Making Cloud-based Systems Elasticity Testing Reproducible

Michel Albonico, Jean-Marie Mottu, Gerson Sunyé, Frederico Alvares

Making Cloud-based Systems Elasticity Testing Reproducible

Michel Albonico[1,3], Jean-Marie Mottu[1], Gerson Sunye[1] and Frederico Alvares[2]
[1] *AtlanModels team (Inria, IMT-A, LS2N), France*
[2] *Ascola team (Inria, IMT-A, LS2N), France*
[3] *Federal Technological University of Parana, Francisco Beltrão, Brazil*
michelalbonico@utfpr.edu.br, {jean-marie.mottu,gerson.sunye,frederico.alvares}@inria.fr

Keywords:
Cloud Computing, Elasticity, Elasticity Testing, Controllability, Reproducibility.

Abstract:
Elastic cloud infrastructures vary computational resources at runtime, i. e., elasticity, which is error-prone. That makes testing throughout elasticity crucial for those systems. Those errors are detected thanks to tests that should run deterministically many times all along the development. However, elasticity testing reproduction requires several features not supported natively by the main cloud providers, such as Amazon EC2. We identify three requirements that we claim to be indispensable to ensure elasticity testing reproducibility: to control the elasticity behavior, to select specific resources to be unallocated, and coordinate events parallel to elasticity. In this paper, we propose an approach fulfilling those requirements and making the elasticity testing reproducible. To validate our approach, we perform three experiments on representative bugs, where our approach succeeds in reproducing all the bugs.

1 Introduction

Elasticity is one of the main reasons that makes cloud computing an emerging trend. It allows to vary (allocate or deallocate) system resources automatically, according to demand [Agrawal et al., 2011, Herbst et al., 2013, Bersani et al., 2014]. Therefore, *Cloud-Based Systems* (CBS) must adapt themselves to resource variations.

Due to their scalability nature, CBS adaptations may be complex and error-prone [Herbst et al., 2013]. This is the case, for instance, of bug 2164 that affects Apache ZooKeeper [Hunt et al., 2010], a coordination service for large-scale systems. According to the bug report, the election of a new leader never ends when the leader leaves a system with three nodes. This happens when the resource (e. g., virtual machine) that hosts the leader node is deallocated.

Furthermore, some bug reproductions require more than adaptation tasks. This is the case of MongoDB NoSQL database bug 7974, where elasticity and consequently, adaptation tasks, happen in parallel with other events. In that case, adaptation tasks are not the unique cause of the bug, though they are still a requirement for its reproduction.

Reproducing bugs several times is necessary to diagnose and to correct them. That is, bugs reported in a CBS bug tracking should be corrected by developers, who need to reproduce them, at first. Moreover, during the development, regression tests should be run regularly [Engstrom et al., 2010], requiring the design of efficient and deterministic tests. Making CBS testing reproducible is then an important concern.

In this paper, we consider any bug that occurs in presence of elasticity as an *elasticity-related bug*. To verify the existence of this kind of bug in the real-world and to get insights about them, we analyze MongoDB's bug tracking, a popular CBS. In the bug tracking we identify a total of 43 bugs that involve elasticity. Since these bugs occur in real-world, testers must manage elasticity when writing and running their tests. However, writing and executing tests considering the reproduction of elasticity-related bugs is complex. Besides the

typical difficulty of writing test scenarios and oracles, it requires to deterministically manage the elasticity and parallel events.

The management of elasticity involves driving the system through a sequence of resource changes, which is possible by varying the workload accordingly [Albonico et al., 2016]. In addition, tester should also manage parameters that cannot be natively controlled. For instance, resource changes are managed by the cloud provider according to its own policy. This prevents the tester to choose which resource must be deallocated, a requirement for 19 of the 43 MongoDB's elasticity-related bugs. Another parameter that the tester should manage is the coordination of events in parallel to elasticity (17/43 bugs) that, among others, requires a precise monitoring of resources.

In this paper, we present an approach for elasticity testing that allows the tester to reproduce tests concerned with elasticity-related bugs. We control the elastic behavior by sending satisfactory workload to CBS that drives it through a sequence of resource allocations and deallocations. Our approach also provides two original contributions: the deallocation of specific resources, and the coordination of events in parallel with elasticity. The two original contributions are required to reproduce 30 of the 43 elasticity-related bugs of MongoDB, i. e., $\approx 70\,\%$. In the paper, we also present a prototype for the approach.

To support our claims and to validate our approach, we select 3 representative bugs out of the 43 elasticity-related bugs previously mentioned, and conducted a set of experiments on Amazon EC2. All the 3 bugs require our approach to satisfy at least one of the requirements we identified. Experiment results show that our approach reproduces real-world elasticity-related bugs fulfilling all the 3 requirements.

The paper is organized as follows. In the next section, we remind the major aspects of cloud computing elasticity. Section 3 details the requirements we claim be necessary for elasticity testing. Section 4 introduces the testing approach we propose. The experiments and their results are described in Section 5. Section 6 discusses threats to validity. Finally, Section 7 concludes.

2 Cloud Computing Elasticity

This section provides the definitions of the main concepts related to Cloud Computing Elasticity that are required for the good understanding of our approach.

2.1 Typical Elastic Behavior

Figure 1 presents the typical behavior of elastic cloud computing applications [Albonico et al., 2016]. In this figure, the *resource demand* (continuous line) varies over time, increasing from 0 to 1.5 then decreasing to 0. A resource demand of 1.5 means that the application demands 50 % more resources than the current allocated ones.

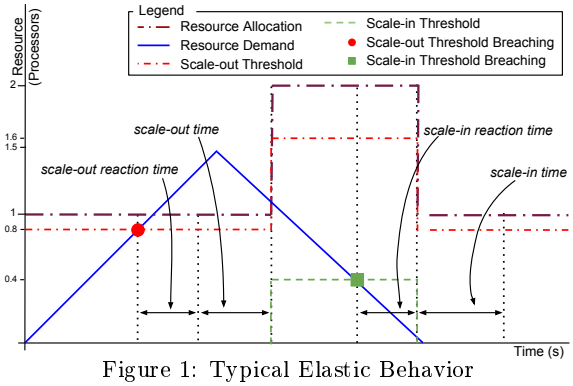

Figure 1: Typical Elastic Behavior

When the resource demand exceeds the *scale-out threshold* and remains higher during the *scale-out reaction time*, the cloud elasticity controller assigns a new resource. The new resource becomes available after a *scale-out time*, the time the cloud infrastructure spends to allocate it. Once the resource is available, the threshold values are updated accordingly. It is similar considering the *scale-in*, respectively. Except that, as soon as the scale-in begins, the threshold values are updated and the resource is no longer available. Nonetheless, the infrastructure needs a *scale-in time* to release the resource.

2.2 Elasticity States

When an application is deployed on a cloud infrastructure, workload fluctuations lead to resource variations (elasticity). These variations drive the application to new, elasticity-related, states. Figure 2 depicts the runtime lifecycle of an application running on a cloud infrastructure.

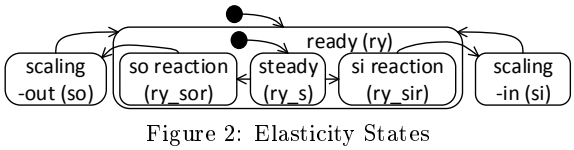

Figure 2: Elasticity States

At the beginning the application is at the *ready* state (*ry*), when the resource configuration is steady (*ry_s* substate). Then, if the application is exposed for a certain time (*scale-out reaction time*, *ry_sor* substate) to a pressure that breaches the scale-out threshold, the cloud elasticity controller starts adding a new resource. At this point, the application moves to the *scaling-out* state (*so*) and remains in this state while the resource is added. After a *scaling-out*, the application returns to the *ready* state. It is similar with the *scaling-in* state (*si*), respectively.

2.3 Elasticity Control

We can categorize *elasticity control* approaches in two groups: (i) direct resource management, and (ii) generation of adequate workload.

The first and simplest one (i) interacts directly with the cloud infrastructure, asking for resource allocation and deallocation. The second one (ii) consists in generating adequate workload variations that drive CBS throughout elasticity states, as previously explained in Section 1. It is more complex since requires a preliminary step for profiling the CBS resource usage, and calculating the workload variations that trigger the elasticity states.

3 Requirements for Elasticity Testing Reproduction

In this paper, we consider three requirements for reproducing elasticity testing: *elasticity control*, *selective elasticity*, and *events scheduling*.

Elasticity Control is the ability to reproduce a specific elastic behavior. Elasticity-related bugs may occur after a specific sequence of resource allocations and deallocations. Logically, all the 43 elasticity-related bugs of MongoDB should satisfy this requirement.

Analyzing elasticity-related bugs, we identify two other requirements: *selective elasticity*, and *event scheduling*. Those requirements are necessary to reproduce 30 of the 43 selected bugs.

Selective Elasticity is the necessity to specify precisely which resource must be deallocated. The reproduction of some elasticity-related bugs requires a deterministic management of elasticity changes. For instance, deallocating a resource associated to the master component of a cloud-based system. From the selected bugs, 19 require selective elasticity.

Event Scheduling is the necessity to synchronize elasticity changes with parallel events. We consider as an event any interaction or stimulus to CBS, such as forcing a data increment or to simulate infrastructure failures. The reproduction of 17 of MongoDB elasticity-related bugs requires event scheduling.

Table 1 shows the number of bugs that should meet those requirements in order to be reproducible. As already said, all the bugs require elasticity control.A total of 30 bugs (70 %) should meet requirements besides elasticity, where 6 bugs need all the requirements. Finally, only 13 bugs only need elasticity control (30 %) .

	Elasticity Control	Selective Elasticity	Event Scheduling	All	Only Elasticity Control
Quantity	*43*	*19*	*17*	*6*	*13*

Table 1: Requirements for Bug Reproductions

4 Elasticity Testing Approach

In this section, we present the overall architecture of our approach.

4.1 Architecture Overview

Figure 3 depicts the overall architecture of our approach. The architecture has four main components: *Elasticity Controller Mock* (ECM), *Workload Generator* (WG), *Event Scheduler* (ES), and *Cloud Monitor* (CM).

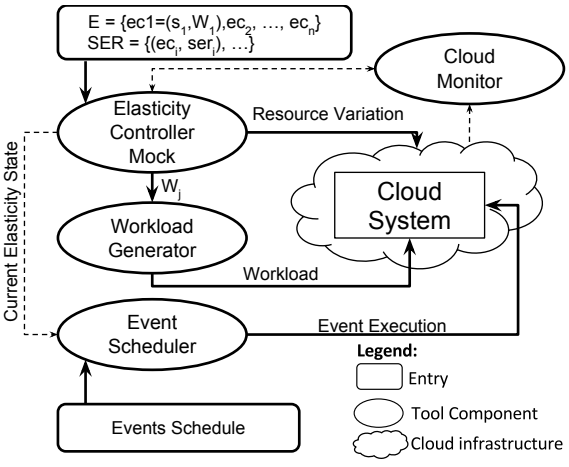

Figure 3: Overall Architecture

The *ECM* simulates the behavior of the cloud provider elasticity controller, allocating and deallocating determined resources, according to test-

ing needs. It also asks the *WG* to generate the workload accordingly, reproducing a realistic scenario. The role of the *ES* is to schedule and execute a sequence of events in parallel with the other components. Finally, the *CM* monitors the cloud system, gathering information that helps orchestrating the behavior of the three other components, ensuring the sequence of elasticity states, and their synchronization with the events.

Table 2 summarizes the requirements that each component helps in ensuring, as we detail in this section.

	REQUIREMENT		
COMPONENT	Elasticity Control	Selective Elasticity	Event Scheduling
ECM	*YES*	*YES*	*YES*
WG	*YES*	*NO*	*NO*
ES	*NO*	*NO*	*YES*
CM	*YES*	*YES*	*NO*

Table 2: Requirements Ensured by Architecture's Components

4.1.1 Elasticity Controller Mock

The ECM is designed to reproduce the elastic behavior. By default, ECM requires as input a sequence of *elasticity changes*, denoted by $E = \{ec_1, ec_2, ..., ec_n\}$, where each ec is a pair that corresponds to an elasticity change. Elasticity change pairs are composed of a required elasticity state (s_i) and a workload (W_i), $ec_i = \langle s_i, W_i \rangle$ where $1 \leq i \leq n$. A workload is characterized by an intensity (i. e., amount of operations per second), and a workload type (i. e., set of transactions sent to the cloud system).

ECM reads elasticity change pairs sequentially. For each pair, ECM requests resource changes to meet elasticity state s_i and requests the Workload Generator to apply the workload W_i. Indeed, we have to send the corresponding workload to prevent cloud infrastructure to provoke unexpected resource variations. In particular, it could deallocate a resource that ECM just allocated, because the workload has remained low and under the scale-in threshold.

Instead of waiting for the cloud computing infrastructures for elasticity changes, it directly requests to change the resource allocation (*elasticity control*). Based on both, required elasticity state and workload (elasticity change pair), ECM anticipates the resource changes. To be sure CBS enters the expected elasticity state, ECM queries the Cloud Monitor, which periodically monitors the cloud infrastructure.

The ECM may also lead to a precise resource deallocation (*selective elasticity*). Typically, elasticity changes are transparent to the tester, managed by the cloud provider. To set up the *selective elasticity*, ECM requires a secondary input, i. e., Selective Elasticity Requests (SER). SER is denoted by $SER = \{(ec_1, ser_1), ..., (ec_n, ser_n)\}$, where $ec_i \in E$, and ser_i refers to a *selective elasticity request*. A *selective elasticity request* is a reference to an algorithm (freely written by tester) that gets a resource's ID. When ec_i is performed by ECM, the algorithm referred by ser_i is executed, and the resource with the returned ID is deallocated by ECM.

ECM helps in ensuring all of elasticity testing requirements. As earlier explained in this section, it deterministically requests resource variations (*elasticity control* and *selective elasticity*), and helps on ensuring the *event scheduling* providing information of the current elasticity state to the Event Scheduler.

4.1.2 Workload Generator

The Workload Generator is responsible for generating the workload (W). We base it on Albonico et al. work [Albonico et al., 2016], which takes into account a threshold-based elasticity (see Figure 1), where resource change demand occurs when a threshold is breached for a while (*reaction time*). Therefore, a workload should result in either threshold breached (for scaling states) or not breached (for ready state), during the necessary time. To ensure this, the Workload Generator keeps the workload constant, either breaching a threshold or not, until a new request arrives.

Considering a scale-out threshold is set as 60% of CPU usage, the workload should result in a CPU usage higher than 60% to request a scale-out. In that case, if 1 operation A hypothetically uses 1% of CPU, it would be necessary at least 61 operations A to request the scale-out. On the other hand, less than 61 operations would not breach the scale-out threshold, keeping the resource steady.

The Workload Generator contributes with the Elasticity Controller Mock to the *elasticity control* requirement.

4.1.3 Event Scheduler

The Event Scheduler input is a map associating sets of events to elasticity changes (ec_i), i. e., the set of events that should be sent to the cloud system when a given elasticity change is managed by the ECM. Table 3 abstracts an input where four events are associated to two elasticity changes.

Elasticity Change	Event ID	Execution Order	Wait Time
ec_1	$e1$	1	0 s
	$e2$	2	10 s
	$e3$	2	0 s
ec_2	$e2$	1	0 s
	$e4$	2	0 s

Table 3: Events Schedule

Periodically, the Event Scheduler polls the ECM for the current elasticity change, executing the events associated to it. For instance, when the ECM manages the elasticity change ec_1, it executes the events e_1, e_2, and e_3. Events have execution orders, which define priorities among events associated to the same state: event e_1 is executed before events e_2 and e_3. Events with the same *execution order* are executed in parallel (e. g., e_2 and e_3). Events are also associated to a *wait time*, used to delay the beginning of an event. In Table 3, event e_2 has a wait time of 10 s (starting 10 s after e_3, but nonetheless executed in parallel). This delay may be useful, for instance, to add a server to the server list a few seconds after the ready state begins, waiting for data synchronization to be finished.

The Event Scheduler ensures the *event scheduling* requirement.

4.1.4 Cloud Monitor

Cloud Monitor helps ECM to ensure *elasticity control* and *selective elasticity*. It periodically requests current elasticity state and stores it in order to respond to the ECM queries, necessary for elasticity control. It also executes the selective elasticity algorithm of SER, responding to ECM with the ID of the found resources.

4.2 Prototype Implementation

Each component of the testing approach architecture is implemented in Java and communicate with each other through Java RMI. Currently, we only support Amazon EC2 interactions, though one could adapt our prototype to interact with other cloud providers.

4.2.1 Elasticity Controller Mock

The elasticity changes are described in a property file. The entries are set as $\langle key, value \rangle$ pairs, as presented in Listing 1. The key corresponds to the elasticity change name, while the value corresponds to the elasticity change pair. The first part of the value is the elasticity state, and the second part is the workload, divided into intensity and type.

Listing 1: Example of Elasticity Controller Mock Input File (Elasticity Changes)

```
ec1=ready, (1000,write)
ec2=scaling-out, (2000,read/write)
...
ec4=scaling-in, (1500,read)
```

As previously explained, for each entry, the ECM sends the workload parameters to the Workload Generator and deterministically requests the specified resource change. Resource changes are requested through the cloud provider API, which enables resource allocation and deallocation, general infrastructure settings, and monitoring tasks. Before performing an elasticity change, the ECM asks the Cloud Monitor whether the previous elasticity state was reached. Cloud Monitor uses the Selenium[1] automated browser to gather pertinent information from cloud provider's dashboard Web page.

We use Java annotations to set up selective elasticity requests (SER), as illustrated in Listing 2. A Java method implements the code that identifies a specific resource and returns its identifier as a String type. It is annotated with metadata that specifies its name and associated elasticity change.

Listing 2: Selective Elasticity Input File

```
@Selection{name="ser1", elasticity_change="ec4"}
public String select1() {
        ... //code to find a resource ID
        return resourceID; }
```

4.2.2 Workload Generator

The WG generates the workload according to the parameters received from the ECM (i. e., *workload type* and *intensity*), whereas the workload is cyclically generated until new parameters arrive. It uses existing benchmark tools, setting the workload parameters in the command line.

For instance, YCSB benchmark tool allows three parameters related to the workload: the preset workload profile, the number of operations, and the number of threads. The preset workload profile refers to the workload type, while the multiplication of the two last parameters results in the workload intensity.

4.2.3 Event Scheduler

Event schedule is set in a Java file, where each event is an annotated method, such as the exam-

[1] `http://www.seleniumhq.org/`

ple illustrated in Listing 3. Java methods are annotated with the event identifier, the related elasticity change, the order, and the waiting time. EC periodically polls the ECM to obtain the current elasticity change. Then, it uses Java Reflection to identify and execute the Java methods related to it.

Listing 3: Example of Event Scheduler Input File

```
@Event{id="e1",elasticity_change="ec1",
order="1", wait="0"}
public void event1() { ... }
```

4.3 Prototype Execution

Before executing our approach's prototype, testers must deploy the CBS, which is done using an existing approach based on a Domain Specific Language (DSL) by Thiery et al. [Thiery et al., 2014]. The DSL enables us to abstract the deployment complexities inherent to CBS. Information in deployment file, such as cloud provider's credentials and Virtual Machine's configuration, are then used by the components of our prototype. However, since the deployment of CBS is not a contribution of our approach, we do not further explain it in this paper.

To execute our approach's prototype, testers write the input files: E, SER, and ES files. These files as well as the deployment file are passed to the prototype as command line parameters. Then, all the execution is automatically orchestrated.

5 Experiments

In this section, we present three experiments that aim at validating our approach for elasticity testing. We reproduce three representative elasticity-related bugs from two different CBS: MongoDB and ZooKeeper. The first bug is the *MongoDB-7974*, which we gather from the MongoDB's elasticity-related bugs described in Section 3. The other two bugs, *ZooKeeper-2164* and *ZooKeeper-2172*, we gather from the official bug tracking of ZooKeeper, another popular CBS.

We attempt to reproduced all the bugs in two ways: using our approach, and relying on the cloud computing infrastructure. Then, we compare both approaches to verify whether the requirements that we identify in this paper are met.

5.1 Experimental Environment

5.1.1 CBS Case Studies

MongoDB[2] is a NoSQL document database. MongoDB has three different components: the configuration server, MongoS and MongoD. The configuration server stores metadata and configuration settings. While MongoS instances are query routers, which ensure load balance, MongoD instances store and process data.

Apache ZooKeeper[3] is a server that provides highly reliable distributed coordination. ZooKeeper is intended to be replicated over a set of nodes, called as an ensemble. Requests from ZooKeeper's clients are forwarded to a single node, the *leader* (which is *elected* using a distributed algorithm). The leader works as a proxy, distributing the request among other nodes called as *followers*. The *followers* keep a local copy of the configuration data to respond to requests.

5.1.2 Cloud Computing Infrastructure

All the experiments are conducted on the commercial cloud provider Amazon Elastic Cloud Compute (EC2), where we set *scale-out* and *scale-in* thresholds as 60 % and 20 % of CPU usage, respectively.

In the experiment with MongoDB, MongoS instance is deployed on a large machine (*m3.large*), while the other instances are deployed on medium machines (*m3.medium*). In the experiments with ZooKeeper, every node is deployed on a medium machine (*m3.medium*)[4].

5.1.3 Workload Tools

To generate the workload for the experiment with MongoDB, we use the Yahoo Cloud Serving Benchmark (YCSB) [Cooper et al., 2010], while for the experiments with ZooKeeper, we use an open-source benchmark tool [Hunt et al., 2010].

5.1.4 Selected Bugs: Requirements for Reproduction

Table 4 summarizes the requirements for the reproduction of the three selected bugs. Those bugs cover all the possible combinations of requirements, constrained by the mandatory presence of *elasticity control*, and the need of at least one of

[2] `https://www.mongodb.org/`

[3] `https://zookeeper.apache.org/`

[4] `https://aws.amazon.com/fr/ec2/itype/`

the other requirements. We do not attempt to reproduce any bug that only requires *elasticity control*, since one could reproduce the required elastic behavior using Albonico et al. elastic control approach [Albonico et al., 2016] and then meets elasticity control requirement only.

	FEATURE		
BUG	Elasticity Control	Selective Elasticity	Event Scheduling
$MongoDB-7974$	*YES*	*YES*	*YES*
$ZooKeeper-2164$	*YES*	*YES*	*NO*
$ZooKeeper-2172$	*YES*	*NO*	*YES*

Table 4: Requirements for Reproduction the Three Selected Bugs

MongoDB bug 7974

This bug affects the MongoDB versions 2.2.0 and 2.2.2, when a secondary component of a MongoDB replica set[5] is deallocated. Indeed, in a MongoDB replica set, one of the components is elected as primary member, which works as a coordinator, while the others remain as secondary members.

To reproduce this bug, we must follow a specific elastic behavior: initialization of a replica set with three members, deallocation of a secondary member, and allocation of a new secondary member. Therefore, the second step of the elastic behavior requires the deallocation of a precise resource, one of the secondary members. The bug reproduction also requires two events synchronized to elasticity changes. Right after the secondary member deallocation, we must create a unique index, and after the last step of the elastic behavior, we must add a document in the replica set.

In conclusion, the reproduction of this bug needs to meet all the requirements that we consider in this paper: *elasticity control*, *selective elasticity*, and *event scheduling*.

ZooKeeper bug 2164

This bug is related to ZooKeeper version 3.4.5 and concerns the leader election. According to the bug report[6], in an ensemble with three nodes, when the node running the leader shuts down, a new leader election starts and never ends.

The reproduction of this bug must follow a precise sequence: initialization (allocation of the first node), followed by the allocation of two nodes and the deallocation of the leader node. The main difficulty of reproducing this bug is that when ZooKeeper is deployed on three nodes, the deallocated node is not necessarily the leader.

In conclusion, the reproduction of this bug needs to meet two requirements: *elasticity control* and *selective elasticity*.

ZooKeeper bug 2172

This bug is related to ZooKeeper version 3.5.0, which introduces the dynamic reconfiguration. According to the bug report[7], when a third node is added to a ZooKeeper ensemble, the system enters an unstable state and cannot recover.

After a thorough analysis of the available logs, we understand that the bug occurs when a leader election starts right after the allocation of a third node. More precisely, when a new node joins the ensemble, its synchronizes the configuration data with the leader. Therefore, if the data is not already synchronized at the moment of the leader election, the bug occurs.

The reproduction of this bug requires a simple elastic behavior: initialization and two node allocations. However, this sequence alone does not reproduce the bug: we need to be sure that the leader election starts before the end of the data synchronization process. We can force this by increasing the data amount through an event synchronized with the completion of the third node allocation.

The reproduction of this bug needs to meet two requirements: *elasticity control* and *event scheduling*.

5.2 Bug Reproduction

In this section, we describe the use of our approach to reproduce the three bugs, and compare the results to reproduction attempts without our approach. We do not explain in details the setup of reproductions without our approach but we assume one can manage the *control elasticity* and meet this requirement. Indeed, reproducing elasticity is a native feature of cloud computing infrastructures, and we just drive CBS through required elastic behavior using Albonico et al. approach [Albonico et al., 2016].

5.2.1 MongoDB-7974 Bug Reproduction

To reproduce MongoDB bug 7974 using our approach, we first manually create the MongoDB

[5] `https://docs.mongodb.com/replica-set`

[6] `https://issues.apache.org/jira/ZK2164`

[7] `https://issues.apache.org/jira/ZK2172`

replica set, composed by three nodes. Then, we set up the following sequence of elasticity changes, which should drive MongoDB through the required elastic behavior:

$$E = \langle ry_1, \langle 4500, r\rangle\rangle, \langle si_1, \langle 1500, r\rangle\rangle, \langle ry_2, \langle 3000, r\rangle\rangle, \langle so_1, \langle 4500, r\rangle\rangle, \langle ry_3, \langle 4500, r\rangle\rangle$$

Since we must deallocate a secondary member of MongoDB replica set at elasticity change ec_2, it is associated to a selective elasticity request (SER). The SER queries MongoDB replica set's members, using MongoDB shell method *db.isMaster*, until finding a member that is secondary.

In parallel to the elasticity changes, we set up two events, $e1$ and $e2$, which respectively create a unique index, and insert a new document in the replica set. The $e1$ is associated to elasticity change ec_3, a ready state that follows the scaling-in state where a secondary member is deallocated. The $e2$ is associated to elasticity change ec_5, the last ready state. Both events are scheduled without waiting time.

Elasticity Change	Event ID	Execution Sequence	Wait Time
ec_3	$e1$	1	0 s
ec_5	$e2$	1	0 s

Table 5: MongoDB-7974 Event Schedule

We repeat the bug reproduction for three times. After each execution, we look for the expression *"duplicate key error index"* in the log files. If the expression is found, we consider the bug is reproduced.

Table 6 shows the result of all the three executions, either using our approach or not. All the attempts using our approach reproduce the bug, while none of the attempts without our approach do it.

Reproduction	Reproduced	Not Reproduced
With Our Approach	3	0
Without Our Approach	0	3

Table 6: MongoDB-7974 Bug Reproduction Results

For the executions without our approach, we force MongoDB to elect the intermediate node (in the order of allocation) as primary member[8], what can occasionally occur in a real situation. In this scenario, independent of scale-in settings, cloud computing infrastructure always deallocate a secondary member, since Amazon EC2 only allows to deallocated the oldest or newest nodes. Despite in this experiment we force a selective elasticity, in a real situation without using our approach it is not deterministic. For instance, the newest or oldest node could be elected as a primary member. Even though cloud computing infrastructures reproduces the required elastic behavior, this bug still needs the event executions, which must be correctly synchronized. This is the reason the bug is not reproduced without our approach.

[8] `https://docs.mongodb.com/force-primary`

5.2.2 ZooKeeper-2164 Bug Reproduction

To reproduce this bug, we translate and complete the scenario (Section 5.1.4) into the following sequence of elasticity changes:

$$E = \langle ry_1, \langle 3000, r\rangle\rangle, \langle so_1, \langle 5000, r\rangle\rangle, \langle ry_2, \langle 5000, r\rangle\rangle, \langle so_2, \langle 10\,000, r\rangle\rangle, \langle ry_3, \langle 10\,000, r\rangle\rangle, \langle si_1, \langle 5000, r\rangle\rangle$$

The sequence of elasticity changes first initializes the cloud system with one node, then it requests two scale-out. Once the three nodes are running, the sequence requests a scale-in.

To discover the leader node, we write a SER that is associated to the last elasticity change $e6$ ($\langle si_1, \langle 5000, r\rangle\rangle$). The SER method connects to every Zookeeper node and executes ZooKeeper command named `stat`. This command describes, among other information, the node execution mode: leader or follower.

The sequence of elasticity states, including a selective elasticity, is supposed to reproduce the bug. To verify whether the failure occurs, we write a test oracle, which is implemented in JUnit [Gamma and Beck, 1999]. It is run after the last elasticity change ($\langle si_1, \langle 5000, r\rangle\rangle$), and repetitively searches for a leader until it is found or the timeout is reached. In the first case, the verdict is *pass*, what means the bug is reproduced and observed. Otherwise, the verdict is *fail*.

As well as in the first experiment, we use two different setups to execute this experiment: with our approach, and without our approach. We repeat the experiment three times for each setup.

Since the selective elasticity is one of the requirements for this bug reproduction, when executing without our approach, we try to reproduce a real scenario, where every node can be elected as a leader. Therefore, we force ZooKeeper to elect a different node as the leader at each execution: the newest, the oldest, then the intermediate node. Then, we use AmazonEC2 to deallocate a node. Its policy is to deallocate either the newest or the oldest node, it is not possible to deallocate the intermediate node. Hence, during the first two

executions we can ask AmazonEC2 to deallocate the leader, but not during the last one.

Table 7 summarizes the results. When using our approach, all the three test executions pass, demonstrating the ability of our testing approach to deterministically reproduce the bug. In contrast, only two executions without our approach pass, the ones where the leader is the newest or the oldest node. Therefore, without our approach the bug was not reproduced deterministically.

Reproduction	Pass Verdicts	Fail Verdicts
With Our Approach	3	0
Without Our Approach	2	1

Table 7: ZooKeeper-2164 Bug Reproduction Results

5.2.3 ZooKeeper-2172 Bug Reproduction

We create the following sequence of elasticity changes to reproduce this bug (Section 5.1.4):

$$E = \langle ry_1, \langle 3000, r\rangle\rangle, \langle so_1, \langle 5000, r\rangle\rangle, \langle ry_2, \langle 5000, r\rangle\rangle, \langle so_2, \langle 10\,000, r\rangle\rangle, \langle ry_3, \langle 10\,000, r\rangle\rangle$$

According to the bug log files, the bug occurs when the leader election starts before the end of the data synchronization between the third node and the previous leader. Thus, the test sequence must ensure that the data synchronization process is longer than the delay needed to start a new election, which is about 10 s according to the log files. Forcing the data synchronization to take long enough, we create an event schedule to associate an event $e1$ to the state so_2, as described in Table 8. The $e1$ requests a data increasing to an amount that should take longer than 10 seconds to synchronize. Since this experiment uses Amazon $m3.large$ machines, which have a bandwidth of 62.5 MB/s, the data amount must be ≈ 625 MB of data.

Elasticity Change	Event ID	Execution Sequence	Wait Time
ec_1	$e1$	1	0 s

Table 8: ZooKeeper-2172 Event Schedule

We use the test oracle as for the bug 2164, which is associated to the last *ready* elasticity state which is supposed not to be able to elect a leader before the timeout. Table 9 summarizes the experiment execution. In all three executions, the test verdict is *pass*, meaning that the testing approach reproduces the bug successfully. Since the AmazonEC2 cannot manage natively the scheduling of events synchronized with elasticity states, then it cannot reproduce the bug deterministically.

Reproduction	Pass Verdicts	Fail Verdicts
With Our Approach	3	0
Without Our Approach	0	3

Table 9: ZooKeeper-2172 Bug Reproduction Results

5.3 Threats to validity

In all the experiments, we set the cloud provider scale-out and scale-in thresholds to 60 % and 20 % of CPU usage, respectively. These values do not influence our experiments, though we seek for bug reproductions that require other thresholds.

In the experiments, our approach to drive CBS works fine. However, the elasticity driving could be compromised when the scalability is not linear.

Finally, controlling the elasticity directly could introduce some bias in the bug reproduction. For instance, one could anticipate an elasticity change, even before CBS enters a ready state. However, for the reproduced bugs we respect typical elastic behavior (see Figure 1).

6 Related Work

Several research efforts are related to our approach in terms of elasticity control, selective elasticity, and events scheduling. The work of Gambi et al. [Gambi et al., 2013b, Gambi et al., 2013a] addresses elasticity testing. The authors predict elasticity state transition based on workload variations and test whether cloud infrastructures react accordingly. However, they do not focus on controlling elasticity and cannot drive cloud application throughout different elasticity states.

Banzai et al. [Banzai et al., 2010] propose D-Cloud, a virtual machine environment specialized in fault injection. Like our approach, D-Cloud is able to control the test environment and allows testers to specify test scenarios. Test scenarios are specified in terms of fault injection and not on elasticity and events (as in our approach).

Yin et al. [Yin et al., 2013] propose CTPV, a Cloud Testing Platform Based on Virtualization. The core of CTPV is the private virtualization resource pool. The resource pool mimics cloud infrastructures environments, which in part is similar to our elasticity controller. CTPV differs from our approach in two points: (i) it does not use real cloud infrastructures and (ii) it uses an elasticity controller that does not anticipate resource demand reaction.

Vasar et al. [Vasar et al., 2012] propose a framework to monitor and test cloud computing web applications. Their framework replaces the cloud elasticity controller, predicting the resource demand based on past workload. Contrary to our approach, they do not allow to control a specific sequence of elasticity states or events.

Li et al. [Li et al., 2014] propose Reprolite, a tool that reproduces cloud system bugs quickly. Similarly to our approach, Reprolite allows the execution of parallel events on the cloud system and on the environment. Differently from our approach, Reprolite does not focus on elasticity, one of our main contributions.

7 Conclusion

In this paper, we proposed an approach to reproduce elasticity testing in a deterministic manner. This approach is based on three main features: elasticity control, selective elasticity, and event scheduling. We applied this approach successfully for reproducing three representative bugs from two popular open source systems: ZooKeeper and MongoDB.

Our approach allows the reproduction of bugs that could not be deterministically reproduced with state-of-the-art approaches. However, testing is not only about reproducing existing bugs, but also about detecting unknown ones. In this context, a likely evolution for our approach is to generate different test scenarios combining elasticity state transitions, workload variations, selective elasticity, and event scheduling.

Another feature we plan to investigate as future work is the speediness of test executions. Deterministic resource allocation can accelerate state transitions and thus optimize the number of executions per period of time, and/or reduce execution costs. This is particularly important when testing cloud systems: to perform our three experiments, we used 136 machine-hours in Amazon EC2 .

Finally, we also plan to apply model-driven engineering to create an unified high-level language. In the current implementation, the writing of test cases involves a mix of shell scripts, Java classes, and configuration files, which are not very suitable for users.

References

[Agrawal et al., 2011] Agrawal, D., El Abbadi, A., Das, S., and Elmore, A. J. (2011). Database scalability, elasticity, and autonomy in the cloud. *Proceedings of the 16th DASFAA*.

[Albonico et al., 2016] Albonico, M., Mottu, J.-M., and Sunyé, G. (2016). Controlling the Elasticity of Web Applications on Cloud Computing. In *The 31st SAC 2016*, Pisa, Italy. ACM/SIGAPP.

[Banzai et al., 2010] Banzai, T., Koizumi, H., Kanbayashi, R., Imada, T., Hanawa, T., and Sato, M. (2010). D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology. In *Proceedings of CCGRID'10*, Washington, USA.

[Bersani et al., 2014] Bersani, M. M., Bianculli, D., Dustdar, S., Gambi, A., Ghezzi, C., and Krstić, S. (2014). Towards the Formalization of Properties of Cloud-based Elastic Systems. In *Proceedings of PESOS 2014*, New York, NY, USA. ACM.

[Cooper et al., 2010] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of SoCC'10*, New York, NY, USA. ACM.

[Engstrom et al., 2010] Engstrom, E., Runeson, P., and Skoglund, M. (2010). A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30.

[Gambi et al., 2013a] Gambi, A., Hummer, W., and Dustdar, S. (2013a). Automated testing of cloud-based elastic systems with AUToCLES. In *The proceedings of ASE'13*, pages 714–717. IEEE/ACM.

[Gambi et al., 2013b] Gambi, A., Hummer, W., Truong, H.-L., and Dustdar, S. (2013b). Testing Elastic Computing Systems. *IEEE Internet Computing*, 17(6):76–82.

[Gamma and Beck, 1999] Gamma, E. and Beck, K. (1999). Junit: A cook's tour. *Java Report*.

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

[Hunt et al., 2010] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX, Boston, MA, USA, 2010*.

[Li et al., 2014] Li, K., Joshi, P., Gupta, A., and Ganai, M. K. (2014). ReproLite: A Lightweight Tool to Quickly Reproduce Hard System Bugs. In *Proceedings of SOCC'14*, New York, NY, USA.

[Thiery et al., 2014] Thiery, A., Cerqueus, T., Thorpe, C., Sunye, G., and Murphy, J. (2014). A DSL for Deployment and Testing in the Cloud. In *Proc. of the IEEE ICSTW 2014*, pages 376–382.

[Vasar et al., 2012] Vasar, M., Srirama, S. N., and Dumas, M. (2012). Framework for Monitoring and

Testing Web Application Scalability on the Cloud. In *Proc. of WICSA/ECSA Companion*, NY, USA.

[Yin et al., 2013] Yin, L., Zeng, J., Liu, F., and Li, B. (2013). CTPV: A Cloud Testing Platform Based on Virtualization. In *The proceedings of SOSE'13*.