HAL open science

A Scalable Benchmark as a Service Platform

Alain Tchana, Noel de Palma, Ahmed El Rheddane, Bruno Dillenseger, Xavier Etchevers, Ibrahim Safieddine

To cite this version:

Alain Tchana, Noel de Palma, Ahmed El Rheddane, Bruno Dillenseger, Xavier Etchevers, et al.. A Scalable Benchmark as a Service Platform. 13th International Conference on Distributed Applications and Interoperable Systems (DAIS), Jun 2013, Florence, Italy. pp.113-126, 10.1007/978-3-642-38541-4_9 . hal-00949561

HAL Id: hal-00949561

https://inria.hal.science/hal-00949561v1

Submitted on 14 Mar 2017

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.

Distributed under a Creative Commons Attribution 4.0 International License

A Scalable Benchmark as a Service Platform

Alain Tchana[1], Noel De Palma[1], Ahmed El-Rheddane[1], Bruno Dillenseger[2], Xavier Etchevers[2], and Ibrahim Safieddine[1]

[1] Joseph Fourier University, LIG Laboratory (UJF/LIG), Grenoble, France
firstname.lastname@imag.fr,
[2] Orange Labs, Grenoble, France
firstname.lastname@orange.com

Abstract. Load testing has always been a crucial and expensive activity for software companies. Classical solutions are a real burden to setup statically and their cost are prohibitive in terms of human and hardware resources. Cloud computing brings new opportunities to stress application scalability as load testing solutions can be provided on demand by the cloud. This paper describes a Benchmark-as-a-Service solution that scales automatically the load injection platform and eases its setup according to load profiles. Our approach is based on: (i) the virtualization of the Benchmarking platform to enable the injector's self-scalability, (ii) an online calibration mechanism to characterize injector capacity and impact on the benched application, (iii) a provisioning solution to scale the load injection platform sufficiently ahead of time. We also report experiments on a benchmark that shows the benefits in terms of cost and resources savings.

Keywords: Benchmarking as a service, Cloud

1 Introduction

Load testing has always been a very crucial and expensive activity for Internet companies. Traditionally, it leverages a load injection platform capable of generating traffic according to load profiles to stress an application, a system under test or SUT for short, to its limits. Such solutions are a real burden to setup statically and their costs are prohibitive in terms of human and hardware resources.

Cloud computing brings new opportunities and challenges to test applications' scalability since it provides the capacity to deliver IT resources and services automatically on a per-demand, self-service (APIs) basis over the network. One characteristic is its high degree of automation for provisioning and on-demand management of IT resources (computation, storage and network resources) and services. IT resources can be provisioned in a matter of minutes rather than days or weeks.

Opportunities lay in the fact that load testing solution can be provided on demand as a service on the cloud. Such Benchmark-as-a-Service (BaaS) solution enables quite a number of benefits in terms of cost and resources. The cost of

hardware, software and tools is charged on usage basis. The platform setup for the tests is also greatly simplified so that the testers can focus on their load injection campaign.

The challenge of Performance as a Service is to provide test teams with on-demand computing and networking resources, able to generate traffic on a SUT. Such test campaigns typically require more than a single load injection machine, to generate sufficient traffic (see Fig. 1). The issue is that the number of necessary load injection machines is not known in advance. It depends on the amount of resources consumed for generating and managing the requests and their responses, as well as on the target's global workload. The tester must empirically cope with these two risks:

- overloading the load injectors, causing scenarios not to behave as specified, and measures to be biased;
- wasting unnecessary resources.

For these reasons, we need a self-scalable load injection software making it possible to automatically adjust the number of load injection machines.

The contribution of this paper precisely addresses this challenge: We describe a BaaS solution (Section 3) which scales automatically the load injection platform. Besides the re-engineering of a load injection (Section 2) tool to enable self-scalability, the main concerns are (i) the injector's online calibration, (ii) the computation, based on from the load profile and the injector characterization, of the right amount of VMs and (iii) the control of their provisioning sufficiently ahead of time (Section 4 details these concerns). We also report experiments on the RUBiS [4] benchmark that shows the benefits in terms of self-scalability, including the cost reduction for long hours campaign (Section 5).

Section 6 and 7 present respectively the related work and the conclusion of this paper.

2 The CLIF load injection framework

This work has been achieved in the context of the CLIF load injection framework which is a versatile load testing, open source software [2]. It is generic and extensible, in terms of target SUT protocols as well as resources to monitor. A workload scenario combines the definition of one or several virtual user (vUsers) behaviors, with the specification of the number of active vUsers as a function of time, called the *load profile*. A *behavior* is basically a sequence of requests interlaced with *think times* (i.e. periods of pause), enriched with conditional and loop statements, as well as probabilistic branches. These behaviors make use of plug-ins to support a variety of features, mainly injection protocols (HTTP, FTP, SIP...) and external data provisioning for request parameters variability.

As described in details in [1], CLIF's architecture is based on the Fractal component model [3], which easies its adaptation. *Load injector* and *probe* components are distributed through the network. The formers are responsible for

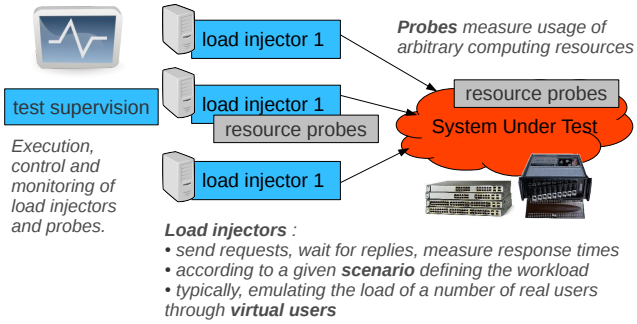

Fig. 1. The big picture of a load testing infrastructure

generating the workload and measuring response times, while the latter measure the usage of given resources: CPU, memory, network adapter or equipment, database, middleware, etc.

Load injectors and probes are bound to a central control and monitoring component, namely the *Supervisor*, and a central *Storage* component that will collect all measures once a test execution is complete (Fig. 1). They are deployed on local or remote hosts. All these components are contained in the top-level, distributed *Clif Application (ClifApp) composite* component. The component based developement of CLIF facilitates its adaptation. We present in the nexts sections the implementation a scalable load testing framework based on CLIF.

3 BaaS Overview

This section describes the main components and design principles of our self-scalable Benchmarking-as-a-Service Platform (BaaSP) based on CLIF. The main purpose of the BaaSP is to minimize the cost of achieving the test in a cloud environment. This cost mainly depends on the number of virtual machines (VMs) used and their up time throughout the test. Since each CLIF injector is running on a separate VM, BaaSP proposes a testing protocol which attempts to reduce both the number of VMs used and their execution time. This protocol relies on dynamic addition/removal of CLIF injectors according to the variation of the submitted load profile. Roughly, instead of statically using an over sized number of VM injectors, the BaaSP dynamically adds or removes injectors during the test as needed by the workload. Besides, the BaaSP attempts to use an injector up to its maximum capacity before adding another one. Let us now present the self-scaling protocol we implement in the BaaSP:

1. Initial VMs allocation and systems deployment in the cloud: The first step is the deployment and the configuration of the CLIF benchmarking system, possibly including the system we want to test (the SUT). This latter is optional since the SUT can be deployed and configured a long time before

the BaaSP, with another deployment system. This phase includes the VMs allocation in the cloud. Note that the cloud platfom which runs the BaaSP can be different from the one, if any, running the SUT.

2. Calibration and planning. The calibration phase aims at knowing the maximum capacity of an injector VM. This knowledge will then allow to plan when to add/remove injectors during the test.
3. Test execution and injectors provisioning. The actual test starts with a minimal number of injectors. The execution of these latter (request injection) should follow the submitted load profile. The BaaSP adds/removes injector VMs according to the planning done in the previous stage.
4. Systems undeployment and VM deallocation. This phase is opposite to the first one. At the end of the test, the BaaSP automatically undeploys and frees all the VMs it has instantiated in the cloud.

Figure 2 presents the BaaSP architecture. It is organized as follows. A VM (called BaaSPCore) is responsible of orchestrating the test: initialization of each test phase (deployment, calibration, test launching and undeployment). The Calibrator component is responsible for evaluating the capacity of an injector, while the Planner plans when to add/remove injectors VM during benchmarking.

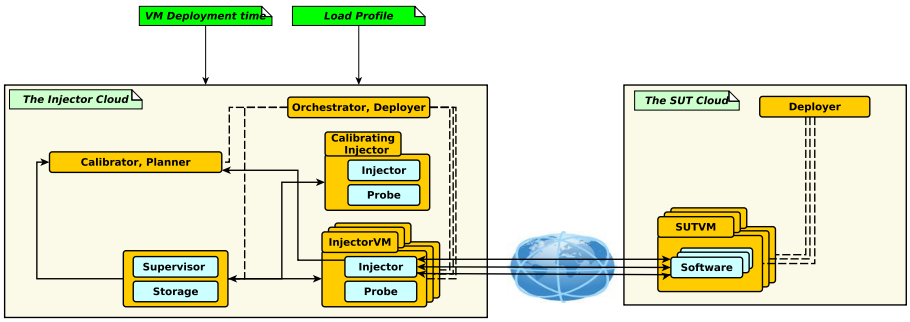

Fig. 2. Self-Scaling BaaS Architecture

4 Self-Scaling Protocol

4.1 Calibrating with CLIF Selfbench

This phase aims at evaluating the load injection capacity of an injector VM, in terms of greatest number of clients it is able to emulate (vUsers).

In order to evaluate the capacity of an injector, the Calibrator uses a CLIF extension module called Selfbench [5]. Selfbench results from research work on automating performance modelling of black boxes. Part of this work consists in

a self-driven workload ramp-up, looking for the maximum number of vUsers a SUT can serve until its resources are considered as insufficient.

Since Selfbench makes no assumption about the SUT capacity, it starts with a single vUser. From the response times and throughput the load injector gets, Selfbench computes the SUT's theoretical maximum capacity, with minimal assumptions in terms of parallel processing capability (single-threaded). Then, Selfbench increases the number of vUsers step-by-step, until reaching either the theoretical maximum capacity, or the SUT saturation limit. The number of steps is defined as a parameter. If the SUT is saturated, then the maximum number of vUsers it can serve has been reached. Otherwise, Selfbench makes a more optimistic assumption about the SUT's capacity, with a greater parallel processing capability, and runs a new step-by-step workload increase. The determination of steps duration combines theoretical results on queuing modeling and statistical considerations about the number of samples and their stability. The SUT saturation is defined as maximum or minimum thresholds on a number of load metrics, such as CPU usage, free memory or any other resource usage that a CLIF probe may monitor.

For the work we are presenting here, we use Selfbench in a slightly different way. The injector VM calibration is not based on detecting the SUT saturation but on detecting the injector VM saturation. Thus, the CLIF probes must be deployed at the injector VMs rather at the SUT side (even though it should be checked that the SUT is not saturating). At the end of its execution, Selfbench gives the number of vUsers reached before injector VM saturation (which represents the capacity of an injector).

4.2 Planning

Assuming that all injector VMs in the BaaSP have the same quantity of resources, then all injectors will have the same capacity (called InjMaxCapacity in the rest of this paper) as evaluated by the Calibrator. Based on this assumption and the time required to deploy injector VMs, the Planner is then able to plan injectors provisioning ahead of time for the given load profile. Let TTSVM be the deployment time function, which gives for a given number of VMs, the deployment time needed to start them in the cloud. This function is given by the operator of the BaaSP platform and depends on the cloud infrastructure utilized (In our case, we profiled our private cloud to configure this parameter as reported Section 4.3). The load profile (W) can be expressed as a discrete function of number of vUsers over the time: vUsers = W(t), means that the load profile requires "vUsers" to emulated the required workload at time t. Thus, the planning process can be expressed as a function: f(W,InjMaxCapacity,TTSVM). The Planner parses the load profile (W) and produces the provisioning rate (taking into account the deployment time TTSVM), according to the capacity of an injector VM (InjMaxCapacity). Thus, we are able to start injectors just ahead of time. In fact, instead of adding an injector at time t, the Planner will fire at time t - TTSVM. The planning algorithm returns a hash table VMAt (key,value)

where each "key" represents a time when the Planner should add/remove injectors.
Let VMAt[t1] = $InjAt_{t1}$ and VMAt[t2] = $InjAt_{t2}$, with t2 be the next key in VMAt following t1. If $InjAt_{t1} < InjAt_{t2}$, the Planner will add $InjAt_{t2}$ - $InjAt_{t1}$ injectors at time t2. Else, the Planner will remove $InjAt_{t1}$ - $InjAt_{t2}$ injectors. The algorithm we propose groups at a unique time, all injectors that will start in the same time frame. The time frame is defined as follows.

Another role of the Planner is to prepare the load profile which will be executed by added injectors during the benchmark. Alg. 1 is the algorithm used by the Planner to generate these load profiles. If Inj_Max represents the maximum number of injectors that can be simultaneously used during the test, the Planner generates Inj_Max of load profiles: W_i, $1 \leq i \leq Inj_{Max}$. The purpose of Alg. 1 is to generate all W_i. Then, when an injector "i" is added, it is configured to use a corresponding W_i. How a load profile is assigned to an injector is given by the algorithm. During the test, injectors which are running are ordered from 1 to currentNbInj, where currentNbInj is the current number of injectors. Thus, each injector i, $1 \leq i \leq CurrentNbInj$, runs the load profile W_i. When the Planner wants to add nbAdd of injectors, it sorts them from $currentNbInj + 1$ to $currentNbInj + nbAdd$. Each new injector j, $CurrentNbInj + 1 \leq j \leq CurrentNbInj + nbAdd$, will run the load profile W_j.

Finally, the Planner is implemented as a control loop. If "Timers" represents the set of keys (which are dates) of the hash table VMAT, then the Planner wakes up at each element in "Timers" and adjusts the number of injectors. The first entry of VMAt (VMAt[0]) represents the initial number of injectors, deployed before the beginning of the benchmarking process.

4.3 Injector dynamic provisioning

Injector addition protocol The Planner initiates the addition of new injectors. Fig. 3 (1) summarizes the protocol we implement:

(a) The Planner asks the Deployer to create a number of new injector VM required at a given time in the existing environment. This request contains the load profile that the injectors will run.
(b) The Deployer asks the IaaS to start each new required VM in parallel.
(c) Each new VM is equipped with a deployment agent which informs the Deployer that it is up.
(d) The Deployer sends to each new injector its configuration, including the load profile it will run.
(e) Each new injector registers its configuration by contacting the ClifApp.
(f) ClifApp integrates the new injector configuration in its injector list and forwards this configuration to its inner component (supervisor...). After this, the ClifApp requests the added injectors to start their load injection according to the profile.

Algorithm 1 : Injectors Workloads Planning

In
- VMAt[e]: Provisioning hash table
- W[t]: The benchmark workload

Out
- W_i[t]: Generated workloads, similar to W[t]

Begin

```
ForEach key e of VMAt do
  If (e is the last key of VMAt)
    exit
  End If
  next_e := next key of VMAt after e
  For i from e to next_e do
    For j from 1 to VMAt[e] do
      W_j[i] := W[i]/VMAt[e]
    End For
  End For
End For
```

End

Injector removal protocol Like with the previous protocol, the Planner initiates the removal of injectors. The removal protocol we implement is presented in Fig. 3(2):

(a) The Planner asks the Deployer to stop the execution of a number of injector VMs.

(b) The Deployer asks the ClifApp to unregister the injectors from the ClifApp injector list. The ClifApp forwards this reconfiguration to its inner components (supervisor...) and requests the corresponding injectors to stop their load injection.

(e) Once the injector has been unregistered from the ClifApp, the Deployer is notified by the ClifApp that the corresponding VMs are no longer taken into account.

(f) Finally, the Deployer asks the IaaS to turn off the VM hosting the injector.

After the calibration and the planning phases, there are two ways to go on with the test. If the SUT needs to be stabilized before being in a usable state, then the calibration phase is considered as the stabilizer phase. Thus, the real test will go on immediately after calibration. Otherwise, the SUT is restarted before launching the test. In both cases, the Planner adapts the initial number of injectors according to the load profile and the injector VM capacity. The next section is dedicated to the evaluation of this protocol.

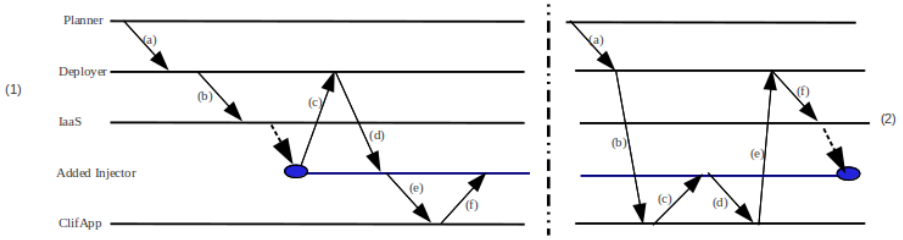

Fig. 3. (1) Example of adding an injector (initiated by the Planner); (2) Example of removing an injector (initiated by the injector itself)

5 Evaluation

5.1 Evaluation context

The System Under Test. The SUT is provided by RUBiS [4] (1.4.3 application version), a JEE benchmark based on servlets. RUBiS implements an auction web site modeled over eBay. It defines interactions such as registering new users, browsing, buying or selling items. For this evaluation, we submitted only browsing requests to the RUBiS application. We deploy the RUBiS open source middleware solution composed of: one Apache (2.2.14) web server (with Mod_JK 2 to connect to the application server), a Jakarta Tomcat (6.0.20) for servlets container (with AJP 13 as the connector), and a MySQL server (5.1.36) to host auction items (about 48 000 of items).

Cloud environment. Our experiments were carried out using the Grid'5000 [10] experimental testbed (the French national grid). Grid'5000 is organized in "sites" (a site represents a city), which in turn are organized in clusters. For our experiments, we configure two Grid'5000's clusters (Chicon at Lille, north of France; and Pastel at Toulouse, south of France) to provide separately the SUT cloud and the Injector Cloud (as shown in the BaaSP architecture in Figure 2). The two clusters run OpenStack [9] in order to provide virtualized cloud. The virtualization system is KVM version 2.0.0. We start each RUBiS VMs with 1GB of memory while injectors and the others BaaSP VMs used 256MB of memory. Each VM is pinned to one processor. They run the same operating system as the nodes which host them, which is Linux Ubuntu 10.04 distribution with a 2.6.30 kernel, over a gigabit connection. In this environment, we have calibrated the deployment time TTSVM. This time has an asymptotic behavior. For example, the deployment time of 1 VM until 10 VMs is the same (100s) while it grows up from 11VMs to 20VMs (with a difference of 75s). For readability, we use in this section TTSVM instead of TTSVM[i] for $1 < i < 10$.

5.2 Evaluation scenarios and metrics

Workload Scenarios Two workload scenarios have been experimented. These two workload scenarios summarize two situations corresponding to the worst case and a better case for our BaaSP system. Theoretically, each workload is designed to run in 1200 seconds.

The first workload scenario (Figure 4) represents a "simple" test workload W_s(t). This workload is composed of two phases: a ramp-up phase (W_s(nt)=nW_s(t)) followed by a ramp-down phase (W_s(nt)=$\frac{W_s(t)}{n}$), forming together a pyramidal workload. It needs addition/removal of single injector.

The second workload scenario (Figure 5) is more complex. It is composed of several kind of phases: gentler upward load, constant load, steep ramp-up load, steep ramp-down load, and gentle ramp-down load. **This kind of workload scenario shows how the BaaSP becomes more beneficial. In fact, unlike the first workload scenario, this second workload scenario needs sometime the addition/removal of more than one injector at once.**

The ultimate goal of our BaaSP is to minimize the cost of benchmarking an application in a cloud environment. Naturally, the main metric used in the evaluation is the cost of the test. We compare the cost of the test in two situations: static injectors deployment (called $Policy_0$) vs self-scaling injectors provisioning through our BaaSP. This second case was evaluated according to the following policies:

- $Policy_1$: injectors are dynamically added/removed without "just ahead time" provisioning.
- $Policy_2$: injectors are dynamically added/removed using a "just ahead time" provisioning strategy.

The cost of running the test in the cloud depends on both the duration of the test and the number of VM used during the test. In this three situations, the duration of the test includes: SUT and BaaSP startup, and the real test performing time. Calibrating time and injector addition/removal time are included when evaluating $Policy_1$ and $Policy_2$.

5.3 Evaluation results

The evaluation results we describe in this section are classified in two categories: quantitative and qualitative. Quantitative evaluation depicts the actual results we obtained regarding SUT and cloud test bed environment behaviors' during the test. Besides, we analyze the behavior of each provisioning policy during the test. Regarding qualitative evaluation, it concerns the formalization of observations coming from the quantitative evaluation. Before describing these results, let us present the variables on which the qualitative evaluation is based:

Let $Cost_{tu}$ be the cost of running a VM (as described earlier) in the cloud, during a time unit TU. Let TTS be the time to start both the SUT and BaaSP VM. Let TTT be the theoretical time to run the benchmark workload (20 minutes in our case). Let nbVMRubis be the number of VM used to run RUBiS. Let nbVMBaaSP be the number of VM used to run BaaSP components. Let nbInj

be the maximum number of injectors used during the test (5 in our case). The cost of running the test in the cloud without dynamic provisioning ($Policy_0$), noted $Cost_0$, is given by the following formula:

$Cost_0 = [nbInj * TTT + (nbVMRubis + nbVMBaaSP) * (TTT + TTS)] * Cost_{tu}$

Workload scenario 1 We compare the cost of $Policy_0$ to the two others policies: $Policy_1$ (injectors addition is done without a "just ahead of time" provisioning), and $Policy_2$ (injectors addition is done in a "just ahead of time" provisioning).

Figure 4(1) presents actual results of the execution of this first workload scenario. It shows two types of curves: (a) the variation of the CPU load of the RUBiS database tier (remember that it is the bottleneck of our RUBiS configuration), and (b) the injector provisioning rate during the test. These curves are interpreted as follows:

- The behavior of the SUT follows the specified workload (pyramidal). We observe that this workload saturates the MySQL node in term of CPU consumption (100%) at the middle of the load profile.
- The execution of the test with $Policy_1$ has a wrong behavior while $Policy_0$ and $Policy_2$ follow the same behavior, which corresponds to what we expected (according to the workload scenario). In fact, $Policy_1$ extends the test duration more than the theoretical duration specified in the workload scenario: about 400s, corresponding to range (c) shown in Figure4(1).
- We observe long stairs during the upward phase with $Policy_1$ compared $Policy_2$ because the deployment time of a new injector is not anticipated by $Policy_1$ as it is with $Policy_2$. We don't observe the same phenomenon in the lowering phase because injectors' removal is immediate.
- As shown in the curves (b) in Figure4(1), we use up to five injectors VM for each workload scenario.

Remember that TTSVM is the time used by the BaaSP to add an injector (about 100s in our experiment). Let TTCal be the time used by the Calibrator to calibrate an injector (60s in our experiment) and InjMaxCapacity be the maximum capacity of an injector (40 vuser in our experiment). With $Policy_1$, as we said that the test runs more than the theoretical time. This time corresponds to the sum of the deployment time of all injectors added by the Planner during the test, (nbInj-1)*TTSVM.

Let TTSI be the time needed to saturate an injector during the ramp-up phase (120s in our experiment). With $Policy_1$, TTSI+TTSVM is the provisioning period during the ramp-up phase whereas it is TTSVM in the ramp-down phase. On the other hand, TTSVM is the provisioning period with $Policy_2$ in both ramp-up phase and ramp-down phase. Figure 4(2) shows the execution time of each injector using $Policy_0$, $Policy_1$ and $Policy_2$. Here are the formulas used to evaluate the cost of these different cases. Let $ExecTime_i$ be the execution time of injector i, and $ExecTime_{RubisBaaSP}$ be the execution time of RUBiS and BaaSP VM.

- $Cost_1 = Cost_0 + (\frac{nbInj(nbj-1)}{2}(TTSVM - 2TTSI) + TTCal + (nbVMRubis + nbVMBaaSP)[TTCal + (nbj-1)TTSVM]) * Cost_{tu}$ **(E1)**
- $Cost_2 = Cost_0 + [(nbVMRubis + nbVMBaaSP + 1)TTCal + nbInj * TTSVM - 2 * nbInj(nbj-1)TTSI] * Cost_{tu}$ **(E2)**

Regarding equations (E1) and (E2), dynamic provisioning is less expensive than static execution when:

- $Policy_1$: $TTSI \geq \frac{(nbVMRubis + nbVMBaaSP + 1)TTCal}{nbInj(nbInj-1)} + \frac{(nbInj/2 + nbVMRubis + nbVMBaaSP)(nbInj-1)TTSVM}{nbInj(nbInj-1)}$ (C1)
- $Policy_2$: $TTSI \geq \frac{(nbVMRubis + nbVMBaaSP + 1)TTCal}{2 * nbInj(nbInj-1)} + \frac{TTSVM}{2(nbInj-1)}$ (C2)

When conditions (C1) and (C2) are respected, $Cost_0 \leq Cost_1 \leq Cost_2$. In our experimental environment, we have: nbInj=5, nbVMRubis=3, nbVMBaaSP=2, TTS=250s, TTCal=60s, TTSVM=100s, and TTSI=120s. In this context, (C1) is not met whereas (C2) is. Then, $Cost_0 = 13250 * Cost_{tu}$, $Cost_1 = 14210 * Cost_{tu}$, and $Cost_2 = 9310 * Cost_{tu}$.

Workload scenario 2 Fig.5 shows results for the second workload scenario. The interpretation of these results is similar to the previous workload scenario. Unlike the first scenario, we only evaluate the BaaSP when the "just ahead of time" provisioning is activated. Looking at the workload of this scenario, curve (a) of Fig. 5(1) shows injectors provisioning:

- An injector is added at time T1.
- Three injectors are simultaneously added at time T2. This is done according to the steep ramp-up phase occurs for a short time (from time 550s to 600s), which is less than TTSVM (refer to the planning algorithm).

Fig. 5(2) shows the execution time of this experiment (Fig. 5(2)(b)) in comparison to the static execution (Fig. 5(2)(a)). From the same variables we used in the previous section, the cost of this experiment ($Cost_3$) is evaluated as follows:

$Cost_3 = [(nbVMRubis + nbVMBaaSP) * (TTT + TTS + TTCal) + \sum_{i=1}^{nbInj} ExecTime_i] * Cost_{tu}$

$Cost_3 = 10570 * Cost_{tu}$

The value of $Cost_3$, in comparison to $Cost_0$ (which is always fix), shows that for some scenarios (long test campaign in preference), the gain of using dynamic injector provisioning becomes more interesting.

6 Related work

Very few works are interesting on adaptive benchmarking tools. However, we study some work situated around this topic. Unibench [12] is an automated benchmarking tool. As our BaaSP, Unibench is able to deploy remotely both the SUT and benchmarking components in a cluster. It is adaptive in the way that

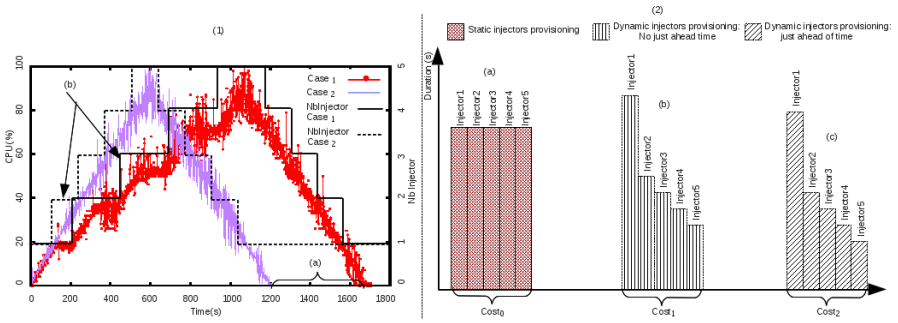

Fig. 4. Injectors provisioning in the BaaSP when running the first scenario

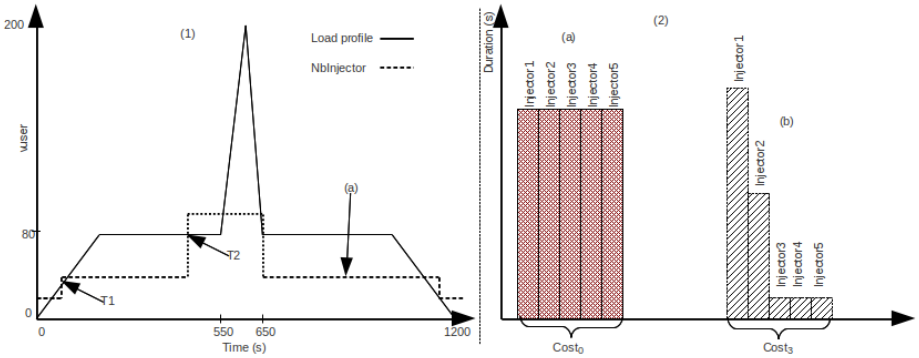

Fig. 5. Injectors provisioning in the BaaSP when running the second scenario

it is able to identify a modification in the SUT and then achieve another benchmarking process related to this modification. To do this, Unibench is supposed to know the source code and the programing language of the SUT. Unlike our BaaSP, the SUT is not considered as a black box. [13] presents research challenges for implementing benchmarking tools for self-adaptative systems. Except the definition of metrics and some principles to be considered when defining workload, it does not care about the self-adaptation of the benchmarking tool itself.

CloudGauge [14] is an open source framework similar to ours. It uses the cloud environment as the benchmarking context. Unlike our BaaSP which evaluate a SUT running in the cloud, CloudGauge SUTs' is the cloud and its capability for VM consolidation. It dynamically injects workloads to the cloud VM and adds/removes/migrate VM according to the fluctuation of the workload. As Selfbench [5] (the calibration system we used), it is able to adjust itself the workload during the benchmarking process. Indeed, like our BaaSP, users can define a set of workloads for benchmarking. Since the SUT is the cloud, injectors are deployed into VM. There is no separation between injectors nodes and SUT

nodes. Thus, unlike our BaaSP, there is no need to dynamically create injectors node as injectors and SUT share the same VMs. Regarding the architecture of CloudGauge, we observe some similarity with ours. For example, CloudGauge defines an orchestrator called *Test Provisioning* which is responsible to orchestrate the benchmarking process. Other tools such as VSCBenchmark [15] and VMark [16] are comparable to CloudGauge. They allow the definition of dynamic workload for VM consolidation benchmarking in a cloud environment.

As far as we know, there is no open source benchmarking framework with comparable characteristic as ours. However, there is some proprietary and commercial tools. Looking at the marketing speech of BlazeMeter [17], it provides same features as ours (except SUT deployment): dynamic injectors allocation and de-allocation in the cloud in order to reduce test cost. It is an evolution of the JMeter [11] tool for cloud platform. Since it is proprietary, there is no technical and scientific description of BlazeMeter. Therefore, it becomes difficult to really compare its functionalities to ours. NeoLoad [18] is another tool similar to BlazeMeter. It allows deployment of injectors in a cloud environment for benchmarking an application. It is able to integrate new injectors throughout the benchmarking process. However, this integration should be initiated by the administrator by planning. NeoLoad does not implement itself an automated planning component, as we did.

7 Conclusion

This paper explores Cloud Computing features to ease application benchmarking and to stress their scalability. Load testing solution can be provided on demand in the cloud and can benefit from self-scalability.

We describe a Benchmark-as-a-Service solution that provides a number of benefits in terms of cost and resources savings. The cost of hardware, software and tools is charged on a pay-per-use basis and platform setup is also greatly simplified. The self-scalability property of the platform eases the benchmarking process and lower the cost for long hours campaign since it does not require to statically provision the platform which is prohibitive in terms of human and hardware resources. Resource provisioning is minimized while ensuring load injection according to a given profile. Our experiments based on the RUBiS benchmark show the benefits in terms of cost reduction for long hours testing campaigns. As for as we know, our Benchmark-as-a-Service platform is the only one that scales automatically the resources used for load injection.

As a future work we plan to add a new mode to our platform. With this mode, load profiles are not required any more. It aims at automatically provisioning and controlling load injection resources until saturating the SUT. The difficulty here is to stress progressively an application near its limits while preventing thrashing. This requires a fine grain load injection control and provisioning.

Acknowledgment

This work is supported by the French Fonds National pour la Societe Numerique (FSN) and Poles Minalogic, Systematic and SCS, through the FSN Open Cloudware project.

References

1. B. Dillenseger, *CLIF, a framework based on fractal for flexible, distributed load testing*, in Annals of Telecommunications, Vol. 64, Numbers 1-2, issue 1, Pp. 101-120, Pub. Springer Paris, 2009.
2. *The CLIF Project*, in http://clif.ow2.org, visited on 2013, February.
3. E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani, *An Open Component Model and Its Support in Java*, in International Symposium on Component-Based Software Engineering (CBSE), Edinburgh, UK, 2004.
4. C. Amza, E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel, *Specification and implementation of dynamic web site benchmarks*, in IEEE Annual Workshop on Workload Characterization, Austin, TX, USA, 2002.
5. A. Harbaoui, N. Salmi, B. Dillenseger, and J. Vincent, *Introducing queuing network-based performance awareness in autonomic systems*, in ICAS, Cancun, Mexico, 2010.
6. A. Tchana, S. Temate, L. Broto, and D. Hagimont, *Autonomic resource allocation in a J2EE cluster*, in Utility and Cloud Computing, Chennai, India, 2010.
7. Amazon Web Services, *Amazon EC2 auto-scaling functions*, in http://aws.amazon.com/fr/autoscaling/, visited on 2013, February.
8. *Righscale web site*, in http://www.rightscale.com, visited on 2013, February.
9. *Openstack web site*, in http://openstack.org/, visited on 2013, February.
10. *Grid'5000 web site*, in https://www.grid5000.fr, visited on 2013, February.
11. The Apache Software Foundation, *Apache JMeter*, in http://jmeter.apache.org/, visited on 2013, February.
12. D. Rolls, C. Joslin, and S.-B. Scholz, *Unibench: a tool for automated and collaborative benchmarking*, in ICPC, Braga, Portugal, 2010.
13. R. Almeida and M. Vieira, *Benchmarking the resilience of self-adaptive software systems: perspectives and challenges*, in SEAMS, Waikiki, Honolulu, HI, USA, 2011.
14. M. A. El-Refaey and M. A. Rizkaa, *CloudGauge: a dynamic cloud and virtualization benchmarking suite*, in WETICE, Larissa, Greece, 2010.
15. H. Jin, W. Cao, P. Yuan, and X. Xie, *VSCBenchmark: benchmark for dynamic server performance of virtualization technology*, in IFMT, Cairo, Egypt, 2008.
16. V. Makhija, B. Herndon, P. Smith, L. Roderick, E. Zamost, and J. Anderson, *VMmark: a scalable benchmark for virtualized systems*, Technical Report VMware-TR-2006-002, Palo Alto, CA, USA, September 2006.
17. BlazeMeter, *Dependability benchmarking project*, in http://blazemeter.com/, visited on 2013, February.
18. Neotys, *NeoLoad: load test all web and mobile applications*, in http://www.neotys.fr/, visited on 2013, February.