Self-scalable Benchmarking as a Service with Automatic Saturation Detection

Alain Tchana[1], Bruno Dillenseger[2], Noel De Palma[1], Xavier Etchevers[2], Jean-Marc Vincent[1], Nabila Salmi[2], and Ahmed Harbaoui[2]

[1] Joseph Fourier University, LIG, Grenoble, France
first.last@imag.fr
[2] Orange Labs, Grenoble, France
firstname.lastname@orange.com

Abstract. Software applications providers have always been required to perform load testing prior to launching new applications. This crucial test phase is expensive in human and hardware terms, and the solutions generally used would benefit from further development. In particular, designing an appropriate load profile to stress an application is difficult and must be done carefully to avoid skewed testing. In addition, static testing platforms are exceedingly complex to set up. New opportunities to ease load testing solutions are becoming available thanks to cloud computing. This paper describes a Benchmark-as-a-Service platform based on: (i) intelligent generation of traffic to the benched application without inducing thrashing (avoiding predefined load profiles), (ii) a virtualized and self-scalable load injection system. This platform was found to reduce the cost of testing by 50% compared to more commonly used solutions. It was experimented on the reference JEE benchmark RUBiS. This involved detecting bottleneck tiers.

Keywords: Benchmarking as a service, Saturation detection, Cloud.

1 Introduction

Software applications providers have always been required to perform load and performance testing. This crucial activity is expensive in both human and resource terms. Traditionally, testing leverages a platform capable of generating enough traffic to stress a System Under Test (SUT), and thus determine its limits. This stress aims to detect the maximal throughput for a distributed system while ensuring an acceptable response time, to determine where bottlenecks lie or how performance is affected by adjusting configuration parameters.

To generate an appropriate level of traffic, commonly used solutions are based on load profiles designed by testers, using empirical expertise. Profiles should be designed to stress the system without trashing it. As IT systems become increasingly complex and distributed, the task of designing an appropriate load profile has become increasingly difficult.

In addition to this increasing difficulty, static testing platforms are exceedingly complex to set up, and are prohibitively costly in terms of human and hardware

D. Eyers and K. Schwan (Eds.): Middleware 2013, LNCS 8275, pp. 389–404, 2013.

resources. A typical test campaign requires several load injection machines to generate enough traffic to the SUT (see Fig. 1), but the number of necessary load injection machines is not known in advance.

To overcome this uncertainty, the injection machines are generally statically provisioned, with the risk of encountering resource shortage or waste. In summary, the tester must empirically cope with two risks:

- provisioning too few load injection machines, which may distort the benchmarking results;
- provisioning too many load injection machines, resulting in useless expenses.

Test system scalability may benefit greatly from the opportunities presented by cloud computing, through its capacity to deliver IT resources and services automatically on-demand, as part of a self-service system. Cloud computing allows IT resources to be provisioned in a matter of minutes, rather than days or weeks. This allows load testing solutions to be developed on-demand as a service on the cloud. This type of Benchmark-as-a-Service platform (BaaSP) provides significant benefits in terms of cost and resources as hardware, software and tools are charged for on a per-use basis. The platform setup for the tests is also greatly simplified, allowing testers to focus on analyzing test campaign results. This paper describes a BaaSP, Benchmark-as-a-Service platform, that:

- returns the maximum number of virtual users a system under test (SUT) can serve, while satisfying operational constraints (e.g. avoid CPU or memory saturation) as well as quality of service constraints (e.g. acceptable response time). This set of constraints is referred to as the *saturation policy*.
- automates load injection until saturation is reached. This eliminates the need for a predefined load profile.
- automates provisioning of load injection virtual machines when necessary, avoiding resource over-booking for BaaSP itself. For detecting when a new virtual machine is required for load injection, the platform uses the same mechanism as the one used to detect SUT saturation.

We tested our platform by stressing a JEE benchmark (RUBiS [1]) to determine the bottleneck tiers. Our solution halved the cost of the test (in terms of VMs uptime) compared to a statically provisioned testing platform. The results for this particular use case show that the data base tier is the bottleneck when receiving a browsing workload. These results are consistent with previous research [13].

Section 2 of this article presents the load injection framework our work is based on for traffic generation. Section 3 describes the BaaSP's architecture, while section 4 details its design. Section 5 presents the experiments performed as part of this study; section 6 related work; and section 7 concludes our study.

2 The CLIF Load Injection Framework

This work builds on previous work [4] based on the CLIF load injection framework [2]. The CLIF open source project provides Java software to define, deploy

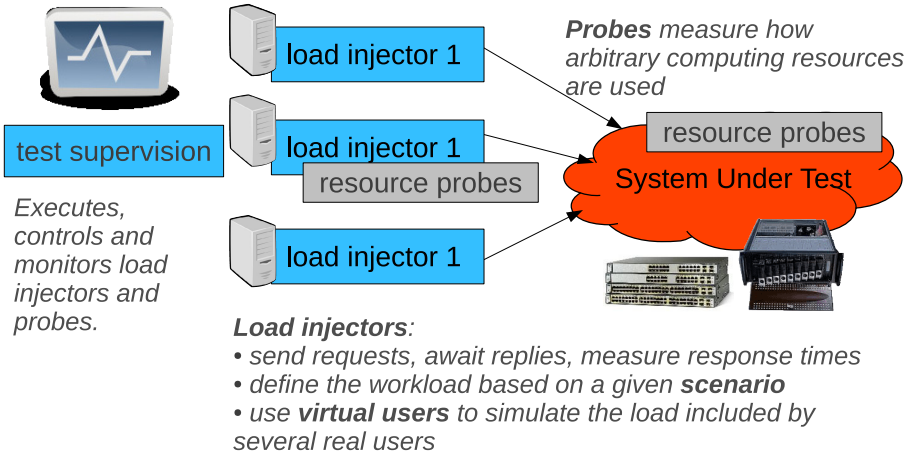

Fig. 1. Overview of the load testing infrastructure

and run performance tests on many kinds of SUT. A CLIF workload is specified through a scenario defining the traffic generated by each load injector. A scenario defines one or several virtual user (*vUser*) behaviors, and the number of active *vUsers* of each behavior over time. This is known as the *load profile*. A *behavior* is basically a sequence of requests separated by *think times* (i.e. pauses), but it can be enriched by adding conditional and loop statements, as well as probabilistic branches. Beyond this logical constructs, behaviors make use of plug-ins to support a variety of features, mainly injection protocols (HTTP, FTP, SIP...) and external data provisioning to enable variability in request parameters.

As shown by figure 1, a CLIF test consists of a number of distributed components: *load injectors*, to generate traffic, and *probes*, to monitor resources usage. Each injector contributes to the global workload by executing a scenario, and measures the response time of each generated request. Probes measure how given resources (CPU, memory, network adapter or equipment, database, middleware, etc.) are used. Injectors and probes are bound to a central test control and monitoring component, the *Supervisor*, and a central *Storage* component which gathers all measurements upon test completion. The next section describes how we adapted CLIF to design the BaaSP platform.

3 Architecture Overview

BaaSP is based on the CLIF load injection framework; it automatically and dynamically drives the number of active virtual users to test a system's capacity. Starting with a minimal load, i.e. a single virtual user, BaaSP increases the number of virtual users step-by-step until the saturation policy is violated. Step duration and increment levels are defined by an *injection policy*, which is computed from a live queuing model equivalent to the SUT. This protocol relies on

the dynamic addition of injectors once current injector VMs become saturated. This dynamic addition avoids static injector dimensioning. Fig. 2 presents the architecture of BaaSP, the main elements of which are:

- *Deployer*. The deployer automates VM allocation in the Cloud and deploys and configures all BaaSP and SUT components. The SUT can also be deployed and configured long before the BaaSP, through an independent deployment system. The cloud platform running the BaaSP can be different from that running the SUT (which may not be run on a cloud platform at all).
- *Injectors and Probes*. Injectors implement the requested injection mechanism. They also compute the response time and throughput metrics for the SUT, which can be used to provide statistics relating to its state. Probe components monitor the injector VMs and provide information relating to their saturation.
- *InjectionController*. The *Injection Controller* implements the injection policy. The injection controller examines the state of the SUT, based on injector statistics, and decides how much and when to increase vUsers and to dispatch them to the injector VMs. A delicate balance must be maintained, with stress applied progressively to the application. The ideal level of stress is near the application's limit, but without causing application trashing.
- *InjectorsSaturationController*. The injector saturation controller monitors saturation of injector VMs (via their associated probes) and triggers the addition of new injector VMs in line with the saturation policies.
- *SutSaturationController*. The SUT saturation controller monitors SUT saturation using injector statistics and based on the SUT's saturation policies. When the SUT becomes saturated, the SUT saturation controller returns the number of vUsers causing saturation, the maximum throughput, the average response time and the resource consumption. The next section provides details on how our solution was designed.

4 Automated Load Injection Design

The self-regulated load injection approach we are using in this work comes from our earlier results [4] and has been extended with dynamic provisioning of injection virtual machines as described in this section.

4.1 Injection Policy

On start-up, BaaSP makes minimal assumptions about SUT performance, applying a single virtual user workload and observing requests response times and throughput. Then, as illustrated on Fig. 3, the virtual user population is increased step-by-step. Each step is characterized by:

- the number of virtual users to add (*injection step*);

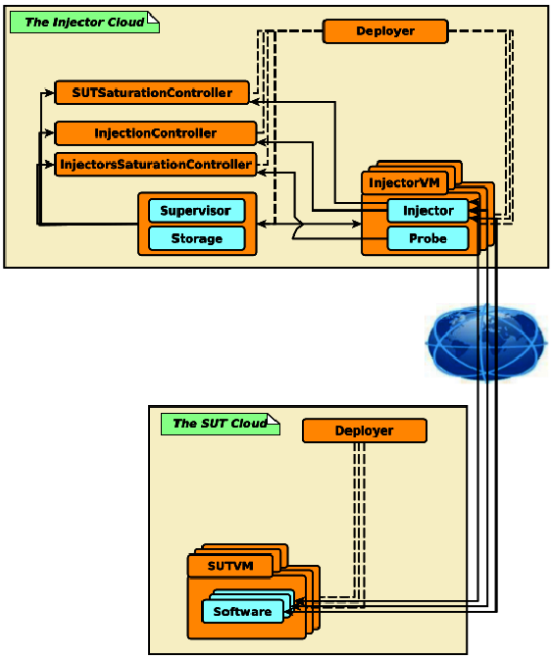

Fig. 2. The BaaSP Architecture

– the step duration, which can be subdivided into a *stabilization time* and a *sampling time*. The stabilization time begins with a fixed *ramp-up delay*, which allows these virtual users to be added progressively.

The ramp-up delay is used to avoid stressing the SUT and the load injectors. Excessive stress and trashing could be induced by a sudden increase in workload. BaaSP aims to determine the maximum number of virtual users possible in stable workload conditions. As part of this, response time measurements are discarded during the ramp-up to ignore transitional effects. Similarly, measurements are also discarded during stabilization. The initial stabilization time is given as a parameter, but subsequent stabilization times are computed for each step by estimating the convergence time for the Markov chain [5], [6] underlying the queue model representing the SUT.

The sampling time is the period for measuring response times. Because of networking and computing overload threats, it is not possible to get all response time measurements during test run-time. Instead, BaaSP relies on CLIF's ability to deliver moving statistics on load injectors and probes. Over a polling period, the injection controller obtains the continuous statistics from the injectors: mean and standard deviation for response times, and the total number of requests issued. Using these numbers, the injection controller periodically assesses the significance of the statistical sample by combining the following two criteria: (1) the minimal number of requests issued (given as a parameter), and (2) the

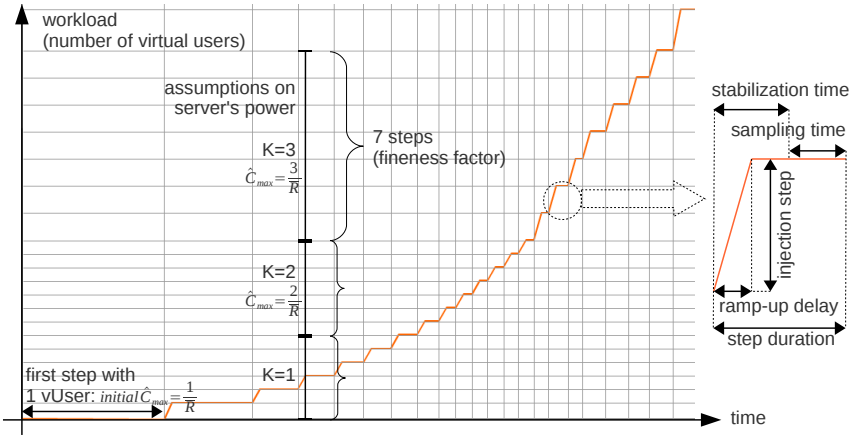

Fig. 3. Queue model-based automatic control of load injection

stability of measurements, based on a formula derived by Jain et al. [7]. This formula can be used to calculate the sample size required to achieve a given level of accuracy and confidence interval. We apply this formula, given the mean and standard deviation values determined by CLIF's moving statistics.

When the sampling time is complete for the current step, BaaSP estimates the queue model parameters, as explained in the next section.

4.2 Estimation of Maximal Load

The Kendall notation [5] of an elementary queuing system, denoted by $T/X/K$, was used in our estimation of the maximal load permissible for a SUT. In this notation, T indicates the distribution of the inter-arrival times, X the service times distribution, and K the number of servers ($K \geq 1$). The number of servers is representative of the parallel processing capability, which is bound to, but not predefined by, the number of physical processing cores. To simplify calculations, K is generally considered to be an integer number, although in reality it is more likely to be a decimal number.

The maximal supported load, $\hat{C}_{max}$, is first estimated from an initial load injection phase assuming a minimal value of 1 for K. During this phase, the SUT is loaded with markovian interarrival requests from a single virtual user, and statistics on response times R are polled from the load injectors. When a single client arrives in an empty queue, there is no concurrence and the waiting time is zero. In such conditions, the service rate μ, i.e. the server's maximum throughput in terms of requests processed per second, equals $\frac{1}{\overline{R}}$, where $\overline{R}$ is the mean response time. Assuming $K = 1$, the first estimation of $\hat{C}_{max}$ equals μ.

With an M/G/K model, the rate of request arrivals converges to $K * \mu$. If the SUT becomes saturated, then the previous assumption on K is confirmed. BaaSP stops load injection and returns the number of active virtual users of the

latest step completed without saturation. But, when $\hat{C}_{max}$ is reached without saturating the server, the assumed value of K is incremented, and the assumption on $\hat{C}_{max}$ is upgraded to $K * \mu$, with $K = 2, 3, 4 \ldots$ for subsequent iterations.

When BaaSP upgrades the assumed value of K, the workload increase towards the new target $\hat{C}_{max}$ is split into a number of steps, determined by a BaaSP parameter in the benchmark policy: the *fineness factor* f. The *injection step* is the number of new virtual users to add to go for the next step. The injection step equals $\frac{\hat{C}_{max}}{f*\lambda}$, where λ is the mean request throughput issued by each virtual user. Greater values for f will give more accurate results but will result in longer experience time.

4.3 Dynamic Injector Provisioning

As shown for the BaaSP architecture (Fig. 2), all injector VMs are equipped with monitoring probes. The InjectorsSaturationController is configured with one or more saturation policies based on information gathered by the monitoring probes. As presented above, a saturation policy takes the form of a threshold policy. For example, the CPU load of the injector VM should be below 100%. Thus, the InjectorsSaturationController periodically compares the information it receives with saturation policies. When a policy is violated, a new injector VM will be added. The injection provisioning protocol is summarized in Fig. 4 and can be interpreted as follows:

(a) The InjectorsSaturationController asks the Deployer to create a new injector VM. The InjectorsSaturationController then disables its saturation detection process (to avoid further new additions before the current one has been treated).
(b) The Deployer asks the IaaS (cloud) to start a new VM.
(c) The VM is equipped with a deployment agent which informs the Deployer that it is started.
(d) The Deployer sends its configuration to the new injector.
(e) The injector registers its configuration by contacting the Supervisor.
(f) The Supervisor integrates the new injector configuration in its injector list and forwards this configuration to its inner component. The Supervisor then requests that the injectors added start their load injection (by setting the SUT URI). The Supervisor also informs the InjectionController of the presence of a new injector. The InjectionController registers the new injector and dispatches the number of vUsers equally between injectors.
(g) Finally, the InjectionController tells the InjectorsSaturationController to re-enable its saturation detection process.

This protocol was successfully applied in several use cases, which are presented in the next section.

4.4 BaaSP Cost Benefit

As we mentioned in Section 1, one of the main contribution of our platform is the reduction of the cost of the test over the cloud (in terms of VMs uptime)

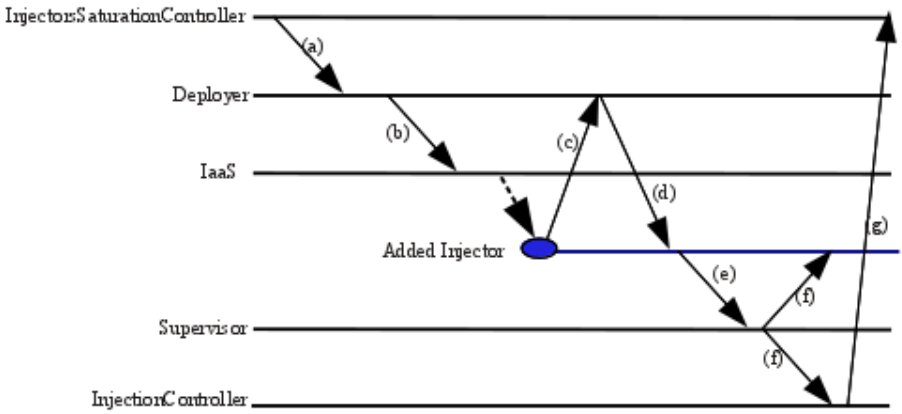

Fig. 4. Injector VM addition process

compared to a statically provisioned testing. The cost of running BaaSP in a commercial cloud (such as Amazon EC2) includes several parameters: the number of VM, their uptime, their type, outgoing network traffics, the number of IO operations, etc. Therefor, modeling the cost of the test should consider different circumstances, including the location of the injection system towards the SUT. We limit our evaluation to one circumstance: the SUT and BaaSP are on the same cloud. For comparison, we also consider that the static load injection tool runs in the same cloud as the BaaSP and they use the same type of VM. With statically provisioned injectors, the number of injector VMs is constant throughout the test. For our comparison, we assume that this number is the total number of VM instantiated by the BaaSP (i.e. the tester choose exactly the right maximum number of VM, which is extremely unusual). The cost of the test in this case (noted Cost0) can be calculated by the following formula:

$Cost_0 = nbInjVM * TestUpTime * Cost_{tu}$ (Equation 1),

where $nbInjVM$ is the total number of injector VMs, $TestUpTime$ is the duration of the test, and $Cost_{tu}$ is the cost of running a VM in the cloud for a given unit of time.

With scalable injector provisioning, the execution time of the i-th injector VM is about $\frac{TestUpTime*(nbInjVM-i+1)}{nbInjVM}$. Thus, the cost of a test in this case is $Cost_{BaaSP} = [\sum_{i=1}^{nbInj} \frac{TestUpTime*(nbInjVM-i+1)}{nbInjVM}] * Cost_{tu}$, which corresponds to $Cost_{BaaSP} = [\frac{TestUpTime*(nbInjVM+1)}{2}] * Cost_{tu}$ (Equation 2).

Regarding (Equation 1) and (Equation 2), **the cost of the test is halved when using our platform**. Notice that the evaluation of $Cost_0$ is the most optimistic one since we don not consider the possible repetitions of the test to determine the appropriate workload. Also we assume that the tester in that case does not overestimate the number of VMs required for the test.

5 BaaSP Use Cases: Benchmarking a JEE Application

We tested how well our solution could determine the bottleneck tiers of a JEE application, and tune it to improve performance.

5.1 Experimental Context

System Under Test. We tested RUBiS [1], a JEE benchmark which implements an auction web site modeled on eBay. It defines interactions such as registering new users, browsing, buying and selling items. This application is deployed on a load-balanced architecture composed of virtual machines providing the following middleware: Apache Tomcat (7.0) as servlet container, and a MySQL server (5.1.36) to host auction items (about 48 000 items). We used a HAProxy load balancer in front of Tomcat servers when several Tomcat servers were tested. The MySQL-Proxy load balancer was also used. To remain within the allowable page length for this article, this paper oly presents experimental results for browsing requests. Fig. 5 summarizes the architecture of these applications.

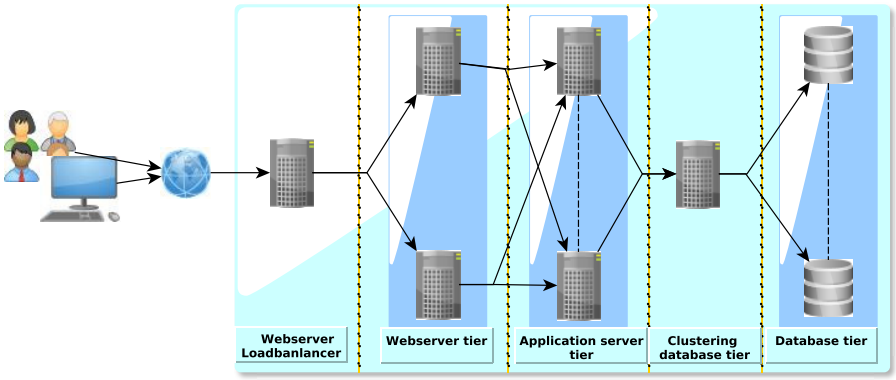

Fig. 5. Architecture of a JEE application

Cloud Environment. Our experiments were carried out using the Grid'5000 [10] platform, which is composed of clusters in different areas of France. We used two Grid'5000's clusters (Chinqchint and Chicon) to deploy the SUT and the injection platform separately. The two clusters run OpenStack [11] and Xen hypervisors (version 3.2) to set up a virtualized cloud providing VMs with configurations similar to the Amazon EC2 [12] Small one. All VMs run the same operating system as the nodes which host them: Linux Ubuntu 10.0.4 distribution with a 2.6.30 kernel, over a gigabit connection. Fig. 6 summarizes the cloud environment configuration.

	Cluster Chinqchint, Represents the SUT cloud	Cluster Chicon, Represents the injector cloud
Model	SGI Altix Xe 310	IBM eServer 326m
CPU	Intel Xeon E5440 QC 2.83 GHz / 4 MB / 1333 MHz (2cpus, 4cores/cpu)	AMD Opteron 285 2.6 GHz / 1 MB / 800 MHz (2cpus, 2cores/cpu)
Memory	8 GB	4 GB
Network	cards Myri-10G (10G-PCIE-8A-C) Gigabit Ethernet	Myri-10G (10G-PCIE-8A-C) Gigabit Ethernet
Storage	SATA 2(driver: ahci)	SATA 2(driver: sata_nvw)
OS	Linux Ubuntu 10.0.4 distribution with a 2.6.30 kernel	
IaaS	OpenStack (StackOps), version 0.4-b1262-d20120223	
	Xen version 3.2, Credit scheduler, network in bridge mode	
Deployed Servers	RUBiS severs	BaaSP with Injectors
The two sites are linked with 10Gb/s dark fibers		
VMs	3Gb of disk, 1740Mb of Memory, 1 vcpu	

Fig. 6. Configuration of our experimental cloud platform (Grid'5000)

Experimental Objectives. In these experiments, we position ourselves as an application provider who wants to benchmark an application to determine the bottleneck tiers. For these experiments, we considered the following metrics:

- The CPU and memory loads of VM servers;
- The response time for requests and their throughput;
- The number of vUsers.

We focus on the maximal throughput provided by the SUT which maintain a percentage of requests under a given response time threshold. The SUT is considered to be saturated when the response time for more than 10% of requests exceeds this threshold (set to 5 seconds). This is in line with the conclusion of [13],where a response time longer than 5 seconds was described as likely to make 10% of potential customers navigate away in a e-commerce application. Based on these parameters, we defined the notions of *goodThroughput* (respectively *badThroughput*), which represents the throughput of requests below the threshold (respectively above the threshold). The throughput metric determines the capacity of the RUBiS application while ensuring an SLO response time. In addition to throughput, we considered the number of vUsers causing SUT saturation. The response time metric in these experiments was computed with a +-30ms margin error; for throughput, the margin of error was +-10req/s. The last metric we assessed was the cost of the experiment. Our solution, based on dynamic injector provisioning, was compared to a static injectors provisioning solution.

Configuration. The servers were configured with default values, with one exception. The JVM of the Tomcat server was configured to avoid invocation of the garbage collector during experiments. The RUBiS servlets handling injected requests manage a JDBC connexion pool. The size of this pool is equal to the sum of *max_connections* configured for the MySQL servers.

The Deployer system is deployed on a VM on the Chicon cluster, while other BaaSP components (SutSaturationController, InjectorsSaturationController, InjectionController, and CLIF) are all deployed on a single Small VM. Each CLIF injector is equipped with a CPU probe and deploys automatically (when requested) on a separate Small VM. The InjectorsSaturationController is configured to detect injector saturation when CPU load reaches 100% (using the mean from 50 statistical values). The InjectionController uses a ramp up and stabilization time (about 35 seconds) when adding vUsers.

5.2 Detecting Bottleneck Tiers.

For this experiment, all RUBiS servers were deployed on Small VMs.

MySQL. The first bottleneck tier was the one limiting application performance (maximum throughput in our case). To identify this tier, we tested a RUBiS configuration comprising a Tomcat server linked to a MySQL server. The results of this experiment, performed using BaaSP, using up to 3 injector VMs and with about 250 vUsers are shown in Fig. 7. It is clear that the MySQL VM CPU reaches 100% at 380s (Fig. 7(a)), while the Tomcat VM CPU load is negligible (close to 1%). In terms of memory load, neither VM becomes saturated (Fig. 7(b)). The maximum throughput for the application (about 180 req/s) is shown to be achieved when the CPU load of MySQL VM reaches 100%. In fact, the throughput increases until 380s, and remains constant for the remainder of the experiment, whereas the number of vUsers continues to increase (Fig. 7(c)). For the response time (Fig. 7(d)), there is no badThroughput until the MySQL VM CPU reaches 100% (time 380s, curve "Good SLA"). After this time, some requests take more than 10s to execute (curve "Bad SLA"). This causes the SUTSaturationController to terminate the experiment. **In conclusion, the bottleneck tier is MySQL and its bottleneck resource is the CPU**.

To check that the BaaSP effectively detects the bottleneck tier of the SUT, we performed the same experiment without BaaSP. To do that, we use a former version of the CLIF tool which allows the tester to design the workload he wants to submit (a function of the number of vUsers over the time). Based on the results of the first experiment with BaaSP, we design a workload which follows the shape of the one generated by the BaaSPs InjectionController component. Notice that in a normal situation, the tester should test several workload to determine the appropriate shape. In order to show that the saturation point determined by BaaSP is correct (about 180 req/s), we run the test until the SUT trashes and observe this point is met or exceed. To prevent a lack of injector VMs, we statically provision up to 50 VMs. Fig. 8 shows the last results of this experiment after several attempts. Fig. 8 (a) shows that the application trashes with more than 25000 vUsers. The maximum throughput (the saturation point) is the same as in the previous experiment (180 req/s) with BaaSP. About 50% of requests took more than 20000 ms to treat (Fig 8 (b)) with more than about 4500 vUsers.

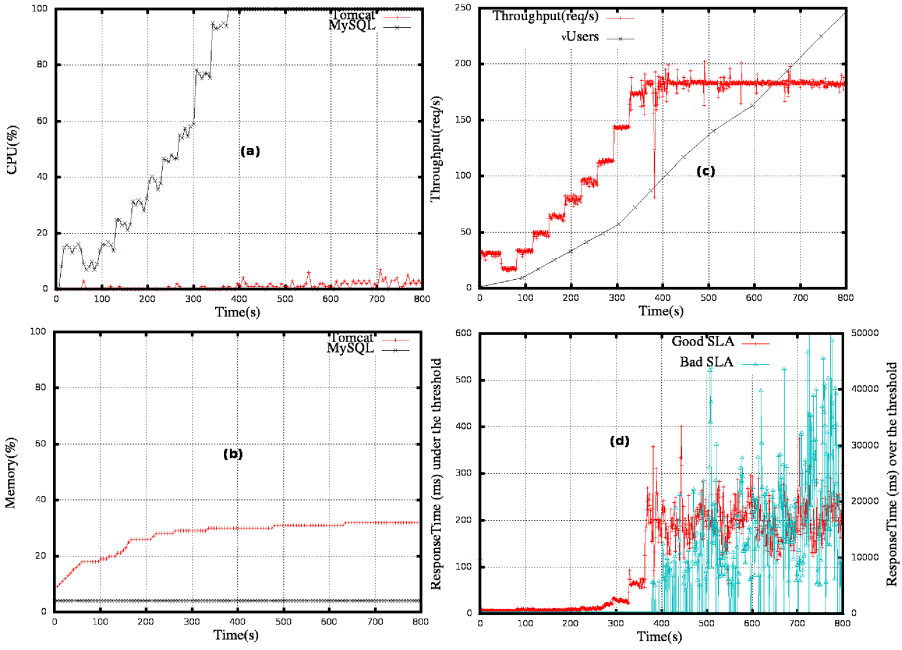

Fig. 7. Bottleneck tier detection: the MySQL server is CPU-bound. (a) Server CPU load, (b) Server memory load, (c) Application throughput, and (d) application response time (below and above the threshold)

Tomcat. MySQL is the first bottleneck tier; in this step, we determined the saturation point for the Tomcat server (i.e., how many replicated MySQL servers are needed to make Tomcat into the bottleneck tier). To do this, the experiment was repeated varying the number of MySQL servers. Experiments were stopped when the SUT's capacity (maximum throughput) in the current experiment (running n MySQL servers) was the same as in the previous experiment (running n-1 MySQL servers); n-1 MySQL servers are therefore required to saturate the Tomcat tier. This was performed for one (Fig. 9(a)) and two (Fig. 9(b)) Tomcat instances.

With one Tomcat instance (Fig. 9(a)) 18 instances of MySQL fully saturate the Tomcat tier. Up to 14 injector VMs were dynamically provisioned as necessary to complete this experiment. Plotting the CPU and memory loads for different servers in these experiments reveals Tomcat as the first bottleneck tier, with a CPU load of 100% (Fig. 10).

With two Tomcat instances (Fig. 9(b)) the Tomcat tier became saturated with 30 MySQL instances (with 25 injector VMs required to perform the test). Note that even when the number of Tomcat instances is doubled, the number of MySQL instances needed to saturate the Tomcat tier does not increase proportionally. Indeed, the application's performance is not doubled either. This is also the case when MySQL instances are doubled.

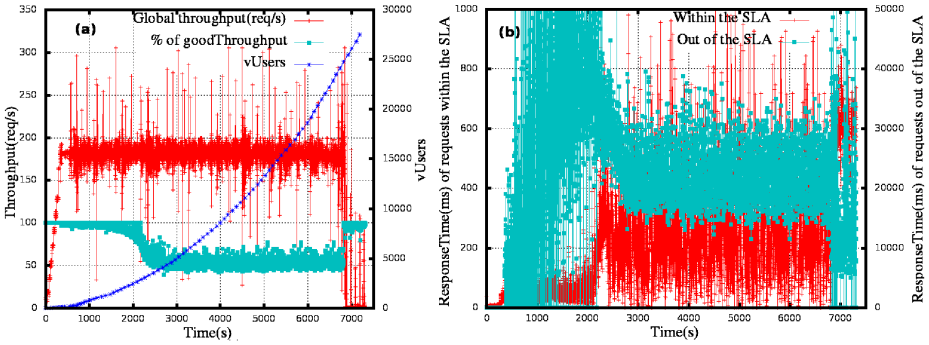

Fig. 8. Experiments without BaaSP: the application trashes with over 25000 vUsers, we observe the same saturation point as with BaaSP

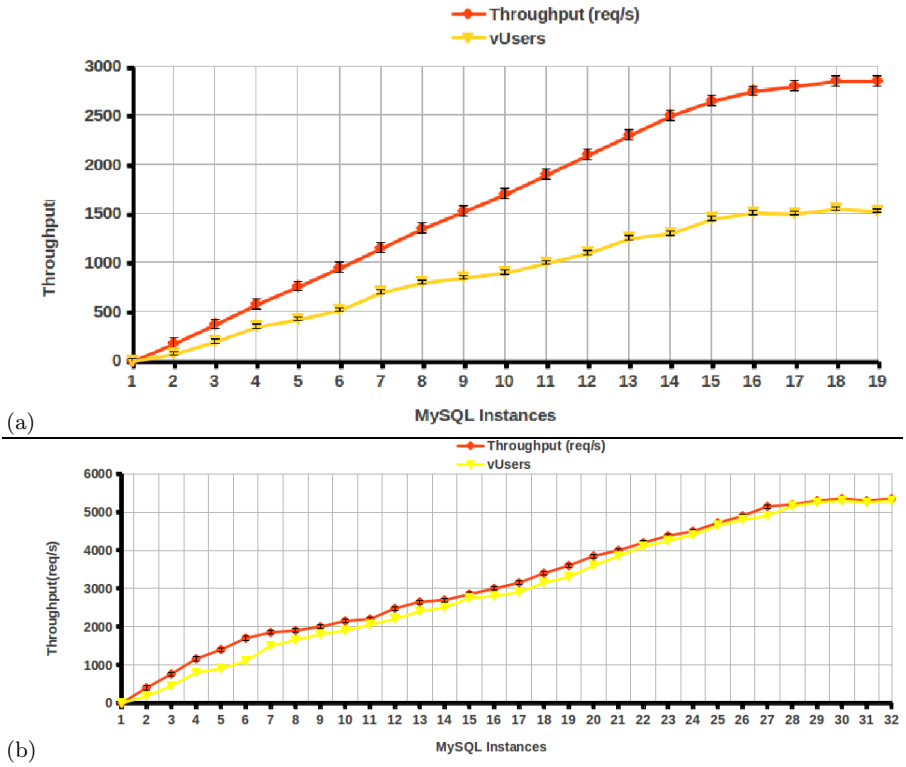

Fig. 9. How many instances of MySQL makes Tomcat the bottleneck tier with one (a) and two (b) Tomcat instances?

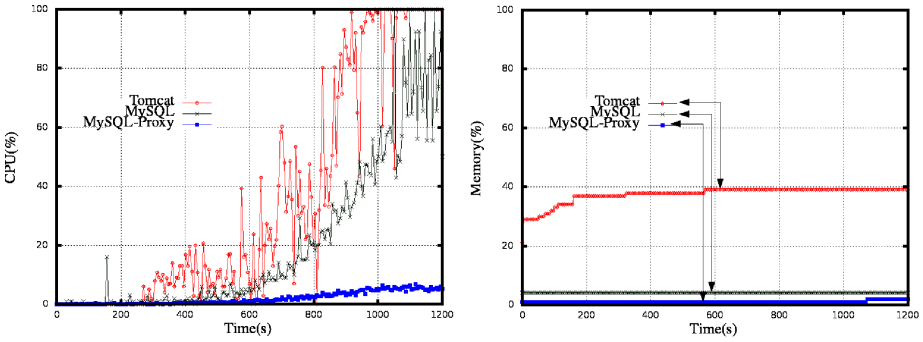

Fig. 10. CPU and Memory loads for Tomcat, MySQL-Proxy and a MySQL server when assessing 19 MySQL instances

6 Related Work

Very few studies have been published on adaptive benchmarking tools. However, we have discovered some work loosely based on this topic. Unibench [15] is an automated benchmarking tool which can remotely deploy both the SUT and the benchmarking components in a cluster similar to that presented here. Almeida and Vierra present the research challenges surrounding the implementation of benchmarking tools for self-adaptative systems [16]. Except for the definition of metrics and some principles defining the workload, the self-adaptation of the benchmarking tool itself is not covered. CloudGauge [17] is an open source framework similar to ours. It uses the cloud environment as the benchmarking context. Unlike BaaSP, which assesses an SUT running in the cloud, CloudGauge's SUT is the cloud and its capacity to consolidate VMs. CloudGauge dynamically injects workloads into the cloud VM and adds/removes/migrates VMs according to fluctuations in the workload. Like our InjectionController component, CloudGauge automatically adjusts the workload during benchmarking. Since the SUT is the cloud, injectors are deployed inside VMs. Thus, there is no separation between injector nodes and SUT nodes. This means that, unlike with BaaSP, there is no need to dynamically create injector nodes. Other tools such as VSCBenchmark [18] and VMark [19] are comparable to CloudGauge. They allow a dynamic workload to be defined to consolidate VM benchmarking in a cloud environment.

To our knowledge, BaaSP is the only open source benchmarking framework to offer automated benchmarking in a cloud-based platform. Expertus [20] automates the benchmarking process, but does not implement dynamic injector provisioning, or automated load generation features. One of the advantages of Expertus is that it generates code to automate the execution of a set of tests. BlazeMeter [21] is an evolution of JMeter [22], it allows dynamic injector allocation and de-allocation in the cloud to reduce the cost of tests. As this tool is proprietary, no technical or scientific description is available, making comparisons difficult. NeoLoad [23] is similar to BlazeMeter, it allows deployment of

injectors in a cloud environment to benchmark an application. New injectors can be integrated throughout the benchmarking process. However, this integration must be initiated by the administrator through planning. Unlike BaaSP, NeoLoad does not include an automated injector saturation detection component.

7 Conclusion

This paper explores Cloud Computing features to facilitate application benchmarking and to test scalability. Load testing solutions can be provided on-demand in the cloud and can benefit from self-scalability.

We describe a Benchmark-as-a-Service platform that provides a number of benefits in terms of self-traffic generation, reduced cost and resource savings. Traffic is generated automatically without tester intervention, as with non-cloud-based BaaSP. The traffic-generating algorithm uses statistical formulas based on the computed response time and throughput of the SUT. The self-scalability of the platform facilitates benchmarking and reduces reduces the cost for lengthy campaigns. In fact, it requires no static provisioning, which can become prohibitive in terms of human and hardware resources. Experiments on the RUBiS benchmark show how BaaSP determines the bottleneck tiers of a JEE application. The same experiments done by hand without BaaSP show the same results, but with more hardware resources used after several attempts.

We next plan to add auto-scalability to the RUBiS benchmark and to enhance our Benchmark-as-a-service platform to report the resource provisioning of the self-scalable RUBiS itself.

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. Amza, C., Cecchet, E., Chanda, A., Cox, A.L., Elnikety, S., Gil, R., Marguerite, J., Rajamani, K., Zwaenepoel, W.: Specification and implementation of dynamic web site benchmarks. In: IEEE Annual Workshop on Workload Characterization, Austin, TX, USA, pp. 3–13 (2002)
2. Dillenseger, B.: CLIF, a framework based on fractal for flexible, distributed load testing. In: Annals of Telecommunications, vol. 64(1-2), pp. 101–120. Springer, Paris (2009)
3. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: An Open Component Model and Its Support in Java. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 7–22. Springer, Heidelberg (2004)
4. Harbaoui, A., Salmi, N., Dillenseger, B., Vincent, J.: Introducing Queuing Network-Based Performance Awareness in Autonomic Systems. In: Proceedings of the International Conference on Autonomic and Autonomous Systems, Cancun, Mexico, pp. 7–12 (2010)

5. Kleinrock, L.: Queueing Systems. Wiley-Interscience, New York (1975) ISBN 0471491101
6. Stewart, W.: Introduction to the Numerical Solution of Markov Chains. Princeton University Press, Princeton (1994) ISBN 0691036993
7. Jain, R.K.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and modelling. John Wiley and Sons, Inc., Canada (1991) ISBN 0471503363
8. Oracle, Java Message Service, (October 2012), http://docs.oracle.com/cd/E19957-01/816-5904-10/816-5904-10.pdf
9. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer 36(1), 41–50 (2003)
10. Grid'5000: a scientific instrument designed to support experiment-driven research (October 2012), https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home
11. Openstack web site (October 2012), http://openstack.org/
12. Amazon Web Services, Amazon EC2 auto-scaling functions (October 2012), http://aws.amazon.com/fr/autoscaling/
13. Simic, B.: The performance of web applications: Customers are won or lost in one second. A. R. Library (2008)
14. Wang, Q., Malkowski, S., Jayasinghe, D., Xiong, P., Pu, C., Kanemasa, Y., Kawaba, M., Harada, L.: The impact of soft resource allocation on n-tier application scalability. In: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, Washington, DC, USA, pp. 1034–1045 (2011)
15. Rolls, D., Joslin, C., Scholz, S.-B.: Unibench: a tool for automated and collaborative benchmarking. In: Proceedings of the IEEE International Conference on Program Comprehension, Braga, Portugal, pp. 50–51 (2010)
16. Almeida, R., Vieira, M.: Benchmarking the resilience of self-adaptive software systems: perspectives and challenges. In: Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Waikiki, Honolulu, HI, USA, pp. 190–195 (2011)
17. El-Refaey, M.A., Rizkaa, M.A.: CloudGauge: a dynamic cloud and virtualization benchmarking suite. In: Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, Larissa, Greece, pp. 66–75 (2010)
18. Jin, H., Cao, W., Yuan, P., Xie, X.: VSCBenchmark: benchmark for dynamic server performance of virtualization technology. In: Proceedings of the International Forum on Next-Generation Multicore/Manycore Technologies, Cairo, Egypt, pp. 1–8 (2008)
19. Makhija, V., Herndon, B., Smith, P., Roderick, L., Zamost, E., Anderson, J.: VMmark: a scalable benchmark for virtualized systems, Technical Report VMware-TR-2006-002, Palo Alto, CA, USA (September 2006)
20. Jayasinghe, D., Swint, G.S., Malkowski, S., Li, J., Park, J., Pu, C.: Expertus: A Generator Approach to Automate Performance Testing in IaaS Clouds. In: Proceedings of the IEEE International Conference on Cloud Computing, Honolulu, HI, USA, pp. 115–122 (June 2012)
21. BlazeMeter, Dependability benchmarking project (October 2012), http://blazemeter.com/
22. The Apache Software Foundation, Apache JMeter (October 2012), http://jmeter.apache.org/
23. Neotys, NeoLoad: load test all web and mobile applications (October 2012), http://www.neotys.fr/