Controlling the Elasticity of Web Applications on Cloud Computing - Extended Version
Michel Albonico, Jean-Marie Mottu, Gerson Sunyé

To cite this version:
Michel Albonico, Jean-Marie Mottu, Gerson Sunyé. Controlling the Elasticity of Web Applications on Cloud Computing - Extended Version. 2016. <hal-01317715>

HAL Id: hal-01317715
https://hal.archives-ouvertes.fr/hal-01317715
Submitted on 18 May 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright
Controlling the Elasticity of Web Applications on Cloud Computing

Extended Version

Michel Albonico
AtlanMod team (Inria, Mines Nantes, Lina)
UTFPR, Brazil and Mines Nantes, Brazil
michelalbonico@utfpr.edu.br

Jean-Marie Mottu
AtlanMod team (Inria, Mines Nantes, Lina)
University of Nantes
jean-marie.mottu@inria.fr

Gerson Sunyé
AtlanMod team (Inria, Mines Nantes, Lina)
University of Nantes
gerson.sunye@univ-nantes.fr

ABSTRACT

Web applications are often exposed to unpredictable workloads, which makes computing resource management difficult. The resource may be overused when the workload is high and underused when the workload is low. A solution to deal with unpredictable workloads is to migrate Web applications to cloud computing infrastructures, where the resource is varied according to demand, i.e., elasticity. With elasticity, all the resource variations happen during the Web application runtime. To deal with this, the Web application, and its service layers must behave in an elastic manner, which comprises adaptation tasks. These tasks may introduce functional and non-functional errors into the Web application. To find these errors, we must test the Web application when the adaptation tasks are performed, during the resource variations. Some tests may require a specific sequence of resource variations, which are difficult to be achieved without controllability. Therefore, in this paper, we propose an approach that controls the required resource variations. We validated our approach by conducting several experiments on Amazon EC2 cloud infrastructures. In these experiments, we successfully lead the Web application through the required resource variations.

CCS Concepts

- Networks → Cloud computing
- Information systems → Web applications
- Software and its engineering → Organizing principles for web applications

Keywords

Web application, elasticity, elasticity states, controllability, cloud computing.

1. INTRODUCTION

Web application workloads vary in an unpredictable way. This variation may lead to both, resource overuse or underuse, when the workload is high or low. A way to improve the workload variations processing is by migrating Web applications to cloud computing infrastructures, which provide elasticity, i.e., the resource is varied according to demand.

Elastic infrastructures vary the resource (allocate, and deallocate) at application runtime. To deal with these variations, the Web application and its service layers (database, application container, etc.) must behave in an elastic manner. This comprises adaptation tasks [7], which may be represented by the operations listed by Bersani et al. [3]: component synchronization, registration, and data replication.

Adaptation tasks may introduce functional, and non-functional errors into the Web application. For instance, a functional error may happen due to an inconsistency on data replication, which may result in an unexpected Web page content. Non-functional errors, such as rejected requests and long response times, may result from a delay in a new resource allocation (e.g., Web server), or a problem in reconfiguring the load balancer. To find these errors, the application must be tested when the adaptation tasks are performed. Otherwise, they do not occur, and cannot be discovered.

Testing Web applications through elasticity may be impracticable without controllability. Each test may require specific elastic behavior, which implies in an exact sequence of resource variations. This sequence may take too long to happen, or not happen when it is not controlled. Therefore, it is necessary an approach that ensures the reproducibility of resource variations according to the test needs.

The reproducibility of resource variations may be controlled by exposing the Web application to proper workload variations. Regarding the generation of Web application workload, some work addressed this topic for the last years [14, 13, 10, 1], though none of these work considers elasticity. Other work [9, 8] vary the workload expecting by resource variations. However, they do not control the resource variations.

In this paper, we propose an approach that controls the generation of workload variations according to the required resource variations. We also propose an implementation of this approach, which executes automatically. This implementation is executed in three phases: profiling, workload calculation and application leading. In the profiling phase, the application is exposed to a workload pattern, then we gather the resource usage profile related to this workload pattern. In the workload calculation phase, we estimate (based on the resource usage profile) the workload intensities necessary to request the required resource variations. Then, in the application leading phase, we expose the application to each one of the calculated workload intensities until the resource variations are completed.

We validate our approach conducting experiments on
In these experiments, we use the bulletin-board benchmark application RUBBoS\(^2\) to represent the Web application class, and its own benchmark tool to generate the workload. Both, the Web application and the benchmark tool, are realistic and previously used in literature\(^3\). In these experiments, our approach is able to lead the Web applications according to required resource variations. It allows to control a sequence of resource variations deterministically, by allocating or releasing virtual machines (the resource we considered).

This approach is the first step in direction to test Web applications through elasticity. In a future work, we will address the aspects of the test, such as test case, test strategy and test oracle.

The paper is organized as follows. In the next section, we present the major aspects of elastic Web applications. Section 3 introduces our approach. The experiments and their results are described in Section 4. Section 5 discusses related work. Finally, Section 6 concludes.

2. BACKGROUND

2.1 Cloud Computing Elasticity Behavior

Different authors\(^1\)\(^2\)\(^7\)\(^3\) have a common definition for cloud computing elasticity: it is the ability of a cloud infrastructure modifying its resource configuration as quickly as possible, according to application demand.

Figure 1 represents the typical behavior of elastic cloud computing applications. In this figure, the resource demand (continuous line) varies over time, at first increasing from 0 to 1.5 (demanding 50% more resources than the current allocated resources) and then decreasing to 0.

Figure 1: Representation of cloud computing elasticity.

When the resource demand exceeds the scale-out threshold, and remains higher during the scale-out reaction time, the cloud elasticity mechanisms release a resource. As soon as the scale-in begins, the threshold values are updated and the resource is no more available. Nonetheless, the infrastructure needs a scale-in time to release the resource.

2.2 Cloud Computing Elasticity States

Fluctuations in workload pressures the Web application. When this application is deployed on a cloud infrastructure, those fluctuations lead to resource variation (elasticity). We classify the resource variation according to resource status, which we call elasticity states. Figure 2 illustrates the elasticity states that Web applications running on a cloud infrastructure are exposed.

![Elasticity States](image)

Figure 2: Elasticity states.

At the beginning the application is exposed to the ready state, when the resource configuration is steady. Then, if the application is exposed for a certain time (scale-out reaction time) to a pressure that breaches the scale-out threshold, the cloud elasticity mechanisms start adding a new resource. At this point, the application is exposed to the scaling-out state: period while the resource is added.

After a scaling-out, the application returns to the ready state. When the application is at the scaling-out state, if it is exposed for a certain time (scale-in reaction time) to a pressure that breaches the scale-in threshold, the cloud elasticity mechanisms start releasing some resource. This puts the application in the scaling-in state: period while the resource is released. After that, the application returns to ready state again.

2.3 Workload Variation

In the literature, several work related to Web application benchmark\(^4\)\(^1\)\(^3\)\(^10\)\(^8\) conduct experiments by increasing the workload (intensity, number of clients, etc.) gradually over time. In this section, we discuss the effects on requesting elasticity states using this kind of workload fluctuation.

We consider a gradual variation where the workload increases in a rate that requests elasticity changes as soon as possible. At this rate, which we call *hurried rate*, a new scale-out threshold is breached immediately after the previous scaling-out state is completed.

Figure 3 (a) illustrates a *hurried rate*. The workload increases the resource demand in a rate determined by the gradient formula \(y = mx\). For this calculation, the \(y\) is represented by the workload intensity that leads the resource usage to a level that breaches the threshold, while the \(x\) is represented by the total scaling time (reaction time + scaling time). In this figure, we also can see that at this rate, the resource is often overused. To avoid the resource over usage, we should increase the workload in a lower rate.

Figure 3 (b) illustrates a *low rate*. This rate is also determined by the gradient formula, though we represent the \(y\) by the difference between the total of resource and the resource usage that breaches the threshold. With this rate, the resource is not over used, but the resource variation occurs after a very long time, and moreover, the next one will be very far away. Therefore, the low rate prevents to test sequence of elasticity states, which could last many hours, or even days.

---

\(^1\)https://aws.amazon.com

\(^2\)http://jmow.org/rubbos.html
3. OUR APPROACH

In this section, we describe our approach to lead Web applications through elasticity states. In our approach we request the elasticity states by varying the workload in stages, instead of gradually, attempting to solve the problems discussed in the Section 2.3. We variate the workload in intensity, keeping the same workload pattern during all the leading process.

As workload pattern, we consider the set of requests sent to the Web application, which we represent in the Table 1. In this table, we consider that, except the requests R2 and R3, all the other requests are executed in parallel. In this case, the requests R2 and R3 are executed in sequence (R2, then R3), since the request R2 is set as predecessor of R3. However, this is only an illustrative case, and other workload patterns may differ from this one. Considering the workload pattern from Table 1 the variation of the workload consists in multiplying the amount of each request by the desired intensity. In this case, all the requests remain the same, only the concurrency is varied.

<table>
<thead>
<tr>
<th>ID</th>
<th>Request</th>
<th>Amount</th>
<th>Predecessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>GET /index.html HTTP/1.1</td>
<td>10</td>
<td>—</td>
</tr>
<tr>
<td>R2</td>
<td>GET /news.php HTTP/1.1</td>
<td>5</td>
<td>—</td>
</tr>
<tr>
<td>R3</td>
<td>GET /news.php?id=1 HTTP/1.1</td>
<td>4</td>
<td>R2</td>
</tr>
<tr>
<td>R4</td>
<td>POST /registration.html HTTP/1.1</td>
<td>2</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 1: Example of workload pattern.

Figure 4 illustrates our approach to variate the workload intensity, and as a consequence, to request the elasticity states.

3.1 Profiling

To lead the target Web application through the required elasticity states, we expose it to the workload intensities that lead the resource usage to levels that breach the thresholds (scale-in, and scale-out). To calculate these intensities, in the profiling phase, we discover which are the workload intensities necessary to breach the thresholds when the resource amount is equal to one. We call these workload intensities as multipliers, and use the symbols $I_{sio}$ and $I_{sii}$ to denote the scale-out and scale-in multipliers, respectively.

Then, based on the multipliers, we are able to calculate the intensities necessary to breach the thresholds when the amount of resource is greater than one (Section 3.2).

To discover the multipliers, we expose the Web application to a workload intensity that increases gradually in a hurried rate (see Section 3.2) until the scale-out threshold is breached. We use this rate, since it is fast, and does not stress the application until the scale-out threshold is breached. Since the scale-in threshold is lower than the scale-out threshold, going up to the scale-out threshold, allows to discover both multipliers ($I_{sio}$ and $I_{sii}$).

Table 2 shows an example of multipliers. In this example, we consider that the scale-out threshold corresponds to 60% of CPU usage, while the scale-in threshold corresponds to 20% of CPU usage. In this case, we also consider that the discovered scale-out multiplier is equal to 144, while the scale-in multiplier is equal to 30.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Threshold Value</th>
<th>Multiplier</th>
<th>Resource Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale-out</td>
<td>60% of CPU usage</td>
<td>$I_{sio}$ = 144</td>
<td>1</td>
</tr>
<tr>
<td>scale-in</td>
<td>20% of CPU usage</td>
<td>$I_{sii}$ = 30</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Example of multipliers.

3.2 Workload Calculation

In the workload calculation phase, we calculate the workload intensities necessary to lead the Web application through the required elasticity states. For this, each calculated workload intensity should breach the threshold related to an elasticity state.

We base the workload intensities calculation on the multiplier values, and on the results of Lloyd and Smith work [10]. This work shows that multiple web servers (a typical deployment of the Web applications on cloud) scale near-linearly (> 99%). Therefore, we assume that the workload intensity necessary to breach the threshold when the amount of resource is greater than one, is equal to the current amount of resource multiplied by the corresponding multiplier, i.e., $I_{sio}$ for scale-out, or $I_{sii}$ for scale-in.

Table 3 exemplifies the calculation of the workload intensities for a sequence of elasticity states. In this table, the required states correspond to a sequence of three scale-outs and two scale-ins, where after each elasticity state the amount of resource is increased or decreased accordingly.
### 3.3 Application Leading

During the application leading phase, we lead the Web application through the desired elasticity states. This consists in exposing the Web application to each one of the workload intensity calculated in the previous phase until the elasticity state related to the intensity is completed. This phase can be better understood in the description of the Figure 6 (next section).

### 3.4 Implementation Architecture

In this section, we explain the implementation of our approach. Figure 5 illustrates the components of the implementation architecture, which is composed by a coordinator, and several generators.

![Figure 5: Implementation architecture of our approach.](image)

The coordinator has several roles: front-end, monitoring, calculation of workload intensity variations and synchronization of load tasks. Each generator runs and controls an instance of a load generation application, leading it to perform the load tasks received from the coordinator. Using multiple generators reproduces a more realistic workload than using a single instance, allows to generate high workload intensities, and may prevent false-positive attacks.

Figure 6 shows the sequence diagram that represents the implementation of the application leading phase. In this diagram, for each calculated workload intensity, the coordinator splits the workload generation task among the multiple generators. Then, each generator generates its portion of the workload, and when the coordinator identifies that the required elasticity state is performed, it sends the next workload intensity to the generators. This process is repeated until all of the workload intensities are generated. To identify the elasticity states, the coordinator monitors the resource status on the cloud frontend.

![Figure 6: Sequence diagram of the application leading phase.](image)

In the current version, our tool is implemented in Java. All the parameters are set using a property file. The synchronization tasks are performed by remote method invocation (RMI). The generators run any load generation application that has an API, or supports command line execution. If an application programming interface (API) is available, the monitoring tasks are performed using it. Otherwise, a virtual machine used to host the web application is accessed remotely, and the `sysstat` tool is used for monitoring. The remote access is also used if frequent monitoring (e.g., every second) is necessary, since this is usually not allowed by the cloud providers APIs. In the case where remote access is not allowed, our tool can also monitor the cloud infrastructures by reading the resource status on cloud provider status pages.

### 4. EXPERIMENTS AND DISCUSSION

In this section, we present the experiments and discuss their results. All the experiments are conducted on Amazon EC2, where we set the scale-out threshold as 60% of CPU usage, and the scale-in threshold as 20% of CPU usage. The workload multipliers that breach these thresholds are predicted as: 144 (Io), and 30 (Ii), the same as the values of Section 3.1. The total scale-out time is equal to 360 seconds, while the total scale-in time is equal to 120 seconds.

We represent the Web application with the PHP version of the RUBBoS, a realistic web application modeled after an online news forum like Slashdot. The RUBBoS is deployed on n web servers with a centralized database server. For each web server we use a distinct virtual machine with a standard capacity (m3.medium). The database server is deployed on a large capacity machine (m3.large) at the same geographic region of the web servers, which avoids bottlenecks.

We generate the workload using the RUBBoS benchmark tool, where the workload intensity is varied in number of clients. For the first two experiments, the RUBBoS benchmark is executed natively, and it is distributed over 10 machines. For the third experiment, we use our tool to control the RUBBoS benchmark. Our tool is deployed with 1 coordinator and 10 generators. In all the experiments, the RUBBoS benchmark is deployed at same cloud provider and geographic region of the Web application, which prevents bandwidth problems. All the machines used to deploy the RUBBoS benchmark are of the large capacity type.

Table 4 describes the configuration of each machine type used in the experiments.

<table>
<thead>
<tr>
<th>Machine Type</th>
<th>CPU</th>
<th>Memory</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>m3.medium</td>
<td>1 vCPU (2.4 GHz)</td>
<td>3.7 GB</td>
<td>10 GB</td>
</tr>
<tr>
<td>m3.large</td>
<td>2 vCPU (2.4 GHz)</td>
<td>7.5 GB</td>
<td>10 GB</td>
</tr>
</tbody>
</table>

Table 4: Machines configuration.

### 4.1 Gradual Workload Variation in a Hurried Rate

This first experiment is conducted to confirm our assumption about leading Web applications through the required elasticity states by using a workload that varies gradually over time in a hurried rate. The hurried rate is calculated according to the gradient formula from Section 2.3. This results in an increasing rate equals to 0.4 (144 / 360) per second, \(^3\)http://sebastien.godard.pagesperso-orange.fr

\(^4\)http://slashdot.org/

\(^5\)Configuration of the benchmark: https://goo.gl/6Mn0ro
and a decreasing rate equals to 0.25 (30/120) per second. We first increase the workload intensity from 0 to 1152, and at the end of this increase we keep the workload at the higher intensity for a certain time (total scale-out time), then we decrease it from 270 to 0. This results in an execution time bigger than one hour (enough time to gather the results that confirm our assumption).

Increasing the workload intensity up to 1152 gradually in a hurried rate should lead the Web application to 7 scale-outs (8 machines), based on the calculations that we make in the Section 3.2. Also based on the calculation of the Section 3.2 with 8 machines, the workload intensity necessary to request a scale-in is equal to 240.

At the workload increase, the workload start being increased from 0, which lasts a little (≈ total scale-out time) until the first scale-out threshold is breached. To reproduce a beginning such as the workload increase step, we set 270 (240 + 30 = 270, where 30 is the scale-in intensity factor) as the value from which the workload starts decreasing. Thus, the workload is decreased during a certain time (≈ total scale-in time) before breaching the first scale-in threshold.

Figure 7 illustrates the behavior of the Web application, and the cloud infrastructure during this experiment.

4.2 Gradual Workload Variation in a Low Rate

This second experiment intends to confirm our assumption about leading Web applications through the required elasticity states. For that, we use the same elasticity states of the first experiment, i.e., 7 scale-outs, followed by 7 scale-ins. We vary the workload intensity step-by-step, according to the calculated workload intensities (see Section 3.2):

- scale-out: 144, 288, 432, 576, 720, 864, 1008, 1152;
- scale-in: 240, 210, 180, 150, 120, 90, 60;

Figure 8 illustrates the results of this experiment execution.

4.3 Our Tool Validation

This last experiment is conducted to validate the ability of our approach leading Web applications through the required elasticity states. For that, we use the same elasticity states of the first experiment, i.e., 7 scale-outs, followed by 7 scale-ins. We vary the workload intensity step-by-step, according to the calculated workload intensities (see Section 3.2):

- scale-out: 144, 288, 432, 576, 720, 864, 1008, 1152;
- scale-in: 240, 210, 180, 150, 120, 90, 60;

Figure 9 illustrates the results of this experiment execution.
5. RELATED WORK

Gambrini et al. have two work that address elasticity testing [6, 11]. In the first work, the authors predict elasticity state transition based on workload variations and test whether cloud infrastructures react accordingly. The second work presents a tool for automatic testing cloud-based elastic systems, however this tool does not take into account elasticity states. Our approach is similar to their first work in two points: both trigger resource variations without stress and consider elasticity states. However, the authors classify the elasticity states according to amount of resource allocated, while we also consider as elasticity state the periods where the resource is being allocated. We also control the elasticity state, while they do not.

Malkowski et al. [12] focus on controlling the elasticity of n-tier applications, such as Web applications. This work is similar to ours in two points: it addresses n-tier applications, and it bases its models on previous application runs. Although, it addresses the cloud infrastructure side, while our approach interacts directly with application (opposite side).

Other work is related to web performance measurement [14, 13, 10, 8]. The Web performance is also addressed by commercial tools, such as Apache JMeter and Blazemeter. However, none of them addresses the elasticity.

6. CONCLUSION

In this paper we propose an approach that automates and controls the workload generation in order to lead Web applications through elasticity states. Our approach gathers the unknown elasticity variables by experimentation, then calculates the workload variations, and leads the application through a list of required elasticity states automatically.

The results of the first and second experiments (Sections 4.1 and 4.2) indicates that the gradual variation of the workload is not a good way to request the elasticity states. In these experiments we see that when the elasticity states are requested in a hurried rate, the Web application is stressed. In addition, to not stress the Web application we experiment the application with a low rate, which extends the time of requesting the elasticity states. The third experiment (Section 4.3) shows that our approach generates the workload variations that lead the web applications through the required elasticity states. In this experiment, the web application is led without stress, and in a time close to the experiment 1 (fastest). Therefore we can say that our approach is able to lead web applications through the required elasticity states, allowing a reduced execution time.

This paper is the first step towards an approach to test Web applications through elasticity. At the moment, we lead web applications through required elasticity states. Although, we are also interested on aspects related to the test, such as test case and test oracle. As a future work, we intend to propose an strategy to test Web applications through elasticity.

Acknowledgments

This work is supported by CAPES Foundation (Science Without Borders process 9070-13-3), Ministry of Education of Brazil.

7. REFERENCES
