A Benchmarking Based SLA Feasibility Study Method for Platform as a Service

Ge Li, Frédéric Pourraz, Patrice Moreaux

Université Savoie Mont Blanc, Annecy le vieux, France
Email: {ge.li, frederic.pourraz, patrice.moreaux}@univ-savoie.fr

***Abstract*—Service Level Agreements (SLA) are contracts established between Software as a Service (SaaS) providers and Platform as a Service (PaaS) providers related to various properties of the services running on the platform. We propose a generic method to evaluate to what extent SaaS provider proposed Quality of Service (QoS) targets, such as response time or maximal throughput, can be guaranteed with constraints on workload, resources and cost. These "what-if" evaluations are based on small scale benchmarking and ability of the application to be scaled. The approach allows us to estimate with minimal costs, to what extend a given SLA can be accepted by both client and provider of a given PaaS.**

Keywords–SLA; Capacity planning; SLA negotiation; Benchmark; PaaS.

I. INTRODUCTION

Cloud services [1] can be classified as Infrastructure as a Service (IaaS), PaaS and SaaS. SaaS providers rely on PaaS features to benefit from the "pay as you go" paradigm corresponding to pay only "adapted" resources usage at runtime while satisfying a set of constraints related to QoS, workload and cost. Elasticity [2] is the property of the platform to meet all these requirements at a given variation rate. SLAs are used to describe the responsibilities of contracting parties (SaaS provider and PaaS provider). SLA management involves two stages [3]: establishing an SLA before runtime, and fulfilling the SLA at runtime. Establishing SLA is usually done through an automated negotiation process. It involves benchmarking and/or modelling the system (application and runtime support) to define the levels of QoS accepted by the SaaS "client" and offered by the PaaS provider and the constraints both parties will fulfil.

Web Service Level Agreement (WSLA), WS-Agreement and other SLA models proposed in SLA@SOI are significant proposals about SLA [4]. However, more work has to be done to take into account SLA in the context of cloud computing [3], especially at runtime. PSLA [5] is a PaaS level SLA description language based on WS-Agreement, which is an extendible SLA skeleton. PSLA takes the particular needs in PaaS level SLA into consideration and is well structured, commonly usable and machine readable for PaaS level SLA management. Modelling arrival rate and resource demands when satisfying QoS targets are foundations of capacity evaluation. Modelling methods can be classified as simulation modelling and analytic modelling.

As it is the case for most of current applications deployed in cloud infrastructures, we assume that the architecture of the application (App) is given as a graph of components (whatever the component model used) and that the designer of App has mapped this architecture on a graph of stages. Vertices of the graph are the components and edges are the relationships service required - service provided between components. A stage s is then instantiated as identical VMs at runtime. Horizontal scaling (HS), or scaling out, of s, means that s can be instantiated by several VMs (with a front-end Load Balancer (LB)). Vertical scaling (VS), or scaling up, of s means that VMs with different resources (for instance 1 to 3 CPUs, 1 to 4 GB of RAM), termed flavors, can instantiate s. Hence, a configuration of App is the deployed set of VMs of its graph of stages according to the HS and/or VS capability of the stages. Since we want to design a non intrusive method with respect to the application, our model will be based on results of benchmarks run in the target PaaS context, using measures also available at runtime.

Our method is as follows. First, we check to what extent the proposed SLA can be meet. To do so, we benchmark various configurations of the application. If the application and the underlying IaaSs on which the configuration is deployed can achieve the SLA, then we control the running application based on a set of configurations derived from the first step and on monitoring both the submitted workload (user's requests) and the behaviour of the running application. In this paper, we address the first step of the method and we propose an application independent, cost effective, benchmarking based, SLA feasibility study method from the PaaS provider's perspective. Since the number of feasible configurations may be "large" (more than 100) and time for benchmarking can be from minutes to hours, we propose to reduce the number of benchmarks by selecting the set of configurations benchmarked.

The paper is organized as follows. Section II briefly reminds the reader benchmarking methodology. Then Section III presents the context of our approach. In Section IV, we describe in detail our benchmark based SLA feasibility study method. Finally, we conclude this paper in Section V.

II. RELATED WORK

Benchmarking methods [6] are a well known set of methods to analyse the behaviour of software and hardware. The increasing number of software systems running in cloud infrastructures has raised a new interest in benchmarking because of the specificities of the running environments of these systems and of the variety of the applications. In [7], Iosup et al., summarise recent work on benchmarking cloud applications and propose to adapt benchmarking methods to cope with properties of these applications, especially for what concerns elasticity and variability. Due to the fact that controlling the runtime cloud application can lead to modify its allocated resources and since the application uses resources which are usually shared with other applications under control of hypervisors, benchmarking such applications must address the behaviour of the application faced up to these endogenous and exogenous variations [8].

III. SLA FEASIBILITY STUDY CONTEXT

SLA feasibility study is based on the evaluations of elasticity constraints described in PSLA. QoS target quantifies

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-388-9

TABLE I. SLA FEASABILITY STUDY

(a) FLAVOR DESCRIPTION

FLAVOR	L_cpu	M_cpu	L_gnrl	M_gnrl	L_ram	M_ram	S_ram
vCPU	8	5	6	4	5	3	1
RAM(G)	2.67	1.67	3	2	10	6	2
DISK(G)	13.35	8.35	15	10	25	15	5
COST(euro/s)	5.34	3.34	3	2	5	3	1

(b) RESOURCE CONSTRAINTS: MAXIMUM NoI

MAX_NoI	L_cpu	M_cpu	L_gnrl	M_gnrl	L_ram	M_ram	S_ram
Apache			2	3			
Jonas	4	6					
Mysql	2	3	2	3	2	3	4

(c) RESOURCE CONSTRAINTS: MINIMUM NoI

MIN_NoI	L_cpu	M_cpu	L_gnrl	M_gnrl	L_ram	M_ram	S_ram
Apache			1	1			
Jonas	2	2					
Mysql	1	1	1	1	1	1	1

(d) MFC(TIER,1,FLAVOR).

MFC(*,1,*)	L_cpu	M_cpu	L_gnrl	M_gnrl	L_ram	M_ram	S_ram
Apache			300	200			
Jonas	150	94					
Mysql	134	84	150	100	400	300	100

(e) MFC(TIER,MAX_NoI,FLAVOR).

MFC(*,MAX_NoI,*)	L_cpu	M_cpu	L_gnrl	M_gnrl	L_ram	M_ram	S_ram	MAX_MFC(TIER)
Apache			600	600	400			600
Jonas	600	564						600
Mysql	268	252	300	300	800	900	400	900

(f) MINIMUM COST TO SERVE MAR

COST	L_cpu	M_cpu	L_gnrl	M_gnrl	L_ram	M_ram	S_ram
Apache			3	4			
Jonas	10.68	13.36					
Mysql	16.02	13.36	6	6	5	3	3

the acceptable QoS metric "Value Range", e.g., Response time $\leq$ 13s. Workload constraints include "Data Size", "Data Composition"; e.g., it is composed of only "atomic request A" with a Maximum Arrival Rate (MAR) of 300 requests/s. Resource constraints indicate the kinds of allowed scaling actions and its limits; e.g., we indicate the minimum (in Table I(c)) and maximum (in Table I(b)) Number of Instances (NoI) for flavors (a flavor is a set of resources allocated to a virtual machine) (in Table I(a)) and tiers, which can be used for scaling actions. Cost constraints tell how much money can be spent per time unit, e.g., cost $<$ 10 euro/s.

Scaling actions for virtualized application are at the virtual machine level. To benchmark the application, we use "CLIF is a Load Injection Framework" (CLIF) and SelfBench services, which are provided as services under OpenCloudware project. CLIF [9] is designed for generating predefined traffic on a system to measure QoS target and observe the computing resources usage at the same time. CLIF can be used for deployment, remote control, monitored measurements collection of its distributed load injectors and probes. SelfBench [10] provides a virtualized and self-scalable load injection system to automatically detect the supportable upper bounder arrival rate of the system. SelfBench is based on CLIF. System MAR and corresponding resource utilization can be achieved by applying SelfBench.

Let us list terms and abbreviations used to describe our feasibility study method (see algorithms). To simplify explanation, we consider a three tier application.

Resource Allocation Scheme (RAS) is in the form of (FLAVOR1(NoI1), FLAVOR2(NoI2), FLAVOR3(NoI3)), where FLAVOR1(NoI1) denotes using NoI1 instances of FLAVOR1 on first tier.

CLIF(ARRIVAL_RATE, RAS) with result CAP/NO_CAP denotes do CLIF benchmark on RAS with ARRIVAL_RATE and get result that RAS can serve ARRIVAL_RATE(CAP) or not(NO_CAP).

SelfBench(RAS) with result MAX_ARRIVAL_RATE denotes do SelfBench on RAS and get the maximum capable arrival rate MAX_ARRIVAL_RATE of RAS.

Average resource utilization (e.g., Avg_RAM_UTIL(ARRIVAL_RATE) and Avg_CPU_UTIL(ARRIVAL_RATE)) can be achieved by both benchmarks.

BCF(TIER) denotes Best Choice Flavor for a given tier TIER . When adopting BCF(TIER), CPU utilization and RAM utilization are relatively balanced compared to non BCF(TIER).

MAXTH is the maximum arrival rate, which can be served by respecting the QoS target in SLA with a RAS of the biggest BCF(TIER) with minimum NoI.

ORAS(ARRIVAL_RATE) is an Optimized RAS, which is able to serve the arrival rate of ARRIVAL_RATE by respecting the QoS target with the minimum NoI and as small as possible size of BCF(TIER) according to SLA.

MFC(TIER,N,FLAVOR) = MAX_ARR_RATE denotes the maximum arrival rate can be served for a RAS with N instances of FLAVOR on TIER is *at most* MAX_ARR_RATE. So the integral capable arrival rate should be the minimum achievable MFC(TIER,N,FLAVOR) of all tiers. It is reasonable to assume that (1) is fulfilled.

$$MFC(TIER, N, FLAVOR) = N * MFC(TIER, 1, FLAVOR) \quad (1)$$

All possible MFC(TIER,1,FLAVOR) will be benchmarked or deduced for estimating the integral capable arrival rate of a RAS.

RAS(MFC(TIER,N,FLAVOR)) means a RAS which can serve maximum arrival rate of MFC(TIER,N,FLAVOR) with N instances of FLAVOR on TIER.

RAS(BCF(TIER),MIN_NoI) denotes a RAS with *the biggest* BCF and minimal NoI.

CAP_SLA_ReCst denotes the CAPable arrival rate according to SLA Resources Constraints.

COST(RAS) denotes the cost per charge unit for RAS and COST_AR_SLA is the minimal cost per charge unit for serving the arrival rate required in SLA.

IV. METHOD DESCRIPTION

Our method estimates the capable workload. Over estimation may lead to under provisioning, which means high probability of SLA violation. Under estimation leads to over provisioning, which means unnecessary cost. From the PaaS provider's perspective, low level under estimation is inevitable for signing a contract.

In this part, our SLA feasibility study method will be introduced according to the steps. For each step, we will formally describe our method and corresponding examples.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-388-9

```
Require: FLAVOR, MIN_NoI
Ensure: BCF(TIER) for each tier.
function SELFBENCH_BCF( ORAS(FLAVOR), ORAS(MIN_NoI))
    SelfBench(S_gnrl(MIN_NoI),S_gnrl(MIN_NoI),S_gnrl(MIN_NoI))
    for each TIER do
        CPU_RAM_RATIO = Avg_CPU_UTIL/Avg_RAM_UTIL
        if CPU_RAM_RATIO ≥ 1.5 then
            BCF(TIER)=X_cpu
        else if CPU_RAM_RATIO ≤ 0.5 then
            BCF(TIER)=X_ram
        else
            BCF(TIER)=X_gnrl
        end if
    end for
    return BCF(TIER) for each tier
end function
```

Figure 1. Algorithm Find BCF(TIER)

```
Require: BCF(TIER), MIN_NoI
Ensure: MAXTH
function SELFBENCH_MAXTH(FLAVOR, MIN_NoI)
    SelfBench(RAS(BCF(TIER),MIN_NoI))
    return MAXTH = MAX_ARRIVAL_RATE
end function
```

Figure 2. Algorithm Find MAXTH

A. Step 1: Find BCF(TIER)

As described in the algorithm of Figure 1, we know BCF(TIER) if several kinds of flavors are allowed in SLA. Using non BCF(TIER) can be interesting when the maximum capability of BCF(TIER) is not sufficient, e.g., BCF(Apache) is X_gnrl, BCF(Jonas) is X_cpu and BCF(Mysql) is X_ram. So, ORAS(ARRIVAL_RATE) should have a RAS of (X_gnrl(1),X_cpu(2),X_ram(1)).

B. Step 2: Find MAXTH

As described in the algorithm of Figure 2, we do SelfBench on RAS (L_gnrl(1),L_cpu(2),L_ram(1)) with the biggest BCF(TIER) on each tier. MAXTH is 300 requests/s.

C. Step 3: ORAS(MAXTH) exploration

"new RAS" can be not existing because of SLA constraints. According to the algorithm of Figure 3, we check the resource utilization of doing SelfBench on (L_gnrl(1),L_cpu(2),L_ram(1)), and only Mysql tier is not yet saturated when arrival rate reaches MAXTH. So, the size of the Mysql tier flavor is decreased to get a "new RAS" (L_gnrl(1), L_cpu(2), M_ram(1)). CLIF benchmark result on the "new RAS" shows that MAXTH can be served, so we do CLIF benchmark on (L_gnrl(1),L_cpu(2),S_ram(1)), which indicate that MAXTH can not be served. So ORAS(MAXTH)=(L_gnrl,2L_cpu,M_ram).

D. Step 4: Exploring MFC of flavors used in ORAS(MAXTH)

According to the algorithm of Figure 4, ORAS(MAXTH) can be approximately seen as a RAS fitly allocated the resources to serve arrival rate of 300 requests/s. A RAS has one instance of L_gnrl on Apache tier, which is the same as ORAS(MAXTH)=300, can serve maximum arrival rate *at most* 300 requests/s. So MFC(Apache,1,L_gnrl)=300, MFC(Mysql,1,M_ram)=300 and MFC(Jonas,1,L_cpu)=MFC(Jonas,2,L_cpu)/2=MAXTH/2=150 according to equation 1.

```
Require: MIN_NoI, MAXTH, BCF(TIER), FLAVOR
Ensure: ORAS(MAXTH)
function FIND_ORAS_MAXTH( MIN_NoI, MAXTH, BCF(TIER), FLAVOR)
    ORAS(MAXTH) = NULL
    repeat
        TIER = 0
        repeat
            TIER ++
            Check Avg_RAM_UTIL, Avg_CPU_UTIL for TIER.
        until Both Avg_RAM_UTIL and Avg_CPU_UTIL are unsaturated
        if unsaturated TIER existing then
            "new RAS" is a RAS using smaller flavor on TIER.
            if "new RAS" existing then
                CLIF(MAXTH, "new RAS")
                if NO_CAP then
                    ORAS(MAXTH)="previous RAS"
                end if
            else
                ORAS(MAXTH)= "previous RAS"
            end if
        else
            ORAS(MAXTH)= "current RAS"
        end if
    until ORAS(MAXTH)!=NULL
    return ORAS(MAXTH)
end function
```

Figure 3. ORAS(MAXTH) exploration

```
Require: ORAS(MAXTH), MAXTH
Ensure: MFC(TIER,1,FLAVOR used in ORAS(MAXTH)
function SPECU_MFC_ORAS(ORAS(MAXTH), MAXTH)
    for each FLAVOR used by TIER in ORAS(MAXTH) do
        N= NoI for FLAVOR in TIER in ORAS(MAXTH)
        MFC(TIER,1,FLAVOR)=MAXTH/N
    end for
    return MFC(TIER,1,FLAVOR used in ORAS(MAXTH)
end function
```

Figure 4. Exploring MFC of flavors used in ORAS(MAXTH)

E. Step 5: Speculating MFC of flavors smaller than the one used in ORAS(MAXTH)

According to the algorithm of Figure 5, linear estimation as in Figure 6 can be used for deducing MFC of smaller flavor based on a bigger flavor. We consider both CPU and RAM. Comparing with the CPU and RAM of the flavor used in corresponding tier of ORAS(MAXTH), we get two ratios Ratio_CPU and Ratio_RAM, and we take the minimum ratio, which is lower or equals to 1. Then, MFC(tier,1,FLAVOR)= MIN_Ratio*MFC(tier,1,flavor in ORAS(MAXTH)). e.g., M_cpu allocated 5/3 of CPU and M_ram 1.67/6 of RAM. So, MIN_Ratio(M_cpu,

```
Require: FLAVOR, ORAS(MAXTH), MFC(TIER,1,flavor used in ORAS(MAXTH))
Ensure: MFC(TIER,1,smaller FLAVOR)
function SPECU_MFC_LESS( FLAVOR, ORAS(MAXTH), MFC(TIER,1,flavor used in ORAS(MAXTH)))
    for each FLAVOR in each TIER do
        Ratio_CPU(FLAVOR, flavor used in ORAS(MAXTH))
        Ratio_RAM(FLAVOR, flavor used in ORAS(MAXTH))
        MIN_Ratio = min(Ratio_CPU, Ratio_RAM)
        if MIN_Ratio ≤ 1 then
            MFC(TIER,1,FLAVOR_smlr_oras)= MIN_Ratio*MFC(TIER,1,flavor used in ORAS(MAXTH))
        end if
    end for
    return MFC(TIER,1,smaller FLAVOR)
end function
```

Figure 5. Speculating MFC of flavors smaller than the one used in ORAS(MAXTH)

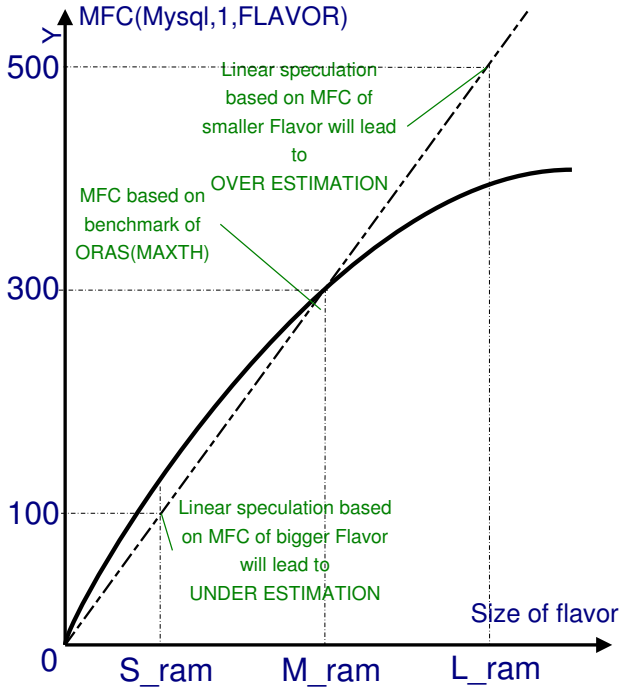

Figure 6. Under estimation and over estimation

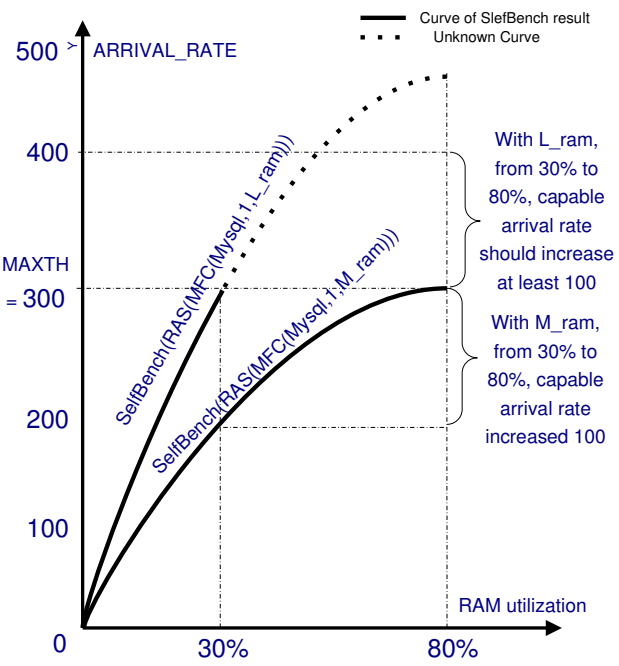

Figure 8. Linear estimation of bigger flavor MFC based on smaller flavor

Require: FLAVOR, ORAS(MAXTH), MIN_Ratio MFC(TIER,1,flavor used in ORAS(MAXTH))
Ensure: MFC(TIER,1,bigger FLAVOR).
1: **function** SPECU_MFC_GREATER(FLAVOR, ORAS(MAXTH), MIN_Ratio, MFC(TIER,1,flavor used in ORAS(MAXTH)))
2: SelfBench(ORAS(MAXTH))
3: **for** each FLAVOR MIN_Ratio $>$ 1 in each TIER **do**
4: Re_util= Avg_RAM_UTIL(MAXTH) for FLAVOR by checking benchmark report in Step2 or Step3
5: Arr_rate=ARRIVAL_RATE(Re_util) by checking SelfBench report in Step6.
6: augmentation = MAXTH - Arr_rate
7: MFC(TIER,1,bigger FLAVOR)=MAXTH + augmentation
8: **end for**
9: **return** MFC(TIER,1,bigger FLAVOR)
10: **end function**

Figure 7. Speculating MFC of flavors bigger than the one used in ORAS(MAXTH)

M_ram)=1.67/6. Since $1.67/6 < 1$, MFC(Mysql,1, M_cpu) = MIN_Ratio(M_cpu, M_ram) $*$MFC(Mysql,1,M_ram)$= 1.67/6 * 300 = 84$. Other MFC(TIER,1,FLAVOR) which have a MIN_Ratio ≤ 1 can be speculated similarly. If MIN_Ratio > 1, MFC(TIER,1,FLAVOR) can't be linearly deduced in the same way. Otherwise, overestimation will be introduced because of the concavity and convexity of the curve in Figure 6 [11].

F. Step 6: Speculating MFC of flavors bigger than the one used in ORAS(MAXTH)

If RAS1 and RAS2 use the same FLAVOR and NoI for TIER, the observable *overlapped* ARRIVAL_RATE of SelfBench(RAS1) and SelfBench(RAS2) results have approximately the same Avg_RAM_UTIL and Avg_CPU_UTIL SelfBench(L_gnrl(1),L_cpu(2),L_ram(1)) is done in Step2. Avg_RAM_UTILs and Avg_CPU_UTILs of Mysql tier for ARRIVAL_RATE from 0 to MAXTH are achievable and can

Require: MFC(TIER,1,FLAVOR), MAX_NoI
Ensure: AAR_RC_SLA.
1: **function** FIND_AAR_RC_SLA(MFC(TIER,1,FLAVOR, MAX_NoI)
2: **for** each TIER **do**
3: **for** each FLAVOR **do**
4: MFC(TIER,MAX_NoI,FLAVOR)= MFC(TIER,1,FLAVOR)* MAX_NoI
5: **end for**
6: MAX_MFC(TIER)= MAX{MFC(TIER,MAX_NoI,FLAVOR)}
7: **end for**
8: CAP_SLA_ReCst= MIN{MAX_MFC(TIER)}
9: **return** AAR_RC_SLA
10: **end function**

Figure 9. Resources constraints evaluation

be seen as the same for RAS(MFC(Mysql,1,L_ram)).

Then, we do SelfBench(ORAS(MAXTH)) and get the curve of Figure 8 for RAS(MFC(Mysql,1,M_ram)). The goal is to estimate ARRIVAL_RATE for RAS(MFC(Mysql,1,L_ram)) when Avg_RAM_UTIL is saturated (e.g., reaches 80%). This ARRIVAL_RATE is MFC(Mysql,1,L_ram). Avg_RAM_UTIL increases from 30% to 80%, ARRIVAL_RATE of RAS(MFC(Mysql,1,L_ram)) augmentation should be *more than* RAS(MFC(Mysql,1,M_ram))'s. Under estimation is made for MFC(Mysql,1,L_ram) 400 by taking the smaller augmentation of SelfBench(RAS(MFC(Mysql,1,M_ram))).

G. Step 7: Resources constraints evaluation

Until now, we have all the MFC(TIER,1,FLAVOR) I(d). The maximum capable arrival rate of all possible RAS, should be the minimum value, which is achieved by comparing the maximum MFC(TIER,MAX_NoI,FLAVOR) values in each tier, among all tiers. e.g., according to MAX_NoI Table I(b) from SLA and MFC(TIER,1,FLAVOR) Table I(d) achieved during benchmark, we know the MFC(tier, MAX_NoI, flavor) Table I(e). The maximum arrival rate can be served according

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-388-9

```
Require: MFC(TIER,1,FLAVOR), FLAVOR, MAR
Ensure: COST_AR_SLA.
function FIND_COST_AR_SLA(MFC(TIER,1,FLAVOR), FLAVOR, MAR)
    for each TIER do
        for each FLAVOR do
            MAX_NoI_MAR=⌈MAR/MFC(TIER, 1, FLAVOR)⌉
            COST(TIER,FLAVOR)= MAX_NoI_MAR * FLAVOR.COST
        end for
        MIN_COST(TIER)= min{COST(TIER,FLAVOR)}
    end for
    COST_AR_SLA= ∑MIN_COST(TIER)
    return COST_AR_SLA
end function
```

FIGURE 10. COST CONSTRAINTS EVALUATION

to resource constraints in SLA is MIN{600,600,900}=600, which is higher than SLA requirement(300 requests/s). So, the SLA can be achieved while fulfilling the resource constraints.

H. Step 8: Cost constraints evaluation

According to the algorithm of Figure 10, the cost for each tier with each flavor is shown in Table I(f). The minimum cost to serve MAR of 300 requests/s is 3+10.68+3 = 16.68(euro/s), which is higher than SLA cost constraint(maximum 10 euro/s). So SLA feasibility study refuses current version of SLA for cost constraints.

V. CONCLUSION

We proposed an SLA feasibility study method based on limited amount of benchmarking with application's Auto-scaling group architecture. The "what-if" evaluations answer the question that whether QoS target can be satisfied with the workload, resource and cost constraints stated in SLA. We experiment our method in the context of the OpenCloudware project [12] and on a small experimental set-up on which we are deploying a configurable virtualized application that we can stress in various ways to check the quality of derivations made from th benchmarks. Future work will be first to verify the accuracy of our proposal. Then, we will design our runtime controller to provide elasticity to virtualized applications using informations recorded during our benchmarking campaign while satsifying SLA.

ACKNOWLEDGMENT

The research described in this paper is supported by the OpenCloudware project [12].

REFERENCES

[1] W. Voorsluys, J. Broberg, and R. Buyya, Introduction to Cloud Computing. John Wiley & Sons, Inc., 2011, pp. 1–41. [Online]. Available: http://dx.doi.org/10.1002/9780470940105.ch1

[2] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in Cloud Computing: What it is, and What it is Not," in Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2013), San Jose, CA, June 24–28, 2013, pp. 23–27.

[3] D. Kyriazis, "Cloud Computing Service Level Agreements - Exploitation of Research Results," European Commission, Directorate General Communications Networs, Content and Technology Unit E2 - Software and Services, Cloud, Brussels, Belgium, Report/Study, Jun. 2013.

[4] P. Patel, A. Ranabahu, and A. Sheth, "Service level agreement in cloud computing," in Cloud Worshop, OOPSLA 2009, S. Arora and G. T. Leavens, Eds. Osrlando, FL, USA: ACM, Oct. 25-29 2009.

[5] G. Li, F. Pourraz, and P. Moreaux, "PSLA: a PaaS Level SLA Description Language," in Intercloud 2014, Boston, United States, Mar. 2014.

[6] R. W. Hockney, The Science of Computer Benchmarking, ser. Software, environments, tools. Society for Industrial and Applied Mathematics, 1996.

[7] A. Iosup, R. Prodan, and D. Epema, "Iaas cloud benchmarking: Approaches, challenges, and experience," in 5th Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS 2012, (SC),, Salt Lake City, UT, USA, Nov. 2012, invited paper.

[8] A. Alexandrov, E. Folkerts, K. Sachs, A. Iosup, V. Markl, and C. Tosun, "Benchmarking in the cloud: What it should, can, and cannot be," Aug 2012.

[9] B. Dillenseger, "Clif, a framework based on fractal for flexible, distributed load testing," annals of telecommunications, vol. 64, no. 1-2, 2009, pp. 101–120.

[10] A. Tchana et al., "A self-scalable and auto-regulated request injection benchmarking tool for automatic saturation detection," Cloud Computing, IEEE Transactions on, vol. 2, no. 3, July 2014, pp. 279–291.

[11] N. Gunther, Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services. Springer, 2007.

[12] "OpenCloudware," [retrieved 02,2015], http://opencloudware.org/.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-388-9