2014. *MEAN.JS – Full-Stack JavaScript Using MongoDB, Express, AngularJS, and Node.js*. Available at: http://meanjs.org.

Menasce, D.A., 2003. Web Server Software Architectures. In *IEEE Internet Computing* (Vol. 7, No. 6). IEEE.

Nahum, E., Barzilai, T., Kandlur, D.D., 2002. Performance issues in WWW servers. *IEEE/ACM Transactions on Networking (TON)* (Vol. 10, No. 1). IEEE.

Netcraft, 2015. *September 2015 Web Server Survey*. Available at: http://news.netcraft.com/archives/2015/09/16/september-2015-web-server-survey.html.

Pokorny, J., 2013. NoSQL databases: a step to database scalability in web environment. In *International Journal of Web Information Systems* (Vol. 9, No. 1, pp. 69-82). Emerald Group.

Ratanaworabhan, P., Livshits, B., Zorn, B.G., 2010. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. *WebApps* (vol. 10). Usenix.

SPECjbb, 2015. *SPECjbb®2015*. Available at: https://www.spec.org/jbb2015/.

SPECweb, 2009. *SPECweb2009*. Available at: https://www.spec.org/web2009/.

Tilkov, S., Vinoski, S., 2010. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing* (Vol. 14, No. 6). IEEE.

Transaction Processing Performance Council, 2016. *About the TPC*. Available at: http://www.tpc.org/information/about/abouttpc.asp.

Welke, R., Hirschheeim, R., Schwarz, A., 2011. Service Oriented Architecture Maturity. In *IEEE Computer*, (Vol. 47, No. 2). IEEE.