Monitoring-Based Testing of Elastic Cloud Computing Applications

Michel Albonico, Jean-Marie Mottu, Gerson Gerson Sunyé

Monitoring-Based Testing of Elastic Cloud Computing Applications

Michel Albonico
AtlanMod team (Inria, Mines Nantes, Lina)
UTFPR, Brazil and Mines Nantes, France
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

Applications that are exposed to large-scale workloads must ensure elasticity, that is the ability to scale up and down rapidly to meet the demand. Cloud infrastructures provide adaptation tasks, which allow applications to automatically scale up and down straightforwardly. These adaptation tasks drive the system to new states, which may expose implementation errors and therefore must be tested. In this paper, we focus on testing elastic applications during different elasticity-related states. This test is difficult since the elasticity states are not directly controlled by the tester. To execute the test at different elasticity-related states, we propose a monitoring-based procedure. This procedure consists in monitoring the resource status to identify the occurrences of the elasticity states at real-time, and in parallel, execute the state-related tests. To validate our test procedure, we performed experiments on Amazon EC2. These experiments successfully identified non-functional errors.

Keywords

Cloud computing; elasticity; testing; elasticity testing.

1. INTRODUCTION

Some applications are often exposed to huge workloads. Sometimes, to deal with these workloads, the applications must be scaled in a large manner. Managing large scale applications is not a trivial task, and it is difficult to achieve manually. The usage of cloud computing may improve the management of these applications. Cloud computing infrastructures provide elasticity, where system resources are allocate and deallocated automatically, according to demand.

Elastic infrastructures vary the resources at runtime. To deal with these variations, the 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. [4]: component synchronization, registration, and data replication.

Adaptation tasks may introduce functional, and non-functional errors into the application. Considering database applications, for instance, a functional error may happen due to inconsistencies on data replication, which may result in an unexpected result. Non-functional errors may be characterized by rejected requests and unacceptable response times. They may result from a delay in a new resource allocation (e. g., database replication node), a problem in reconfiguring the master component, or even a problem in the algorithm for master election.

Some of the adaptation tasks may only execute during specific resource variations periods, during resource allocation and deallocation. We call these periods *elasticity states* and classify them as: scaling-out (resource allocation), ready (resource steady), and scaling-in (resource deallocation). We believe that to find most of the errors resulted from the adaptation tasks, we should test the application during all the elasticity states. This is a challenging task, since we must identify the elasticity states at real time, and in parallel, execute the test accordingly.

In this paper, we focus on two questions about elasticity testing:

- Question 1: Is it necessary to run the test of the application during different elasticity states?
- Question 2: Is it possible to execute the test during different elasticity states and to assign the test verdicts accordingly?

To answer to these questions, we conduct two experiments on Amazon EC2, using the MongoDB as the database layer, and the Yahoo! Cloud Serving Benchmark as the database application. In the first experiment, we focus on non-functional benchmarking and we detect significant performance variation that only occur during the resource variations. This indicates that it is necessary to test the application during different elasticity states. To answer the second question, we propose a *test procedure*, which consists in monitoring the resource status to identify the elasticity states, and switching the test (according to the test specifications) every time a different elasticity state is identified. Using this test procedure, in a second experiment, we identify all the non-functional failures of the first experiment, which are successfully assigned to the elasticity states during which they occur.

ICPE'16 Companion, March 12-18, 2016, Delft, Netherlands

 ISBN 978-1-4503-4147-9/16/03...$15.00

DOI: http://dx.doi.org/10.1145/2859889.2859890

The rest of the paper is organized as follows. In the next section, we present the major aspects of cloud computing elasticity. In Section 3, we propose a procedure for testing elasticity. In Section 4, we describe the experiments and results. In Section 5 we discuss the related work. Finally, in Section 6 we conclude.

2. BACKGROUND

2.1 Cloud Computing Elasticity

Different authors [1, 3, 7, 4] 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.

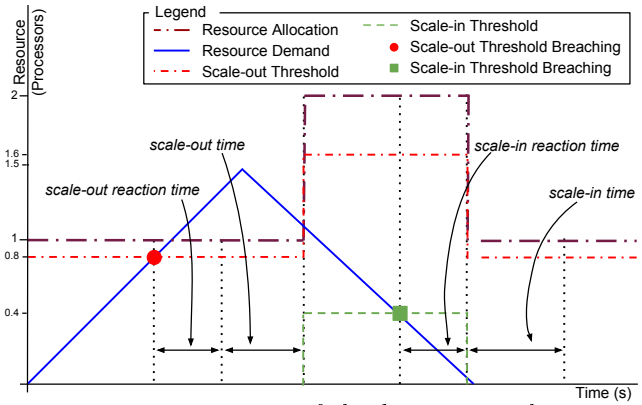

Figure 1: Representation of cloud computing elasticity.

When the resource demand exceeds the *scale-out threshold*, and remains higher during the *scale-out reaction time*, the cloud elasticity mechanism assigns a new resource. The new resource is available after a *scale-out time*, the time the cloud infrastructure spends to allocate the new resource. Once the resource is available, the threshold values are updated accordingly.

When the resource demand decreases, breaches the *scale-in threshold*, and remains lower during the *scale-in reaction time*, the cloud elasticity mechanism releases 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 pressure the application, demanding different amounts of resource. When the application is deployed on a cloud infrastructure, those fluctuations lead to resource variation (elasticity). Figure 2 illustrates the elasticity states of cloud applications.

At the beginning, the application is in the *ready* state: the resource configuration is steady. Then, if the cloud elasticity mechanism adds a new resource, the application enters the *scaling-out* state: the period while the resource is added. Otherwise, if some resource is released, the application enters the *scaling-in* state: the period while the resource is released. After a *scaling-out* or *scaling-in*, the application returns to the *ready* state, and remains in this state until a new resource variation is requested.

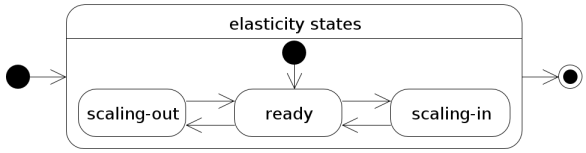

Figure 2: Elasticity states and their transitions.

2.3 Adaptation Tasks

Given that the elasticity is related to the ability of an application to adapt itself to workload changes and resource demands, the existence of at least one specific adaptation process is assumed [7]. We call these processes *adaptation tasks*. For instance, coordination activities, such as master election and data replication, may be considered as a type of adaptation task.

3. ELASTICITY TEST PROCEDURE

In this section, we present a procedure for testing the elasticity of cloud applications. For this, we expose the application to a workload that leads it through different elasticity states. In parallel, we run our elasticity test procedure which monitors the resource status to identify the elasticity states, and executes the tests according to their test specifications.

This section details the two parts of the test procedure: *test specification* and *test execution*. Here, we do not address aspects related to the workload generation. This is done by using a proper benchmark/load tool [2].

3.1 Test Modelization

Figure 3 presents the model of how we specify test structure, which is composed by one or more *TestCase* elements. Each one of the *Test Case* element sets test activities, which consist in sending inputs to the application, gathering the outputs (application reaction to these inputs), asserting these outputs, and returning the test case verdict. These elements are implemented by JUnit TestCase classes, wherein the outputs are asserted by using JUnit assertion methods. Additionally, *Test Case* elements are also related to one or more elasticity states, which is set in the variable *eState*.

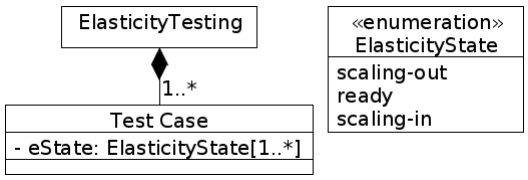

Figure 3: Elasticity Test model.

3.2 Test Execution

The test execution is represented by the Algorithm 1. This algorithm receives as input the test specification (T), discussed in the previous section, and the execution timeout (*eto*) as the stopping condition. In the Algorithm 1, we first start a monitoring process. The monitoring process gathers information from different sources (i. e., API, HTTP reading, and SSH session), which makes it independent of cloud provider. Based on the monitored information,

we identify the occurrences of elasticity states. In parallel, we execute the test cases related to the current elasticity state. This process is repeated up to the execution time ($t_{begin} - current\_time < eto$).

The test execution is switched in a while loop, at each loop we execute all the test cases associated to the current elasticity state (cs). After each test case execution, we store its verdict (JUnit assertion) into a matrix ($V^{cs_i,tc}$). To prevent verdict overwriting, we use the index i in the current state. Since the application remains in the same state for a while, several adaptation tasks may run throughout this period. Using the while loop, the associated test cases are re-executed along the current elasticity state (cs), which allows us to cover all the adaptation tasks.

Algorithm 1: Test Execution

```
Data: Test Specification T, Execution Timeout eto
mon ← monitor();
i = 0;
t_begin ← current_time;
while t_begin − current_time < eto do
    foreach tc ⊂ T.testCases do
        i = i + 1;
        cs_i ← mon.currentState();
        if tc.states.contains(cs_i) then
            V^{cs_i,tc} ← run(tc);
        end
    end
end
```

4. EXPERIMENTS

To answer the two motivation questions (Section 1) of the paper, we conduct two experiments. Both of them focus on non-functional problems. In the first one, we benchmark the application to check the existence of performance problems during resource variation (scale-in and scale-out elastic states). In the second one, we check whether we are able to execute a test during different elasticity states, and set the test verdict accordingly. In both experiments, the completeness of the test is not an objective: we are interested in the necessity and possibility of elasticity testing during different elasticity states. The performance and completeness of the test is part of the future work.

In those experiments, we use a distributed MongoDB (version $r3.0.7$) to represent an elastic database. The workload is generated by the Yahoo Cloud Serving Benchmark (YCSB), which we consider as the database application. Both, MongoDB and YCSB, are deployed on Amazon EC2 cloud infrastructure. The mongoDB components are deployed on distinct virtual machines with standard capacity ($m3.medium$): 1 vCPU (2.4 GHz), memory (3.7 GB), and disk (10 GB). The YCSB is deployed on a virtual machine with large capacity machine ($m3.large$): 2 vCPU (2.4 GHz), memory (7.5 GB), and disk (10 GB).

4.1 First experiment: elasticity testing necessity

In this experiment, MongoDB is deployed as a replica set. We start the replica set with a master replica, then we add/remove two slave replicas one at a time, several times.

We execute the YCSB at this experiment, without using our procedure. We send a YCSB workload (workload A) to the application, whereas this workload is fixed at 2500 operations per second. The YCSB sends the operations to the MongoDB within 10 seconds of interval and the test case execution lasts $\approx$ 22 minutes. We consider the number of operations per second (ops) as the performance metric. Then, we verify whether this property remains higher than 2000 ops. Otherwise, we consider it as a performance problem.

The YCSB executes continuously, without elasticity state distinction, and the result is gathered after its execution, from the logs. Figure 4 summarizes the result of this experiment.

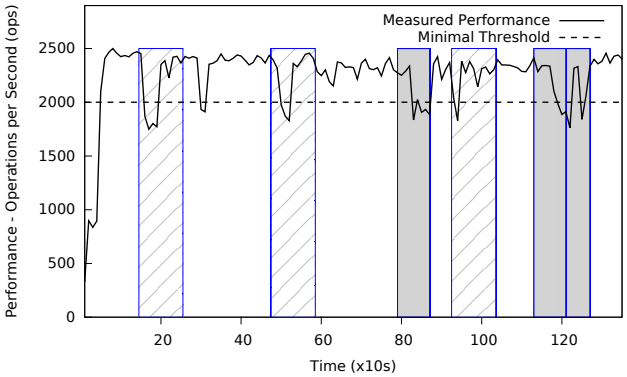

Figure 4: Result of the first experiment.

In Figure 4, the solid line represents the measured performance, the dashed line represents the criteria performance, the stripped areas represent the scaling-out states, the gray areas represent the scaling-in states, and the other areas (no outline) are the ready states. We see that during several periods there are performance drops and that these drops happen during any of the elasticity states. Most of them happen at scaling states (scaling-out, and scaling-in), where they always happen, and that only two of the performance drops happen at ready states. The two performance problems at ready states happen due to adaptation tasks that only occur once at the MongoDB run: when the benchmark is warming up and during the master election (from a standalone to a distributed deployment).

We conclude this first experiment answering to the Question 1 positively: it is necessary to run the tests of the application during different elasticity states since errors occur during any of elasticity states.

4.2 Second experiment: elasticity test possibility

In this experiment, we use our test procedure to set the test according to the test specification model (Section 3.1) and to implement our test execution algorithm.

We repeat the MongoDB replica set configuration and set a unique test case, which is associated to every elasticity state. The test case consists in sending to the MongoDB a YCSB workload at a rate of 2500 operations per second. Here, we aim at getting a finite set of test verdicts, assigned according to the reached elasticity states. For that, the test case is associated to every elasticity state, which results in its re-execution along the experiment (see Algorithm 1).

The verdict *pass* is assigned when the amount of opera-

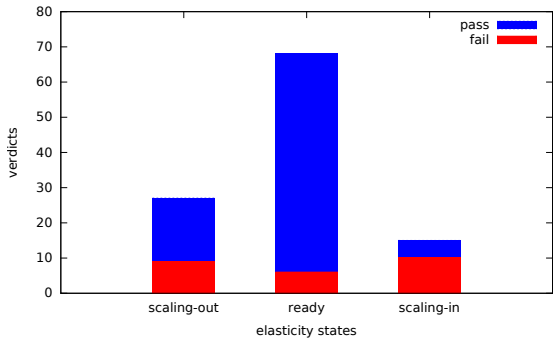

Figure 5: Result of the second experiment.

tions per second is higher than 2000, otherwise, the verdict is *fail*. Figure 5 illustrates the amount of pass and fail verdicts assigned to each elasticity state.

In the figure, each stacked bar represents the verdicts of a single elasticity state. We see that all the states have both verdicts, pass and fail. This answers to the Question 2 positively: we made possible to execute a test exactly during different elasticity states, and to assign the test verdict accordingly. The proportion between pass and fail verdicts in both experiments is similar, which indicates that our test approach does not influence the test case verdicts. For this comparison, at the first experiment we consider the performance drops (< 2000 ops) as fail, and the other performance measurements as pass verdict.

At the first experiment we find the performance problems manually, without identifying the elasticity states at real time. In this manner, we must discover this information after the execution, such as by analyzing log files, which may be toilsome and error-prone. Our test procedure verdict assignment helps by analyzing test failures automatically, at real time, and one elasticity state at a time (instead of getting a continuous line of performance, such as the one of the Figure 4).

5. RELATED WORK

Gambi et al. have two works that address elasticity testing [6, 5]. 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 since both 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 (i. e., scaling-out, and scaling-in). In addition to these two work, Truong and Dustdar also have two work [9, 8] related to elasticity testing. Their first work proposes a building block for multi-dimensional elasticity programming, which resembles our work in monitoring the elasticity. However, they consider different dimensions, such as quality and costs elasticity. In the second work, they claim for new techniques for testing elastic applications, however they do not propose any test approach. Therefore, to our knowledge, our work is the only one that focus on testing applications during the scaling states.

6. CONCLUSION AND FUTURE WORK

In this paper we aim at answering two main questions: Is it necessary to run the tests of the application during different elasticity states? Is it possible to execute the test during different elasticity states and to assign the test verdicts accordingly? We answered to the two questions positively, based on the execution of a couple of experiments on a large-scale application on Amazon EC2. The second question is answered thanks to an elasticity test procedure that we propose. Our test procedure innovates by running the test through different elasticity states, assigning the verdicts automatically and at real time.

This paper is a step towards an approach to test applications through elasticity. At the moment, we are able to identify the elasticity states at real time and to execute the test according to test specification, set by the user. As a future work, we intend to consider test criteria helping selecting test when testing elastic applications. We illustrate this paper with non-functional testing, but we are currently working on identifying specific elasticity bugs. We will also focus on analyzing and categorizing the errors detected, it will help when creating test cases seeking them but also when debugging. In particular, we do not discriminate yet between application errors and cloud infrastructure errors.

Acknowledgments

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

7. REFERENCES

[1] D. Agrawal, A. El Abbadi, S. Das, and A. J. Elmore. Database scalability, elasticity, and autonomy in the cloud. *Proceedings of DASFAA'11*, 2011.

[2] M. Albonico, J.-M. Mottu, and G. Sunyé. Controlling the Elasticity of Web Applications on Cloud Computing. In *Proceedings of the 31st SAC*. ACM, 2016.

[3] L. Badger, T. Grance, R. Patt-Comer, and J. Voas. *Draft Cloud Computing Synopsis and Recommendations*. Nist Special Publication 800-146, 2011.

[4] M. M. Bersani, D. Bianculli, S. Dustdar, A. Gambi, C. Ghezzi, and S. Krstić. Towards the Formalization of Properties of Cloud-based Elastic Systems. In *Proceedings of PESOS 2014*. ACM, 2014.

[5] A. Gambi, W. Hummer, and S. Dustdar. Automated testing of cloud-based elastic systems with AUToCLES. In *Proceedings of ASE 2013*. IEEE, 2013.

[6] A. Gambi, W. Hummer, H.-L. Truong, and S. Dustdar. Testing Elastic Computing Systems. *IEEE Internet Computing*, 2013.

[7] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not. *ICAC*, 2013.

[8] H.-L. Truong and S. Dustdar. Programming Elasticity in the Cloud. *Computer*, 2015.

[9] H. L. Truong, S. Dustdar, G. Copil, A. Gambi, W. Hummer, D. H. Le, and D. Moldovan. CoMoT - A Platform-as-a-Service for Elasticity in the Cloud. In *Proceedings of IC2E*, 2014.