Evaluating Performance of Web Applications in (Cloud) Virtualized Environments

Fernando G. Tinetti[1] and Christian Rodríguez

III-LIDI, Fac. de Informática, UNLP, La Plata, Argentina
[1]Corresponding author, also at CIC Provincia de Bs. As., La Plata, Argentina
{fernando,car}@info.unlp.edu.ar

Abstract. Web applications usually involve a number of different software libraries and tools (usually referred to as *frameworks*) each carrying out specific task/s and generating the corresponding overhead. In this paper, we show how to evaluate and even find out several configuration performance characteristics by using virtualized environments which are now used in data centers and cloud environments. We use specific and simple web software architectures as *proof of concept*, and explain several experiments that show performance issues not always expected from a conceptual point of view. We also explain that adding software libraries and tools also generate performance analysis complexities. We also shown that as an application is shown to scale, the problems to identify performance details and bottlenecks also scale, and the performance analysis also requires deeper levels of details.

Keywords: Performance Monitorization, Web Applications Performance, IaC (Infrastructure as Code).

1 Introduction

As web applications and services have grown in functionality and scale, operation teams have been adopting different practices to achieve automation. Their main goal is to be up to date with development teams that have evolved much more quickly than operations. Besides, the operation teams necessarily focus on scalability problems that arises when their sites acquire popularity and the corresponding large requirements generate system failures and/or unacceptable response times. Failure in scaling up the computing resources (i.e. under-provisioning of resources) implies losing quality of service and, possibly, making a website, application or service, unavailable. Virtualization and cloud environments have provided successful tools and solutions for scaling up computing resources, but oversizing resources (i.e. over-provisioning of resources) also implies oversizing costs.

Elastic cloud computing environments claim to be appropriate for dynamically provisioning and de-provisioning resources. Furthermore, elastic cloud computing environments follow the "utility computing" model [12], and the its corresponding "pay-as-you-go" billing model, which turns to have the best cost/benefit relationship. However, depending on specific scenarios, it is hard to know how much and when scaling up or down, because it is almost completely web (site or application or service) dependent.

DevOps [9] [3] and SRE (Site Reliability Engineering) [10] practices emerged and established along the last ten years. As a consequence, IaC (Infrastructure as Code) [7] frameworks became the recommended way to simplify automation and bring resilience to infraestructures, being on-premises or cloud based data centers [13] [4]. Moreover, IaC adoption simplifies migrations from/to on-premises and cloud based datacenters and even build hybrid solutions.

Specifically related to scalability, applications must be implemented with some guidelines in mind. The Twelve Factor App Methodology [6] is a suitable starting point to adhere. However, application scalability depends on many domain-specific details, and there is not a single recipe to achieve acceptable/good results. We are going to divide the problem by services, each with different problems and options to scale

- Web application:
  - Stateless designs are scalable. State can be moved outside the application, using storage services like filesystem, databases or NoSql store engines as Memcached or Redis, among others.
  - Stateful applications can be scaled using sticky sessions. This approach is not recommended, but is preferred than no scalability.
  - The Twelve Factor App Methodology [6] enumerates best development practices to achieve scalability.
- Shared file system: not every shared file system can be scaled. Integrity is a must in some scenarios, but not for others. A shared file system can be a solution to grow, but availability becomes an issue depending on specific implementations (e.g. NFS: Network File System).
- Database: ACID (Atomicity, Consistency, Isolation, and Durability) database transaction properties is one of the main problems in a cluster of database engines. It depends on the DBMS engine to support a clustered environment or not. Some solutions involve multiple slaves and a master server. Only the master server carries out update queries, and slaves and master can be balanced to carry out read only queries. Some specific database load balancers can be used.

Web applications *tend to be scalable*, but some problems emerge when state is maintained outside the application. In this case, the whole software architecture relies in a third-party service that is not easy to implement in a scalable way as is the case of databases or shared file systems. In this paper, we are going to analyze web server configurations scalability considering a dynamic application server behind a reverse proxy to understand where the bottlenecks are, and which configuration do its best considering scalability. More specifically, we will try to provide some insight for the analysis of popular web applications, focusing in how a reverse proxy works and identifying what kind of content is served, identifying the requirements slowly served which eventually make the whole application unavailable to end users.

The rest of the paper is organized as follows. We define some important terminology and the underlying problems to which some terms are referring to. In Section 3 we show a simple experiment defined to show that some performance problems are found in details which are sometimes hidden or non-properly identified. Section 4 focuses horizontal web application scalability and performance evaluation. Finally, in Section 5 we outline some conclusions from the work presented in this work as well as our guidelines for the future work in this area.

2 Defining Terms and Problems

There are plenty of development languages, software libraries, and frameworks combined/configured in software architectures for building web applications. And web application architectures have been evolved and redefined, from a monolithic or single tiered web application, to the popular three tiers architecture, service based, or even microservices patterns architectures. Each architecture can be implemented by the number of available languages. Moreover, there are frameworks to easily develop applications following standards and so called *best practices*.

We will define some terms to better understand the context as well as specific details of our work. Web applications provide content that can be served:

- **Statically**: this content usually does not suffer any delays when served, and most of the times can be cacheable.
- **Dynamically**: depending on the requirement, the corresponding reply include delays of computing requirements and third party services (e.g. web services or database queries) used to build a response. Dynamically defined replies do not always can be cached.

Serving dynamic content requires more processing (and its corresponding delay time from the clients' point of view) and required resources than serving static content. At this point it is necessary to define and differentiate from one another application servers and (static content) web servers:

- An application server generates dynamic content as well as services related to a web application. Usually, (web) application servers are more complex than static web servers, and the "extra" complexity usually makes application servers slower than static web servers. As more time is required to reply requests, there are stronger limits to concurrency.
- Web servers commonly provide static content as assets, files, images, etc., and in some cases they are used to implement reverse proxies and even content delivery networks [11].

In this context, overcoming a limit for concurrency is directly related to scaling [1], i.e. the way in which more resources are available for processing, and in this specific case: for the application servers [2] [8]. Vertical scaling is related to hardware, i. e. resources are provided almost directly by the available hardware. Horizontal scaling, on the other hand, is usually related to services, provided by servers on hardware. Thus, horizontal scaling is usually cheaper in terms of required hardware and amount of work/configuration. Also, horizontal scaling is specially fitted to cloud environments, where the hardware is virtualized and (new and/or more) services are relatively easy to be deployed.

Application servers are naturally related to programming languages because programming is required by each specific application. Depending on each development/programming language, there are different application servers:

- Java application servers: Glassfish, JBoss EE, Apache Tomcat, etc.
- PHP: Apache with PHP module imposes the Multi Processing Module, a non-threaded, pre-forking web server. Alternatively, PHP-FPM can be used as application server, and use a web server (e.g. Apache or nginx) that reverse proxies' HTTP requests using the FastCGI protocol.
- Ruby: Unicorn, Puma, and Passenger are the most popular ruby application servers. Each one depends on a web server, usually nginx is the best companion to each case.
- Python: Gunicorn and Daphne are popular python application servers. They implement WSGI (Web Server Gateway Interface) and ASGI (Asynchronous Server Gateway Interface) protocols to communicate with reverse proxies in front of them. Nginx is generally the chosen reverse proxy.

The above list enumerates several of the most popular web development languages and their corresponding application servers. In this work, we will use Apache with PHP module to emulate an application. We will handle the PHP application for experimentation purposes, e.g. controlling/defining its response time. Our approach will be to include a delay time, emulating a third-party time service, and statically (e.g. as a parameter) set in a specific amount of time.

We will make several experiments by building different service and software architectures. In each experiment, we will simulate traffic/a pattern of requirements for the analysis of results by recording reply time and/or errors (e.g. timeouts). We will take advantage of IaC tools, i.e. the same tools currently used for maintaining on production websites and applications.

From an operations point of view, we have many alternatives for implementing our experiments: virtualization based on Virtualbox, VMWare, Hyper-V or Xen, or even cloud provided PaaS (Platform as a Service). Although all of them are suitable implementation tools, their use implies to develop shell scripts, playbooks, and/or receipts in order to take advantage of idempotent framework custom scripts such as Ansible or Chef. Instead, we are going to implement our experiments using docker and docker-compose tools which will allow (easy) versioning, replication, and scalability.

3 Simple Experiments: Where are the Problems?

We will use a PHP script specifically designed to run in a fixed amount of time by sleeping the script by two seconds before generating the reply to the corresponding request. Besides, we configure the apache server for handling only two concurrent requests. The combination of the apache web server configuration and the PHP script request handling imposes some restrictions on how this web architecture works. With this fixed time and resources restrictions, we expect the following behavior: a) Only two requests can be served concurrently, b) Each request will have a delay of 2 seconds, and c) We expect to serve 60 requests per minute without any errors.

We test this architecture using the Apache Benchmark tool, for different concurrent requests configurations and a total of 600 requests. The experiment for 600 requests with a concurrency level of 3 requests is made by executing

```
ab -l -c3 -n 600 http://localhost:8080/
```

And the following summary is obtained:

```
Concurrency Level:        3
Time taken for tests:      600.459 seconds
Complete requests:       600
Failed requests:             0
Total transferred:           148200 bytes
HTML transferred:         31200 bytes
Requests per second:   1.00 [#/sec] (mean)
Time per request:          3002.296 [ms] (mean)
Time per request:          1000.765 [ms] (mean, across all concurrent
requests)
Transfer rate:                 0.24 [Kbytes/sec] received
Connection Times (ms)
                       min     mean[+/-sd]  median   max
Connect:            0          0     0.2           0          3
Processing:  2001   2999 1000.0    2007    4005
Waiting:              0    998 1000.1          6     2004
Total:            2001  2999   999.9    2007     4005
```

As another example, the experiment for 600 requests with a concurrency level of 10 requests is made by executing

```
ab -l -c10 -n 600 http://localhost:8080/
```

And the following summary is obtained:

```
Concurrency Level:        10
Complete requests:        600
Failed requests:              0
Total transferred:            148200 bytes
HTML transferred:          31200 bytes
Requests per second:    1.00 [#/sec] (mean)
Time per request:           10007.641 [ms] (mean)
Time per request:           1000.764 [ms] (mean, across all concurrent
requests)
Transfer rate:                  0.24 [Kbytes/sec] received
Connection Times (ms)
```

```
                       min  mean[+/-sd] median   max
Connect:            0       0      0.2      0            2
Processing:   2002  9941  629.9  10007   10012
Waiting:             1   7940  629.9    8007     8012
Total:            2002   9941  629.8  10007   10012
```

Fig. 1 shows the results for different number of concurrent requests, and as concurrency grows, the clients experience a slower response because of the limitation of two simultaneous clients configured at the web server.

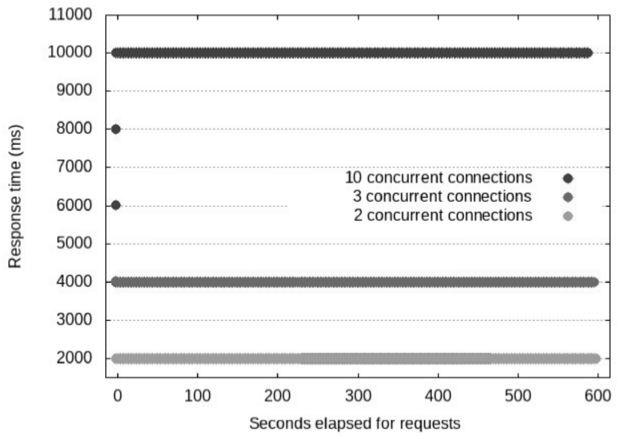

Fig. 1. Simple Server configuration, 2 concurrent requests limit set at the Apache web server.

From the point of view of the PHP server, more concurrent requests should imply less average reply time (within the limits of PHP server computer resources such as RAM size). For 2 concurrent requests, server throughput is only one request per second, as shown in Fig. 1, because each request will be replied in 2 seconds, and they are concurrent. In the example, when more than 2 concurrent requests are received, only the first two are handled as expected, all the other requests are queued at the Apache web server (not the PHP server). As more concurrent requests are made, the average reply time will proportionally grow, because the (low) fixed number of concurrent requests handled by the Apache server. Clearly, the problem is not the PHP server (maybe the *traditionally* first “candidate” for optimization and/or performance analysis), but the Apache web server configuration.

4 Looking at More Complex/*Real* Problems

As explained in the previous section, limits in the number of concurrent connections configured in the web server may result in increasing response time once those limits are exceeded. In general, each request implies to acquire and use an amount of resources by the application server, like memory, CPU, or even IO. This resource consumption is the main factor to consider when calculating how many requests to serve concurrently. When resources usage get near the physical limits, we must approach upward scalability. At this point is when horizontal scaling is usually adopted as a general solution. Experimentally, it is possible to horizontally scale up the above architecture and test it with the same tools to compare results. The scale down problem is rather analogous from the experimental point of view.

We have set an application server which can be horizontally scaled by means of a standard load balancer, as schematically shown in Fig. 2. It is worth noting that using a load balancer for horizontal scaling is usually easy: install the corresponding tool/service and configuring a few parameters, such as an *upstream timeout*, the maximum waiting time for a backend server reply.

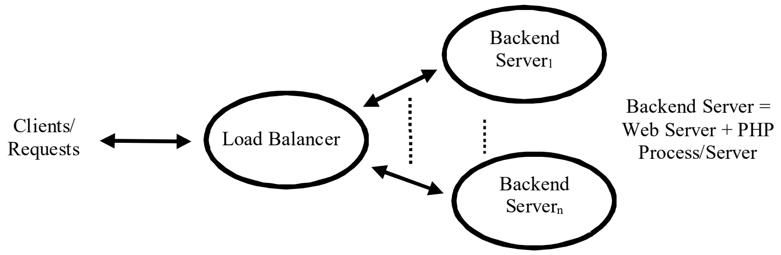

Fig. 2. Horizontal Scale Up of Web Application Server.

Fig. 3 shows the results obtained with a load balancer configuration timeout of 15 seconds from each upstream (Apache + PHP server, or *backend server*) to obtain a response. We are testing the worst scenario from the previous experiments, i.e. that with 10 concurrent requests, and scale up the backend servers from one backend server to two and ten backend servers.

As Fig. 3 shows, using only one backend server there is no performance difference from that shown of Fig 1. For two instances of the backend, the requests are served at about half the time in average: 50% being served in 4 seconds, and the other 50% served in 6 seconds. With ten backend server instances, the 600 requests are served in approximately 2 minutes, being all requests replied in 2 seconds in average, the minimum time each request would be served.

Horizontal scale up is easily implemented with a load balancer, as shown in the previous experiment. However, adding a load balancer also adds new details, including configuration parameters and monitorization data. More specifically, the load balancer is defined with a timeout of 15 seconds for each upstream backend server, as detailed above. As this timeout is lowered, more non-200 status codes HTTP responses will be generated directly from the load balancer to the clients. In the experiments shown in Fig. 2 above all the requests had a 200 HTTP response code, i.e. every request was successfully replied. Reducing the timeout below 10 seconds, the load balancer replies a fraction of the requests with non-200 HTTP response codes, more specifically: 504 and 499 response codes. The specific fraction replied with error depend of the specific timeout set at the load balancer, as expected. The load balancer timeout value must be fine-tuned, knowing how much time each upstream request will last and the number of concurrent requests each upstream would successfully handle to achieve a proper web application response behavior.

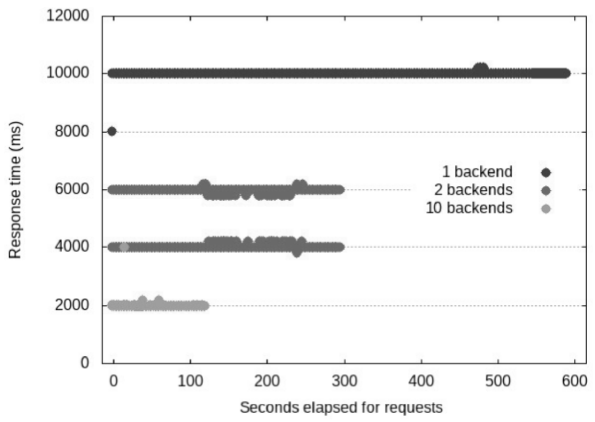

Fig. 3. Horizontal Scale Up of Web Application Server: load balancer with multiple backend servers, each handling up to 10 concurrent requests/connections.

The full software configuration and running experiments of the previous section as well as the ones in this section can be found in a repository at [14]. Even when a lot of "what-if" questions can be analyzed by statistic methodologies and queueing theory in particular [5], having an experimentation environment provides several advantages. In the extreme case the real application can be used along with real data collected from on production site/s. Besides, some simple changes in the environment would provide direct results, just as those described before: reducing the load balancer upstream timeout results in a proportionally large number of non-200 HTTP response codes for the corresponding HTTP requests.

5 Conclusions and Further Work

We have shown in simple scenarios and experiments several details of web application server analysis of performance and scalability. We also have set our experiments with easy replication and control version by means of currently IaC (Infrastructure as Code) tools. More specifically, we have shown that performance reply and also timeout errors may be related to standard tools such as web servers and load balancers instead of the application specific code/service (the PHP code in our example). We have also shown the effects of horizontal scale up the application server and the corresponding performance enhancement. Even when we have focused the problem at the load balancer, in a real application there are three sources of delay time and possible problems for replying each client request:

- The load balancer process, which depends on its configuration, e.g. the timeout defined for the upstream backend server/s in the example we have given above, and the number of upstream backend server processes.
- The web server, part of the backend server process, which also depends on its configuration, e.g. the number of allowed concurrent requests in the experiments/examples given above.
- The application itself, which in this case is a simple PHP process with a predefined delay, but maybe as complex as the problem/business requires, including specific source code and third-party services such as databases.

All the experiments and details explained above are extremely important in a real web application, because it is almost impossible to determine response times with a new release of an application. At this point, statistics tools, monitoring, and observability must be driving the limits as part of the testing phase of a new web application. And, once in production, almost the same monitoring process should be carried out for at least aiding the runtime performance evaluation process. Even when we have shown scale up examples, similar tests/experiments can be defined and carried out for scale down resources in order to avoid over-provisioning of resources and the corresponding extra costs in cloud environments.

The tools and web application architectures are widely available and used in current in-production web applications. Furthermore, deploying new web application versions without the proper performance experimentation usually ends up in failure or over-provisioning of resources and the corresponding extra cost in either cloud services or hardware in a data center.

One of the most interesting work to be carried out as a next step is focused in defining a methodology for monitorization and triggering process for automating at least some scaling (up and down) processes. Current alarm triggers are usually defined per monitorization-tool and/or OS resource usage. Our plan is at least verifying their usefulness and reduce the amount of false alarms (positives and/or negatives) by taking into account the combination of different tools and monitorization data collected at runtime. The minimum result, in this context, would be to be able to verify the (specific and sometimes complex) scaling services provided by public clouds.

References

1. Abbott, M. L., Fisher, M. T., The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, 2nd Ed., Addison-Wesley Professional, ISBN 0134032802, 2016.
2. Anandhi, R., Chitra, K. “A Challenge in Improving the Consistency of Transactions in Cloud Databases - Scalability”, International Journal of Computer Applications (0975 – 8887), Vol. 52– No.2, August 2012.
3. Davis, J., Daniels R., Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale, O'Reilly Media, ISBN 1491926309, 2016.
4. Glitten. S., “Cloud vs. on-premises: Finding the right balance”, Computerworld, May 2017.

5. Harchol-Balter, M., Performance Modeling and Design of Computer Systems: Queueing Theory in Action, Cambridge University Press, ISBN 1107027500, 2013.
6. Hoffman, K., Beyond the Twelve-Factor App O'Reilly Media, Inc., ISBN: 9781492042631, 2016.
7. Jourdan, S., Pomes, P., Infrastructure as Code (IAC) Cookbook Paperback – February 17, 2017.
8. Michael, M., Moreira, J. E., Shiloach, D., Wisniewski, R. W., "Scale-up x Scale-out: A Case Study using Nutch/Lucene", 2007 IEEE International Parallel and Distributed Processing Symposium, doi:10.1109/IPDPS.2007.370631, 2007.
9. Morris, K., Infrastructure as Code: Managing Servers in the Cloud, O'Reilly Media, ISBN 1491924357, 2016.
10. Murphy, N. R., Beyer B., Jones, C., Petoff, J., Site Reliability Engineering: How Google Runs Production Systems, O'Reilly Media, ISBN 149192912X, 2016.
11. Robinson, D., Content Delivery Networks: Fundamentals, Design, and Evolution, Wiley, ISBN 1119249872, 2017.
12. Sill, A., Spillner, J., (Eds.), 2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC), Zurich, Switzerland, IEEE CPS, ISBN 978-1-5386-5504-7, Dec. 2018.
13. Hewlett Packard Enterprise, On-Premises Data Centers vs. Cloud Computing, https://www.hpe.com/us/en/what-is/on-premises-vs-cloud.html, last accessed 2019/14/3.
14. https://github.com/chrodriguez/php-scale-probe, "Proof of Concept on PHP Scaling", in Spanish, "Prueba de concepto sobre el escalado con PHP", 2019.