Computing (2015) 97:1023–1044
DOI 10.1007/s00607-014-0394-9

Is Node.js a viable option for building modern web applications? A performance evaluation study

Ioannis K. Chaniotis · Kyriakos-Ioannis D. Kyriakou · Nikolaos D. Tselikas

Received: 5 November 2013 / Accepted: 15 March 2014 / Published online: 28 March 2014

Abstract We examine the implications of end-to-end web application development, in the social web era. The paper describes a distributed architecture, suitable for modern web application development, as well as the interactivity components associated with it. Furthermore, we conducted a series of stress tests, on popular server side technologies. The PHP/Apache stack was found inefficient to address the increasing demand in network traffic. Nginx was found more than 2.5 times faster in input/output (I/O) operations than Apache, whereas Node.js outperformed both. Node.js, although excellent in I/O operations and resource utilization, was found lacking in serving static files using its built in HTTP server, while Nginx performed great at this task. So, in order to address efficiency, an Nginx server could be placed in-front and proxy static file requests, allowing the Node.js processes to only handle dynamic content. Such a configuration can offer a better infrastructure in terms of efficiency and scalability, replacing the aged PHP/Apache stack. Furthermore we have found that building cross platform applications based on web technologies, is both feasible and highly productive, especially when addressing stationary and mobile devices, as well as the fragmentation among them. Our study concludes that Node.js offers client-server development integration, aiding code reusability in web applications, and is the perfect tool for developing fast, scalable network applications.

I. K. Chaniotis · K.-I. D. Kyriakou · N. D. Tselikas (✉)
Department of Informatics and Telecommunications, University of Peloponnese,
End of Karaiskaki street, 22100 Tripolis, Greece
e-mail: ntsel@uop.gr

I. K. Chaniotis
e-mail: tst08037@uop.gr

K.-I. D. Kyriakou
e-mail: tst07086@uop.gr

Keywords Node.js · End-to-end JavaScript · WebSockets · WebRTC · Real-time Web · Software performance evaluation

Mathematics Subject Classification 68M20 Performance evaluation; queueing; scheduling · 68M11 Internet topics · 68M15 Reliability, testing and fault tolerance · 68M14 Distributed systems · 68N01 General

1 Introduction

The birth of the World Wide Web was facilitated by a social interaction through the internet, when in 1991, Tim Berners-Lee, posted a summary of the project on the alt.hypertext newsgroup on Usenet, one of the oldest computer network communications systems still in widespread use [1]. Six years later, the first recognisable social network site launched, SixDegrees.com [2]. In its fourth quarter earnings results for 2013, Facebook, the current largest social network, revealed that it has over 1.23 billion monthly active users and 945 million mobile monthly active users [3]. The viral growth of social networks and applications today can be explained through both societal developments and technological advancements, that together have enabled new types of applications where users co-create content. The spreading effect of social networking can be attributed to another factor, the emergence of the iPhone, ultimately leading to the smartphone culture [4]. Web enabled devices have become extremely commonplace, with mobile devices leading the way, but this brings to developers a challenge [5]. This is the fragmentation which runs both length (device fragmentation) and breadth (operating system fragmentation) across the mobile landscape. Devices with different processing, memory, communication and displaying capabilities are examples of device fragmentation [6]. There are different companies with their own platforms, running different operating systems and requiring expertise in each development environment, making it hard and costly to address multiple devices. As such, a traditional native approach is not always an ideal solution. Mobile phones were social tools to begin with, by making interpersonal communications portable. With world population 3G coverage estimated to reach 85 % by 2017, it makes sense to use them as such online too, after they evolved into essentially web clients to be carried around, at all times [7].

What all platforms have in common, is the increasing compliance to the web standards brought to them through modern web browsers. Technological achievements, such as the improvement of modern browsers and the standardization of HTML5, led to the creation of advanced web applications, offering features only available to native applications before. This phenomenon of humongous interaction with the web has brought up some issues that need to be addressed, as its impacts are of the same extent. The infrastructure software had to evolve to address the service demand. “The cloud” is the current paradigm of computing, building on this notion of co-creating both content and services [8], while a promising protocol for real-time client-server data exchange, i.e., the WebSocket protocol, was standardized just two years ago, on December 2011 [9]. With JavaScript being one of the most used and dominant web technologies for client side programming, the birth of server side JavaScript i.e., Node.js on May 2009, offered the potential of client-server programming integration, by using the same language end-to-end. We were triggered to test and evaluate whether a pure JavaScript based environment could provide great synergy with cross-device web application development. In other words, in this paper we try to answer in a frequently asked question of web programmers: “Is pure JavaScript a viable option for building modern web apps?” In that line we compare and evaluate the performance of Node.js against the most deployed web language i.e., PHP by using the two most deployed and popular open source web servers, Apache and Nginx, occupying above 60 % of the market share [10,11]. We combine the findings and the results, in order to conclude whether JavaScript is the optimal way to build end to end systems for modern web applications, from the infrastructure layer, to the user’s experience, with great ease and efficiency, and not just an ephemeral trend.

In Sect. 2 we present the tendencies related to the trending web technologies coupled with current implications and corresponding research. In Sect. 3, the design and implementation principles are presented followed by the server and the client architecture. Section 4 describes the experimental environment and the methodology followed, while Sect. 5 presents the results and an elaboration of the findings. Finally, in Sect. 6 we summarise the research conducted.

2 Current tendencies and challenges

Web applications can be defined as “software systems based on technologies and standards of the World Wide Web Consortium (W3C). They provide Web-specific resources such as content and services through a user interface, the Web browser” [12]. HTML5 is a markup language, used for structuring and presenting content for the World Wide Web and a core technology of the Internet, maintained by W3C [13]. HTML5 is also used as a simplifying term, for addressing a family of other related web standards and technologies, like CSS3 and JavaScript together, with which it represents the complete package, or idea, that is HTML5 [14].

The evolution of smartphones, tablets and the corresponding mobile operation systems (OS) boosted the mobile applications’ bloom. Mobile applications are applications that run in mobile devices and they are categorized into native, hybrid or pure web applications, respectively. Native applications are vertically developed for use on a particular operation system. They are coded in specific programming languages, e.g., Java for Android platform, Objective C for iOS, etc. They provide the richest and most compelling user experience, since they are optimized for the specific operation environment of the device they run on, they can fully leverage device software and hardware capabilities and they have the ability to run offline. The obvious main disadvantage of native mobile applications development is the cost, since native applications require to be re-coded from scratch for each native platform. In order to reduce this cost, the hybrid approach can be followed. Hybrid mobile applications is the marriage of web technologies and native execution. They run inside a native container and leverage the device’s browser engine—but not the browser—to render the HTML and process the JavaScript locally. They are built with cross-platforms frameworks, such as PhoneGap [15], Appcelerator [16], Worklight provided by IBM [17], RhoMobile provided by Motorola [18] to name a few, which allow developers to write an application once and use the framework to adapt it for multiple platforms. Web-technologies are in the foreground once again, since HTML5 and JavaScript are usually required to use these frameworks, with RhoMobile Suite, which is based on Ruby on Rails, to be an exception. The main disadvantages of hybrid mobile applications are that each platform presents different pros and cons, while, in practice, hybrid mobile applications require tweaking for each platform [19]. Pure mobile web applications are software that use web technologies, i.e., once again JavaScript and HTML5, to provide interaction, navigation, or customization capabilities. Mobile web applications run within a mobile device's web browser. This means that they are delivered on the fly via the internet and they are not separate programs that need to be stored on the mobile device, thus, they present the key advantage of the widest reach. The main drawback of mobile web applications is that lack on user experience and performance [19]. In order to increase the performance and offer a user experience similar to that of a native application, web applications can be developed as single page applications. Single page applications load required resources either in a single page load, or load appropriate resources dynamically, in response to user actions. Based on HTML5, on structured JavaScript in both client and server side and on responsive design technique, there are no page reloads at any point and thus the user experience is differentiated from the usual web page browsing experience [20].

Several related studies address the challenges of modern web applications. Lautamki has examined the problems of modern web application development, as well as client-server real-time communication possibilities, concluding that newer tools such as Node.js, the WebSocket protocol and WebRTC (Web Real-Time Communication) aid in realizing the Real-Time Web [21]. Taivalsaari et al. [22] focus on the increasing usage of the Web as a software platform. They have concluded that "the transition towards web-based applications will mark the end of binary end user software". Kai Shuang and Feng Kai, have compared different methods of server push technologies, and have evaluated their delays and unnecessary connection costs. They have concluded that WebSocket is superior to all the other methods examined [23]. Several efforts have evaluated the advantages and limitations of different cross platform development solutions, and have arrived to the conclusion, that web based applications are viable options [23–26]. Furthermore, several companies such as Microsoft, LinkedIn,

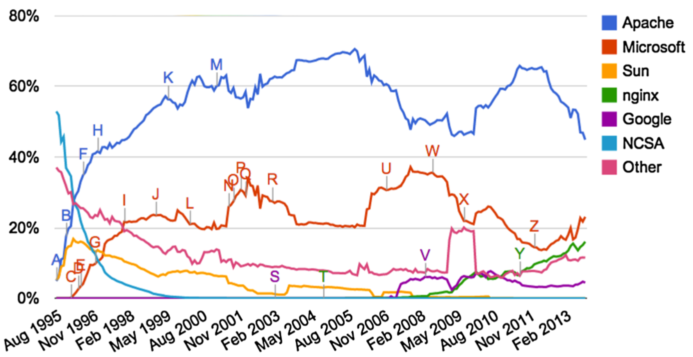


Fig. 1 October 2013 Web Server deployment-Netcraft

eBay and YAHOO! have been already active with Node.js software development [27]. In addition, Firebase is a company offering real-time services, supporting the WebSocket protocol and abstracting the idea of the backend [28]. Finally, two examples utilizing near real-time effect are Facebook's newsfeed [29] and Twitter [30] through HTTP polling techniques.

Summarizing the current challenges that modern web applications face, the high concurrency of web server architectures is of high importance. Actually, the most deployed web server is Apache, as shown in Fig. 1 and the server side language of choice for web applications is PHP [31]. They are both reported to have around 40 % market share. However, PHP was never meant to be used to write complex web-based applications [32]. Apache on the other hand is incapable of scaling linearly with the number of CPU cores [33]. In our study we'll use Node.js with Apache and Nginx in a try to highlight and prove that the popular combination of PHP/Apache isn't the optimal tool to address the future upsurge in traffic, and is rather inefficient in utilizing resources [34].

The cross-platform mobile application development is the second challenge. According to the International Data Corporation (IDC), Android and iOS combine for 92.3 % of all smartphone OS shipments in the first quarter of 2013 [35]. Android devices come in vastly different configurations, while iOS devices are more consistent. This is easily noticeable from Figs. 2 and 3, showing Application Programming Interface (API) and screen size fragmentation for each OS, respectively. As shown, the Android platform boasts a proliferation of different screen sizes [36]. Designing and coding layouts that work well across all these screens is hugely challenging, which does not apply in the case of iOS devices that maintain same physical screen sizes, while

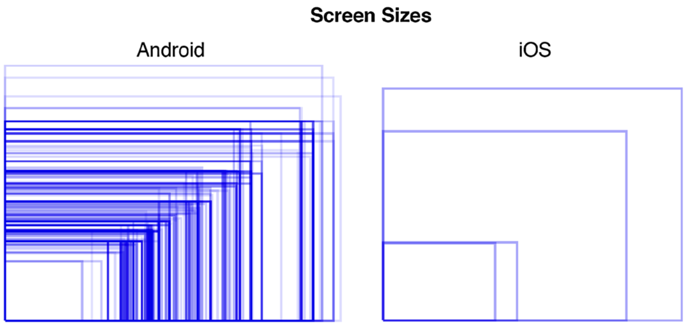


Fig. 2 Android and iOS screen size fragmentation

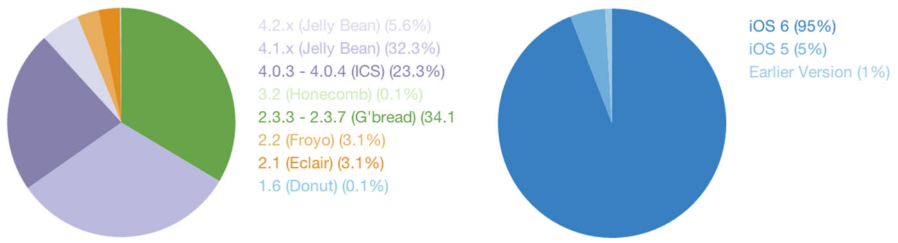


Fig. 3 Android and iOS API fragmentation

increasing their pixel density [36]. The hardware is a limiting factor when choosing an OS version, since older devices, especially low-end Android based devices, can’t handle the requirements of newer software releases [36]. As Java applets and flash aren’t supported on mobile devices, they can no longer be considered as cross platform solutions. The only viable solution left for cross-platform mobile application development is HTML5 and the other modern web technologies, associated with it; and this is the line we’ll also follow in our study.

Furthermore, real time web requires real-time communication. However, HTTP was not designed to support real-time, that is to say, the server is not able to send notifications to the client. With the emergence of JavaScript and AJAX (Asynchronous JavaScript and XML), polling techniques enabled near real-time communication. Depending on the polling interval there is a trade-off between latency and efficiency [21]. Another way to achieve real-time communication is the Flash XMLSocket implemented in the Adobe Flash platform [37]. While flash tends to be installed in most desktop browsers, Adobe has seized support in mobile devices. It impacts performance and can cause the application to hang for up to 3 s while connecting [38]. Recent studies overcome the real-time communication problem by using the WebSocket Protocol of IETF (Internet Engineering Task Force) and the WebRTC, i.e., the usage of a simple JavaScript API and HTML5 [40,41].

Last but not least, a big challenge arises from the current state of social applications. Social applications have been a huge part of Web 2.0, leading to the rise of social networking applications [39]. Social networking has been perhaps the most popular trend on the Internet over the past several years. Over time, society has started taking highly interactive platforms through which, individuals and communities, share, co-create, discuss and modify, user-generated content, for granted [39]. We believe that social networking, while great in many respects, doesn’t fulfill the fundamental human desire to be in the actual presence of other people. That’s where location and real time communication come in. We consider that proximity awareness and true real time interaction among users have the power to bridge digital and physical social interactivity. The aforementioned concepts are tightly coupled when addressing social application development and should be examined thoroughly together, as they can provide great synergy in efficiency, code reusability, as well as the total experience perceived by the user.

3 Design and implementation

This section is an extension of work originally reported by the authors in [40]. During the conference, we demonstrated a social web application built entirely with JavaScript, called Proximity. From the inception of Proximity our intention was to design a service that would be both scalable and maintainable throughout its entire development cycle. Another important consideration was the user experience. The responsiveness and efficiency of the application had to meet that of a native one, especially on mobile clients where resources such as CPU, memory and battery are limited. Session management, image manipulation, messaging, closeness-awareness, as well as audio and video communication are features considered as indispensable design parameters in modern social web applications and Proximity incorporates them [40].

3.1 Server architecture

As pre-mentioned, Proximity is an end-to-end JavaScript application. This is achieved by using Node.js at the server side part of the architecture. In JavaScript, functions are first class objects, which means that JavaScript functions are just a special type of object that can do all the things that regular objects can do, e.g., they can be passed as arguments to other functions in the same way as other values. A function may refer to free variables outside of its scope, and when that function is passed around, it encapsulates the necessary execution context with it, hence a closure is created. These features are the foundations that make events possible, rendering Node.js to follow the event-driven design approach and its core concept to be asynchronous, event-driven I/O [42]. That is to say, handlers (functions) are assigned to events (the I/O operations) and wait to be executed. Thus, an event-driven, service oriented architecture was chosen for the proposed system.

In order to evaluate whether it was feasible to implement a large-scale application with Node.js, such as Proximity, it's model had to be reflected to the application architecture. To ensure high availability HTTP and WebSocket servers were placed behind a reverse proxy. Their purpose was to transform connections to events and forward them to the Event Switch. The Event Switch is the heart of the application. Its sole purpose is to forward events from and to the correct services by following a set of rules (like an Ethernet switch). At the same time by using a mediator layer for the event switching, a single point of failure is introduced. To eliminate the single point of failure and ensure high scalability, multiple instances were spawned with the Node.js cluster module. Redis publish/subscribe mechanism was used as an interprocess message broker [44].

Another critical issue was the control flow of the asynchronous code. Traditionally that was achieved with extensive callback nesting. That can be addressed with the use of promises [45] combined with ECMA6 generators [46], which at the time being are available in the V11 of Node.js behind the—harmony flag. Thus, the extensive nesting was avoided ensuring code readability.

At this point each additional component was broken down into a separate service. Services can be connected with the Event Switch and listen to, or emit events. Based on user demand, additional resources can be allocated to subcomponents, thus ensuring consistent QoS for various scenarios of different workloads. Contrary to a monolithic approach it was a straightforward choice since it aids testing, code reusability and continuous integration [47]. Figure 4 depicts the high level system architecture with all the services combined together in Proximity [40].

Even though the HTTP protocol is stateless, state management is mandatory for security reasons since the server is responsible for delivering the appropriate content to each user. For these reasons, all control over states is delegated to a separate service, i.e., the Session Service in Fig. 4.

Location awareness can be implemented via the HTML5 Geolocation API. In order to build closeness-awareness for the users, a Nearest Neighbour Searching method had to be defined. One efficient approach is locality-sensitive hashing. We used a variant known as Geohash [43]. For this service, frequent small queries for updates require a very efficient database. Since the total size of the data, i.e. a user id and coordinates

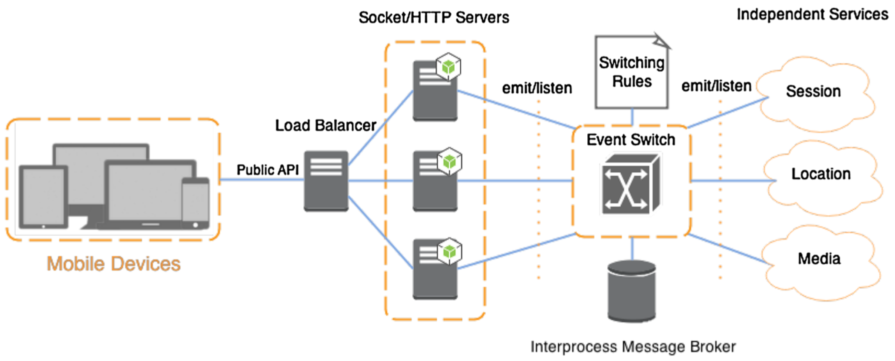


Fig. 4 High level system architecture

hash, is rather small, a NoSQL in-memory database, fits the requirements. The Redis key-value store offers sorted sets and can be partitioned providing both robustness and scalability. Since the whole process is intensive, it is designed as a separate service, i.e., the Location Service in Fig. 4, in order to avoid any performance limitations and offer the, critical to the user's experience, real-time performance.

Media manipulation is inherently characterized by bandwidth, memory and CPU intensive operations. It usually requires complex and application specific implementations, such as the usage of specialized libraries for image manipulation e.g., imageMagick [48] or graphicsMagic [49]. Equally important is the option to deploy a network file system or cloud storage for binary files. As shown in Fig. 4, in order to decouple relatively slow media manipulation from real time operations, the Media Service has also been separated.

Therefore, in order to implement such an architecture, which is heavily I/O bound and asynchronous by design, demanding in high concurrency support, efficient servers are required. An event driven approach seems rather better than a threaded one [50]. Node.js is a promising new technology; for this reason, we were triggered to investigate whether it fulfils the above requirements or not.

3.2 Client architecture

While developing the client-side, numerous challenges occurred as the code became exposed to many different execution environments. Each browser reacts differently to the DOM manipulation, the HTML5 APIs and to the processing of the CSS, leading to a differently rendered page. Furthermore, the same events are not fired consistently in all browsers. Those are just a few of the obstacles encountered. Fixing those issues one by one while developing becomes a tedious task and can lead to unmaintainable, tightly coupled code, severely damaging productivity by bug-tracing.

Consequently, the first step was to implement a feature detection layer, as shown in Fig. 5. During the initialization phase, the necessary features were verified and any API inconsistencies were normalized. The next step towards decoupling concerns, was the adoption of the MVC pattern as implemented in the AngularJS framework

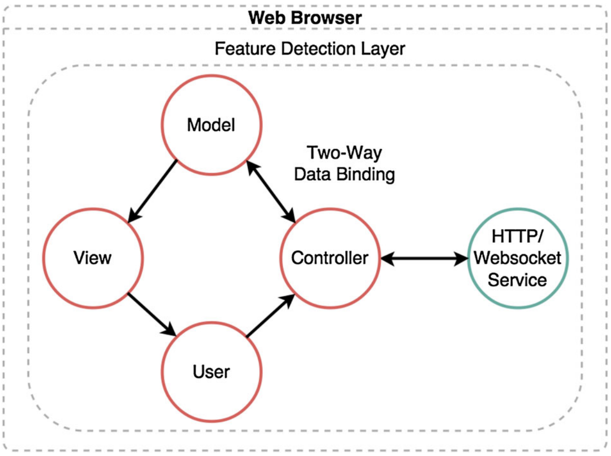


Fig. 5 Client architecture

[51]. AngularJS encourages dependency injection, that is, any hard coded objects that a function depends on, have to be explicitly declared in the constructor function. Combined with the prototypical inheritance of JavaScript, the dependencies can be dynamically changed during runtime, or be substituted with stub methods and mock objects, aiding testability.

The user's state along with the data pushed from the server, must remain synchronized with the view at all times. Instead of manually updating the view, AngularJS provides two-way data binding, minimizing the required code to a single variable assignment. In addition, socket.io provides a robust layer over the WebSocket API, adding the ability to fallback to XMLHttpRequest-polling, in environments where WebSocket is not supported.

WebRTC offers peer to peer real time media exchange, that is data, audio and video. A signaling server is required only during the initialization of the call, to allow the clients to discover each other. Even though webRTC is not yet standardized and is still under heavy development its features greatly surpassed any implementation difficulties. In order to address the screen-size fragmentation, responsive web design techniques were employed. In the web browser the rendering is traditionally done using its software engine. Although this may be adequate for plain web pages, any animations present inferior performance to those of native applications. That problem was addressed by using CSS3 hardware accelerated animations (e.g., in graphics processing units, GPU), which are compatible with all modern browsers [52]. The result was a smooth, native-like experience.

4 Performance evaluation

4.1 Experimental environment

The server was equipped with an Intel Core i7 920 @ 2.6GHz with 9GB DDR3 @ 1066 MHz and the client with Intel Core 2 Quad @ 2.4GHz with 3GB DDR2 @ 1066 MHz.

Both machines run Ubuntu server 13.10, kernel 3.8.0-31 generic. The server and the client shared a direct Gigabit Ethernet connection. The versions of the software tested were, Apache: 2.2.22, Nginx: 1.4.3, PHP: 5.4.6-1, Zend Engine: 2.4.0 and Node.js: 0.10.21.

The Linux kernel can't support multiple concurrent connections and high TCP traffic out of the box, thus we had to optimize the default settings of Linux kernel in order to take measurements for a high number of concurrent connections. We followed the Linux kernel documentation [53] and the guidelines published by IBM ITSO [54]. After series of verification tests we finally used the settings proposed in [33], since we also have a multicore testbed with 1 Gbit network connection.

In Apache, multi-processing modules (MPMs) are responsible for binding to network ports on the machine, accepting requests, and dispatching children to handle the requests [55]. The most deployed MPMs are prefork and worker modules, respectively [56]. Each installation of Apache can be compiled with only one MPM. The prefork module is the default one and comes preinstalled in Ubuntu server. With the prefork module compiled, Apache is non-threaded. This means that each process is a child that is either servicing a request or waiting for a request, respectively. Prefork is the only MPM fully compatible with the PHP module (mod_php) and non thread-safe PHP libraries [57]. On the other hand, by using the worker MPM, Apache becomes multi-process and multi-threaded, that is, each child process can handle multiple requests at a time [56]. Due to the nature of the benchmarks conducted, the server had to be optimised for higher concurrency. Apache documentation states that the worker MPM fits better for that purpose [58]. It's notable that the default configuration parameters, either explicitly declared in apache2.conf, or hardcoded in the server itself, caused significant performance issues. Therefore a thorough analysis of the official documentation was necessary.

Nginx is an open-source, HTTP server and reverse proxy, as well as an IMAP/POP3 proxy, designed to address the C10k problem with an asynchronous, event-driven architecture [59]. Workers are used to distribute load to multiple CPU cores, while Nginx does not ship with a PHP runtime. With the above configuration both servers could serve only static files and are unable to handle PHP execution. With this in mind, an external PHP runtime had to be deployed. We selected PHP-FPM (PHP-FastCGI Process Manager), since it is compatible with both Apache and Nginx [60]. Thus, the same PHP-FPM server was used for both Apache and Nginx. PHP-FPM is a multi-process implementation of the PHP FastCGI (Fast Common Gateway Interface) protocol [61]. It runs applications in processes isolated from the core Web server [40,41]. PHP-FPM was configured to use a dynamic pool of processes, depending on the load. To serve a PHP request both Apache and Nginx dispatch it to PHP-FPM through a local TCP socket and then forward the results to the request origin.

Node.js is a platform built on Chrome's JavaScript runtime [62] and uses an event-driven, non-blocking I/O model [63]. Since Node.js is single threaded, we implemented a web server with the bundled "cluster" module [64] so as to utilise all the available CPU cores. It is remarkable that any additional tuning was not necessary.

4.2 Testing framework

Figure 6 depicts the testing framework. As described above, each server spawns multiple child processes, ranging from the number of CPU cores (Node.js, Nginx) to a few thousand (Apache2, PHP5-FPM), with the later frequently killing idle processes in order to conserve random-access memory (RAM). That behavior posed two challenges, the first one was timing, as the processes had to be profiled under load and before the master process terminated them. The second challenge was maintaining a low central processing unit (CPU) and memory footprint, while profiling under the maximum system load.

Even though there are built in methods in each server to report its status, our goal was to decouple the profiling procedure from server specific APIs, in order to avoid making multiple implementations for each server. The tests had to be fully automated and configurable, in order to minimize human error and have reproducible results. To address the aforementioned issues a testing framework using Node.js as a wrapper and weighttp as the HTTP traffic generator was developed (Fig. 6). The weighttp library, is a benchmarking tool for web servers written in c. Additionally it supports multithreading to make good use of modern CPUs with multiple cores as well as asynchronous i/o for concurrent requests within a single thread [65].

4.2.1 Resource profiler

The resource profiler is deployed in the same machine with the server under test, as shown in Fig. 6. Its task is to take snapshots of the system's resources and send them back to a master node. This is achieved by resolving a process name to an array of PIDs (Process Identifiers) using the pgrep command and then parsing the /proc pseudo filesystem. The total CPU usage is calculated by sampling the time each core spent in user, system, nice, irq and idle states for 300 ms, using the Node.js os module.

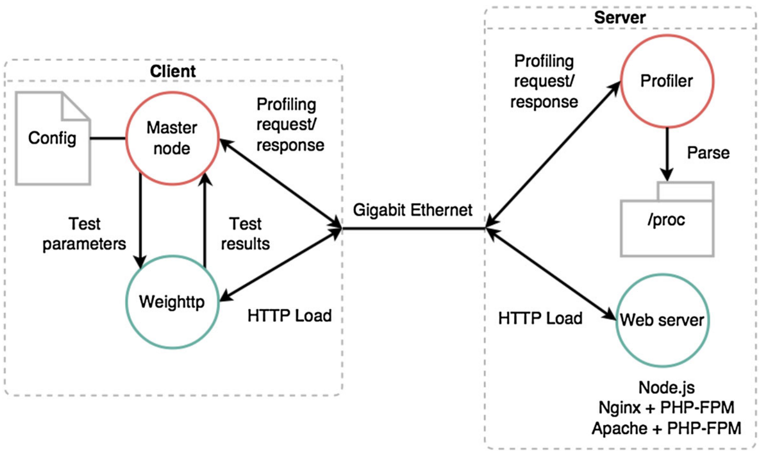
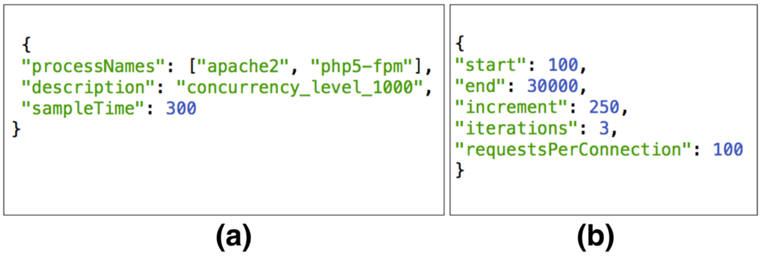


Fig. 6 Testing framework



```
{
"processNames": ["apache2", "php5-fpm"],
"description": "concurrency_level_1000",
"sampleTime": 300
}
```

```
{
"start": 100,
"end": 30000,
"increment": 250,
"iterations": 3,
"requestsPerConnection": 100
}
```

Fig. 7 **a** Sample profiling request, **b** sample benchmark parameters

The resource profiler communicates with the master node by exposing a JSON/HTTP API. The request body contains parameters that describe the snapshot to be taken. For instance, the sample request body in Fig. 7a would return the system wide CPU and memory and an array containing the same information per process level for all the PIDs of PHP and Apache.

4.2.2 Master node

The role of the master node (Fig. 6) is to generate server load, query the resource profiler and log performance analytics. Each benchmark is initialized by a configuration file, which contains elements that define the concurrency range and step, the iterations per concurrency level and the requests per connection. As an illustration, the settings in Fig. 7b, would take a total of 357 samples that is:

$$samples = Math.floor((end - start)/increment) \times iterations \tag{1}$$

with total requests for the entire session equal to:

$$\sum_{n=1}^{samples} [(requestsPerConnection) \times start \times n] \tag{2}$$

For each sample weighttp is spawned as a child process with different parameters. By continuously monitoring its output, a snapshot request is sent to the profiler at 70 % progress.

4.3 Test cases

Three different test scenarios are conducted in order to evaluate I/O, computational overhead and static file serving of the aforementioned technologies.

The first test regards an I/O test, testing Apache with PHP-FPM, Nginx with PHP-FPM and Node.js, respectively. Each server responds with a “hello world” string. The test is intentionally lightweight to avoid straining PHP and Javascript. The purpose of this test is to give an insight about the required resources and potential bottlenecks in performance due to misconfiguration, e.g., if all servers start to fail at the same concurrency level. Furthermore, this test provides a solid reference for the next tests.

In the second test we examine the computational overhead in the same configurations, i.e., Apache with PHP-FPM, Nginx with PHP-FPM and Node.js respectively, since web servers usually perform hashing operations, e.g., cookie encryption, images' conversions to/from base64, etc. With this in mind, the geohashing algorithm, previously used in Proximity [40], has been implemented in PHP and Javascript. In each request, a fixed set of geographical coordinates is encoded.

For the last test, the afore-mentioned three configurations respond with a 13.5 kB static file, given that it represents the average number of bytes per web page request [66]. During this test, PHP-FPM was not running.

The first and second tests were carried out with 100 requests per connection since it is a typical number of recent real life desktop web pages [66]. However, for the third one we decreased this parameter to 10 requests per connection so as to avoid saturating the TCP stack. For each test the concurrency level fluctuates from 100 to 30,000, with a step of 250. Each test is repeated three times and the average results were computed. Notice also that we use HTTP keep-alive for all connections.

At this point, someone could raise an objection, claiming that the afore-mentioned load traffic is rather light weight. First of all, the scope of the performance analysis is to evaluate the technologies in high concurrency rather than against big data (e.g., downloads, video streams, etc.). One of the reasons is the absence of similar research in the literature in that concurrency level. The concurrency factor becomes more and more important, since mobile devices are evolving into continuous connected web clients. Moreover, online social networks' users tend towards high concurrency rather than big data streams. For example, in [67], manual data traces taken from Facebook, result in 32 MB for 5036 requests, that is 6.51 kB per request, while the size of a single Google maps HTTP request per session presents a mean value of 11.61 kB. Nevertheless, in our tests, the size per HTTP request has been chosen to be representative to the average service call [66], i.e., 13.5 kB, which is also higher than the afore-mentioned values.

Another important consideration that led us to this decision is also our tesbed's restriction because of the 1Gbit network adapters. If we increased the data size, the link would saturate resulting in misleading measurements. Finally, granted that the data usually reside in a database system, it would make sense to use one too. Since the database is decoupled from the web server technology and in order to avoid benchmarking filesystems or numerous database drivers, we chose to eliminate that parameter so as the measurements remain focused on the web servers.

5 Results and discussion

5.1 I/O test

In the I/O test each server responds with a "hello world" string. According to Fig. 8, which depicts the requests per second against concurrent connections, Nginx is more than 2.5 times faster in I/O operations than Apache, while Node.js outperforms both. The fact that both Apache and Nginx are using PHP-FPM can be used to support the

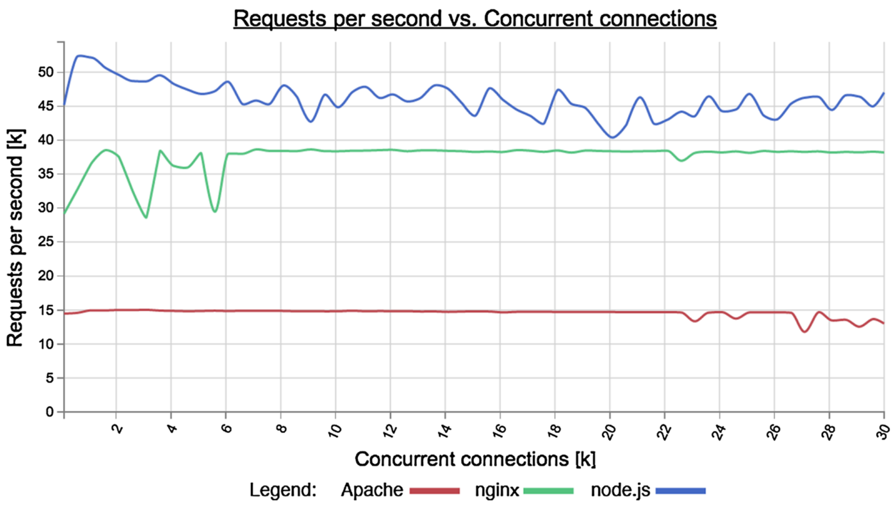


Fig. 8 I/O test, requests per second vs. concurrency

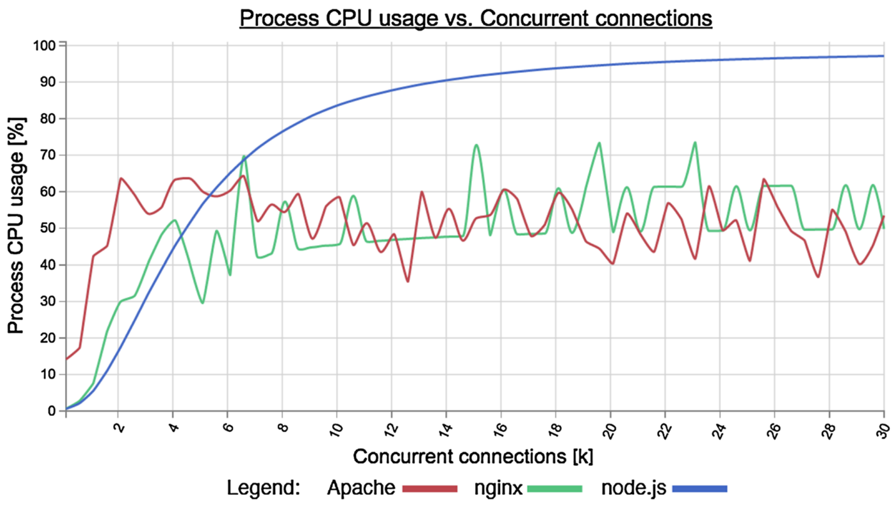


Fig. 9 I/O test, CPU usage vs. concurrency

argument that Apache is the limiting factor and not PHP. Based on CPU usage against concurrent connections (Fig. 9), both Apache and Nginx show a similar CPU usage pattern, even though Apache manages to handle fewer requests per second. Thus, Apache is more inefficient in handling I/O operations. Node.js demonstrates higher CPU utilization, by about 1.5 times more. As shown in memory against concurrent connections (Fig. 10), although the same resources were available to both Apache and Nginx, Apache failed to take advantage of them, due to its I/O bottleneck. On the other hand, Node.js shows excellent memory utilization for the whole test range.

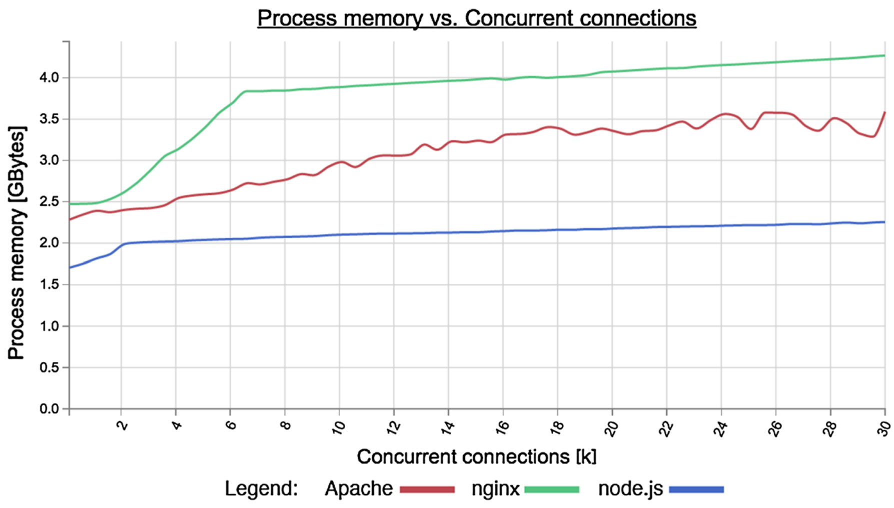


Fig. 10 I/O test, memory vs. concurrency

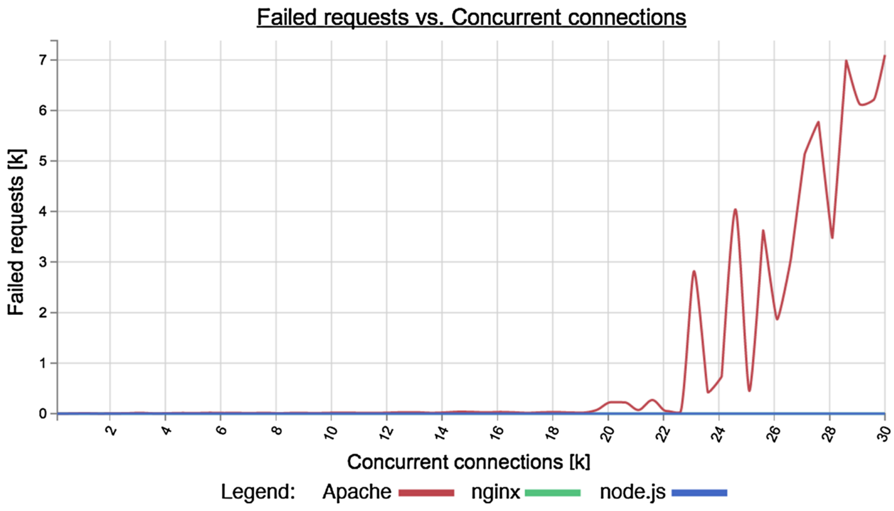


Fig. 11 I/O test, failed requests vs. concurrency

While both Nginx and Node.js had no failed requests, Apache started failing requests significantly right after the 19k concurrent connections threshold (Fig. 11).

5.2 Computational/hashing test

In the computational/hashing test a fixed set of geographical coordinates is encoded in each request using the geohashing algorithm. Based on the requests per second against concurrency (Fig. 12), Nginx appears to have lost most of the performance advantage demonstrated during the I/O test. The performance gap to Apache has been

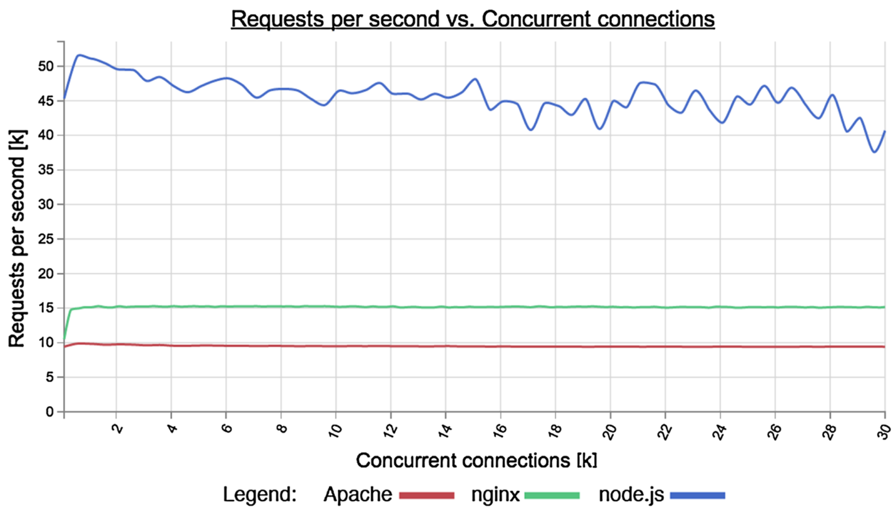


Fig. 12 Hashing test, requests per second vs. concurrency

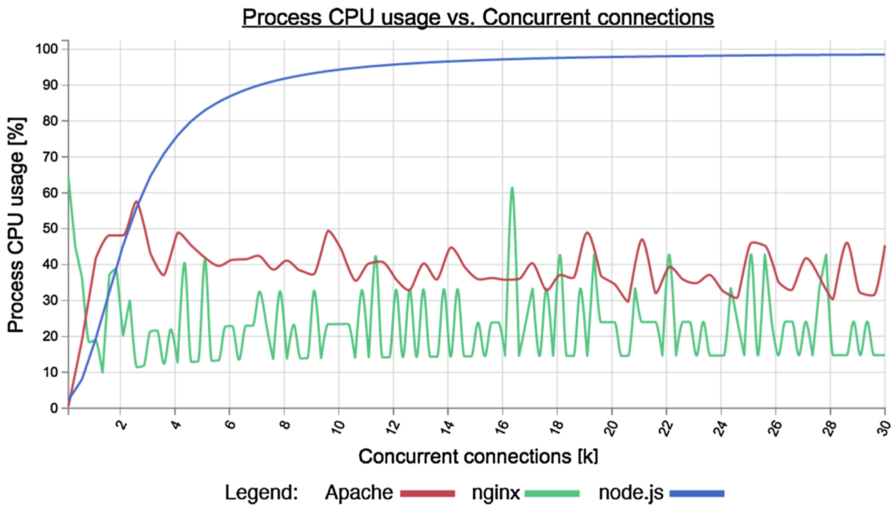


Fig. 13 Hashing test, CPU vs. concurrency

minimized, which can be attributed to computational inefficiencies of PHP. Node.js clearly outperforms PHP, by a magnitude of more than 2.5 times. It is notable that Node.js exhibits no significant performance losses when compared to the I/O test, meaning it is able to handle data processing more efficiently than PHP. As shown in CPU usage against concurrent connections (Fig. 13), Nginx demonstrates higher CPU efficiency than Apache. Node.js presents a steeper curve in reaching the maximum CPU, which is the only difference observed in comparison to the I/O test. Regarding memory against concurrent connections (Fig. 14), there are no significant differences observed by all examined technologies, in comparison to the respective I/O test. Almost

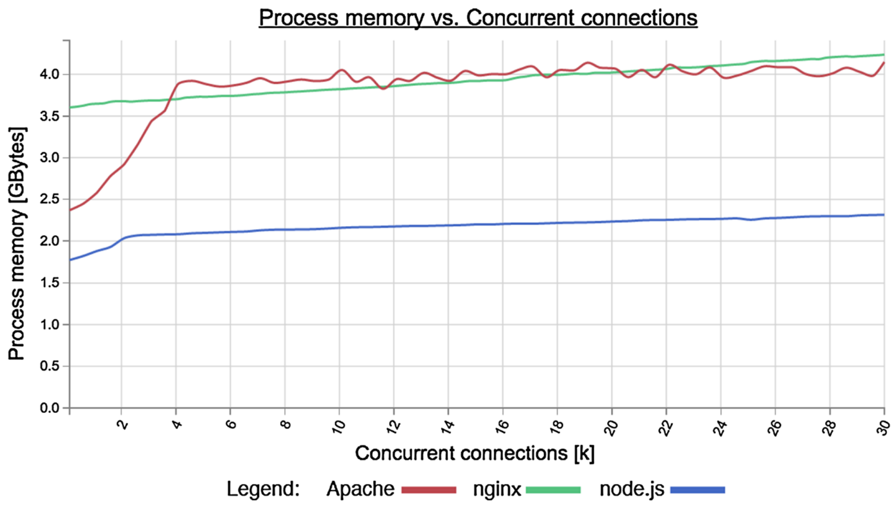


Fig. 14 Hashing test, memory vs. concurrency

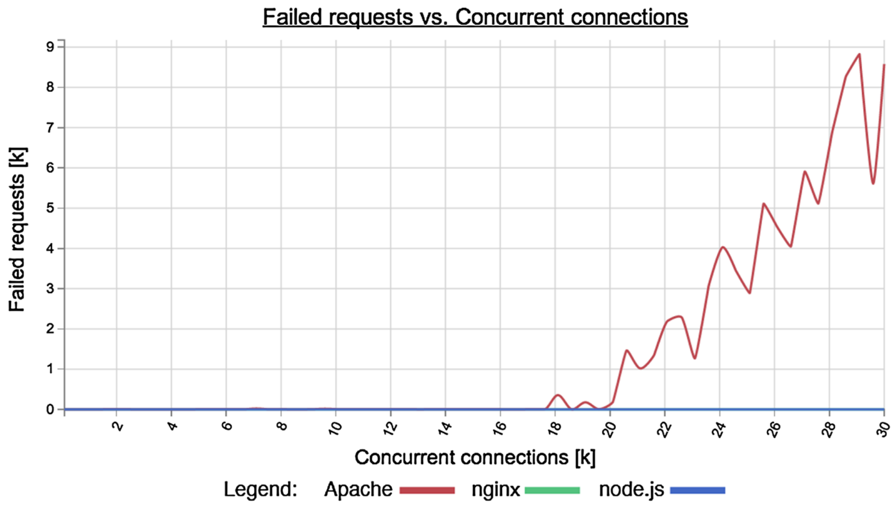


Fig. 15 Hashing test, failed requests vs. concurrency

the same pattern applies to the failed requests as well (Fig. 15), with the only difference that Apache started failing even for less concurrent connections, i.e., after the threshold of 17.5 k concurrent connections.

5.3 Static file test

In the static file test, a 13.5 kB file is served, as explained in Sect. 4. All technologies demonstrate similar I/O performance throughout the entire concurrency range (Fig.

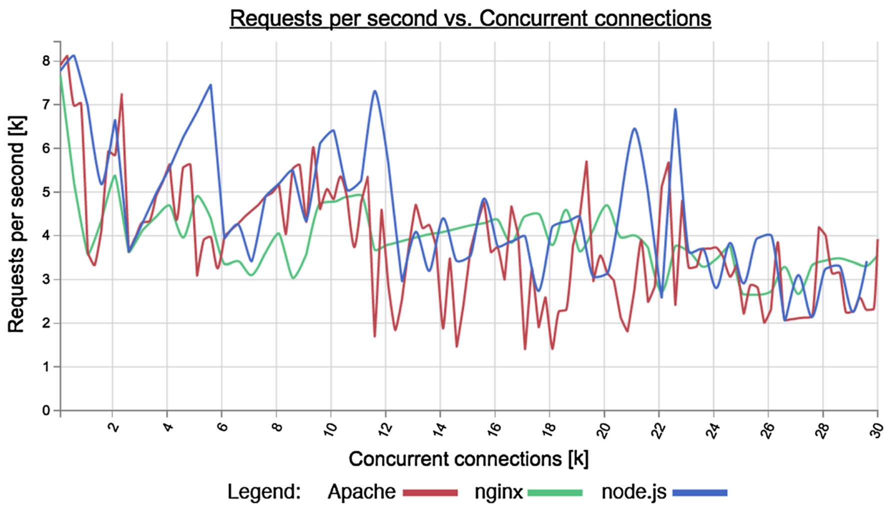


Fig. 16 Static file test, requests per second vs. concurrency

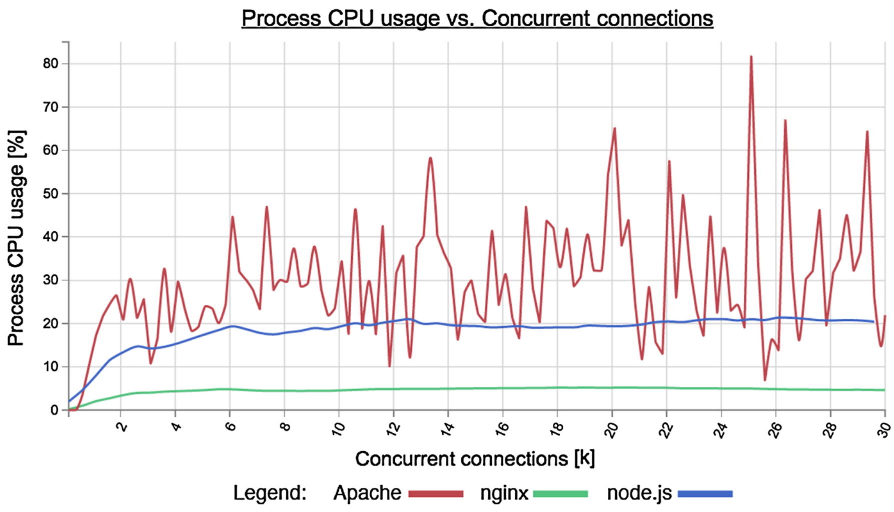


Fig. 17 Static file test, CPU vs. concurrency

16). Regarding the CPU usage, represented in Fig. 17, Nginx demonstrates superior efficiency, by about 4 times less resource consumption than Node.js and 5–6 times less than Apache. Both Node.js and Nginx demonstrate consistent CPU utilization for the entire concurrency range examined. This observation can be attributed to the event driven approach of both Node.js and Nginx, since Apache wastes resources to spawn and kill children processes. Memory utilization for serving static files' is the only case that Node.js falls behind both Apache and Nginx, while the latter shows superiority in memory utilization, as presented in Fig. 18. Regarding Fig. 19, both Apache and Nginx start failing requests, as early as 5 k concurrent connections. Node.js starts

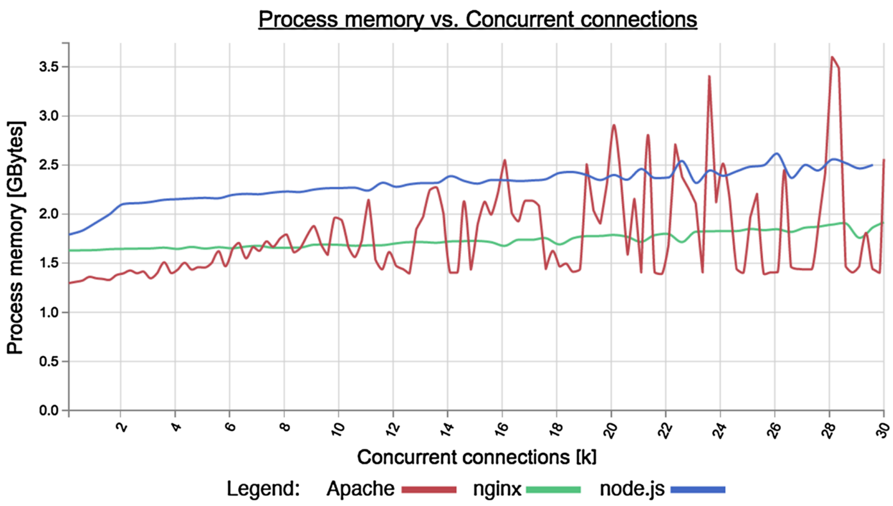


Fig. 18 Static file test, memory vs. concurrency

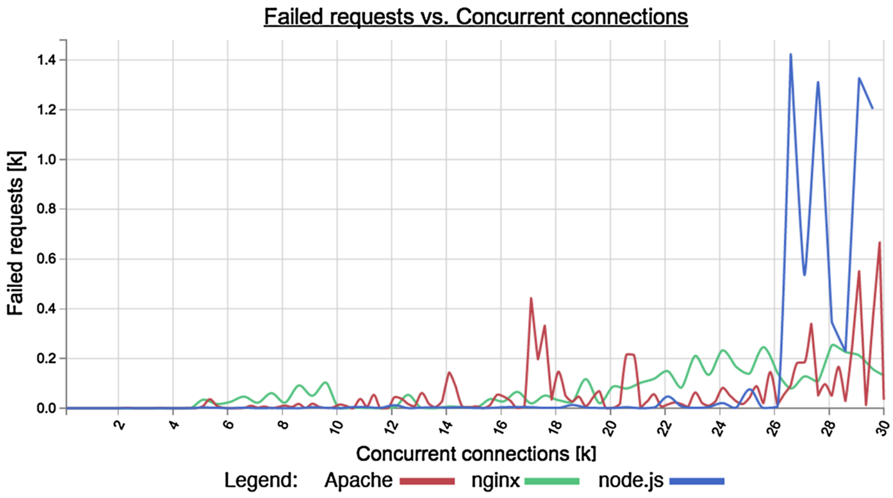


Fig. 19 Static file test, failed requests vs. concurrency

failing requests at 18 k concurrent connections, but climaxes after 26 k, where both Nginx and Apache demonstrate considerately fewer failed requests.

6 Conclusions

The performance evaluation conducted, shows that Apache is the least performing web server in all tested cases. It's rather CPU inefficient and memory demanding, without offering any significant performance gains. Nginx indisputably outperforms Apache in all tests. Furthermore, if Nginx acts as a static file server, it is even more efficient than Node.js, too. On the other hand, Node.js clearly outperforms PHP in computational performance, by being more memory efficient and by utilizing all available processing power.

We presented an architecture suitable for modern web application development, that's legacy free and aids to the client-server code integration. Additionally, we proposed solutions to frequently encountered problems of web developers, such as code structure for modularity, closeness-awareness, real-time information exchange and scalability. Furthermore we conducted a series of stress tests, on popular server side technologies. According to our performance evaluation, Node.js is lacking in serving static files using it's built in HTTP server, while Nginx performs great at this task. So, in order to address efficiency in such an architecture, it could be wise to place an Nginx server in-front and proxy static file requests, allowing the Node.js processes to only handle dynamic content. Such a configuration offers a better infrastructure in terms of efficiency and scalability, replacing the aged PHP/Apache stack. The rise in total network traffic and social web applications, are reasons for further research the infrastructure, as it impacts businesses and individuals alike. The state of the social web currently is monopolized, but more opportunities arise from propriety-free technologies and open-source tools, that allow developers to build novel services and applications. The emergence of WebRTC has the potential to be used by both businesses and individuals, to offer server-less real-time communications over the web, making server-side bandwidth expenses obsolete.

We have found that building cross platform applications based on web technologies, is both feasible and highly productive, especially when addressing stationary and mobile devices, as well as the fragmentation among them. So to conclude, the answer to the frequently asked question of web programmers: "Is end-to-end JavaScript a viable option for building modern web apps?", is yes, and highly advisable.

References

1. http://groups.google.co.uk/group/alt.hypertext/msg/06dad279804cb3ba. Accessed 22 Mar 2014
2. Lueg C, Fisher D (2003) From Usenet to CoWebs: interacting with social information spaces. Springer, Berlin
3. Facebook Reports Fourth Quarter 2013 and Full Year 2013 Results. http://investor.fb.com/releasedetail.cfm?ReleaseID=821954. Accessed 22 Mar 2014
4. Holzinger A, Treitler P, Slany W (2012) Making apps useable on multiple different mobile platforms: on interoperability for business application development on smartphones. In: Multidisciplinary Research and Practice for Information Systems. Lecture Notes in Computer Science, vol 7465, pp 176–189
5. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update (2012) 2017 http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html. Accessed 22 Mar 2014
6. Amatya S, Kurti A (2014) Cross-platform mobile development: challenges and opportunities. In: ICT Innovations 2013, Advances in Intelligent Systems and Computing, Vol 231. Springer, New York, pp 219–229
7. Ericsson Traffic and Market Report (2012) http://www.ericsson.com/res/docs/2012/traffic_and_market_report_june_2012.pdf. Accessed 22 Mar 2014
8. Juwel R, Hallberg J, Synnes K, Kristiansson J (2010) Harnessing the cloud for mobile social networking applications. Int J Grid High Perform Comput 2(2):1–11
9. RFC 6455#section-1.7, http://tools.ietf.org/html/rfc6455. Accessed 22 Mar 2014

10. Web Server Survey, Netcraft (2013) http://news.netcraft.com/archives/2013/10/02/october-2013-web-server-survey.html. Accessed 22 Mar 2014
11. PHP just grows and grows, Netcraft (2013) http://news.netcraft.com/archives/2013/01/31/php-just-grows-grows.html. Accessed 22 Mar 2014
12. Kappel G et al (2004) Web engineeringold wine in new bottles?. Springer, Berlin Heidelberg
13. W3C, HTML5.1, http://www.w3.org/TR/html51/. Accessed 22 Mar 2014
14. Juntunen A, Jalonen E, Luukkainen S (2013) HTML 5 in mobile devices-drivers and restraints. System Sciences (HICSS), 46th Hawaii International Conference on IEEE
15. PhoneGap Framework, http://phonegap.com/. Accessed 22 Mar 2014
16. Appcelerator Platform, http://www.appcelerator.com/. Accessed 22 Mar 2014
17. IBM Worklight, http://www-03.ibm.com/software/products/en/worklight/. Accessed 22 Mar 2014
18. RhoMobile Suite, http://www.motorolasolutions.com/US-EN/Business+Product+and+Services/Software+and+Applications/RhoMobile+Suite. Accessed 22 Mar 2014
19. White Paper by RapidValue Solutions: How to Choose the Right Architecture For Your Mobile Application (2012)
20. Mikowski M, Powell J (2013) Single Page Web Applications: JavaScript end-to-end, Manning Publications (Sep 27, 2013, ISBN: 1617290750)
21. Janne L (2013) On the development of real-time multi-user web applications. Tampereen teknillinen yliopisto. Julkaisu-Tampere University of Technology (Publication 1167)
22. Antero T, Mikkonen T (2011) The web as an application platform: the saga continues. Software Engineering and Advanced Applications (SEAA), 37th EUROMICRO Conference on. IEEE
23. Shuang K, Kai F (2013) Research on server push methods in web browser based instant messaging applications. J Softw 8(10):2644–2651
24. Spyros X, Xinogalos S (2013) A comparative analysis of cross-platform development approaches for mobile applications. Proceedings of the 6th Balkan Conference in Informatics. ACM
25. Smutny P (2012) Mobile development tools and cross-platform solutions. Carpathian Control Conference (ICCC), 2012 13th International IEEE
26. Henning H, Hanschke S, Majchrzak TA (2013) Evaluating cross-platform development approaches for mobile applications. Web information systems and technologies. Springer, Berlin Heidelberg
27. http://nodejs.org/industry/. Accessed 22 Mar 2014
28. http://www.firebase.com/. Accessed 22 Mar 2014
29. Tornado: Facebook's Real-Time Web Framework for Python, http://developers.facebook.com/blog/post/301/. Accessed 22 Mar 2014
30. Real-Time Delivery Architecture at Twitter, http://www.infoq.com/presentations/Real-Time-Delivery-Twitter. Accessed 22 Mar 2014
31. Web Server Survey, Netcraft (2013). http://news.netcraft.com/archives/2013/08/09/august-2013-web-server-survey.html. Accessed 22 Mar 2014
32. Simons P, Babel R (2001) FastCGI the forgotten treasure. ApacheCon Europe
33. Veal B, Foong A (2007) Performance scalability of a multi-core web server. Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems. ACM
34. The Zettabyte EraTrends and Analysis. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/VNI_Hyperconnectivity_WP.html. Accessed 22 Mar 2014
35. IDC, Press Release. http://www.idc.com/getdoc.jsp?containerId=prUS24108913. Accessed 22 Mar 2014
36. Android Fregmentation Visualized (2013) http://opensignal.com/reports/fragmentation-2013/fragmentation-2013. Accessed 22 Mar 2014
37. XMLSocket, AS3. http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/XMLSocket.html. Accessed 22 Mar 2014
38. Wang V, Salim F, Moskovits P (2013) Deployment considerations. The definitive guide to HTML5 WebSocket, chapter 8. Apress, pp 149–162
39. Kaplan Andreas M, Haenlein Michael (2010) Users of the world, unite! the challenges and opportunities of social media. Bus horizons 53(1):59–68
40. Chaniotis IK, Kyriakou KID, Tselikas ND (2013) Proximity: a real-time, location aware social web application built with Node.js and AngularJS. Mobile Web and Information Systems. Springer, Berlin Heidelberg, pp 292–295
41. Lundar J, Grnli TM, Ghinea G (2013) Performance evaluation of a modern web architecture. Int J Inform Technol Web Eng (IJITWE) 8(1):36–50

42. Tom H-C, Mike W (2012) “Node: up and running”, O’Reilly Media, ISBN: 978-1-4493-9858-3
43. http://geohash.org/. Accessed 22 Mar 2014
44. Redis documentation, http://redis.io/documentation. Accessed 22 Mar 2014
45. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. Accessed 22 Mar 2014
46. http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts. Accessed 22 Mar 2014
47. Mark Ethan Trostler (2013) “Testable JavaScript”, O’Reilly Media, ISBN: 978-1-4493-2339-4
48. ImageMagick, http://www.imagemagick.org. Accessed 22 Mar 2014
49. GraphicsMagick, http://www.graphicsmagick.org. Accessed 22 Mar 2014
50. Dabek F et al (2002) Event-driven programming for robust software. Proceedings of the 10th workshop on ACM SIGOPS European workshop. ACM
51. http://angularjs.org/. Accessed 22 Mar 2014
52. Stone P (2013) Pixel perfect timing attacks with HTML5. Context Information Security (White Paper)
53. /proc/sys/net/ipv4/* Variables, http://www.cyberciti.biz/files/linux-kernel/Documentation/networking/ip-sysctl.txt. Accessed 22 Mar 2014
54. IBM, International Technical Support Organisation Linux Performance and Tuning Guidelines (2007). http://www.redbooks.ibm.com/redpapers/pdfs/redp4285. Accessed 22 Mar 2014
55. Pache Multi-Processing Modules, http://httpd.apache.org/docs/2.2/mpm.html. Accessed 22 Mar 2014
56. Temme S (2007) Apache Performance Tuning Part One: Scaling Up
57. Apache MPM prefork. http://httpd.apache.org/docs/2.2/mod/prefork.html. Accessed 22 Mar 2014
58. Apache documentation. http://httpd.apache.org/docs/2.2/mpm.html. Accessed 22 Mar 2014
59. http://wiki.nginx.org/Main. Accessed 22 Mar 2014
60. http://php-fpm.org/. Accessed 22 Mar 2014
61. http://www.fastcgi.com/. Accessed 22 Mar 2014
62. https://code.google.com/p/v8/. Accessed 22 Mar 2014
63. http://nodejs.org/. Accessed 22 Mar 2014
64. http://nodejs.org/api/cluster.html. Accessed 22 Mar 2014
65. http://redmine.lighttpd.net/projects/weighttp/wiki. Accessed 22 Mar 2014
66. Johnson T, Seeling P (2013) Desktop and mobile web page comparison: characteristics, trends, and implications. arXiv preprint arXiv:1309.1792. Accessed 22 Mar 2014
67. Schneider F (2010) Analysis of new trends in the web from a network perspective. Ph.D thesis, TU Berlin