Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 337 (2018) 45–65

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Performance Evaluation of Replication Policies in Microservice Based Architectures

Marco Gribaudo[1]

Politecnico di Milano,
via Ponzio 34/5, 20133 Milano (Italy)

Mauro Iacono[2]

Università degli Studi della Campania "L. Vanvitelli"
viale Lincoln 5, 81100 Caserta (Italy)

Daniele Manini[3]

Università degli Studi di Torino
corso Svizzera, 185, 10129 Torino, Italy

Abstract

Nowadays applications tend to be executed on distributed environments provisioned using on-demand infrastructures. The use of techniques such as application containers simplifies the orchestration of complex systems. In this context, microservices based architectures offer a promising solution for what concerns software development and scalability. In this paper, we propose an approach to study the automatic scalability of microservices architectures deployed in public and private clouds. A Fluid Petri Net model describes the characterise of the platform, and a real trace drives the approach to consider a realistic scenario. Our focus is on evaluating the performances, costs and energy consumptions from both the service provider and infrastructure provider point of view.

Keywords: Performance evaluation, microservices.

1 Introduction

The need for a fast, short development cycle and an agile management of Software-as-a-Service (SaaS) applications has led to the rise of the Microservice-Based Software Architecture (MBSA). This paradigm is based on the parcelization of complex

[1] marco.gribaudo@polimi.it

[2] mauro.iacono@unicampania.it

[3] manini@unito.it

https://doi.org/10.1016/j.entcs.2018.03.033

software applications into a high number of small software services, namely MicroServices (MSs). This approach has been proposed by the industry, as a result of the application of the concepts of the Service Oriented Architecture (SOA) to the design and implementation of cloud provisioned software. SOA concepts, abstracted from the SOA implementation issues, lead to a definition of an application as an interaction (basically, a workflow) of several independent software units that provide self-contained logical functionalities. Each software service is developed and managed independently from the others, but offers a well defined and known interface to other services, thus decoupling the overall design and life cycle of an application and the design and life cycle of each service. In the MBSA approach, this is, in principle, pushed towards a higher granularity of services per application. Three advantages characterize this approach: i) services are very simple, so that they can be developed and managed by very small teams with very short cycles (compatible with DevOps and, in general, agile development approaches); ii) management of a service implies a small amount of knowledge and the development team may easily compensate a high-frequency turnover of its members; iii) the resulting architecture is, in principle, highly scalable, as every service may be executed in a variable number of instances according to the instant needs generated by workload fluctuations, and fault tolerant, as a fault in a service does not cause a failure of the application, and compensation is easy by using another instance of the same service. A MS is an isolated, loosely-coupled unit of development that works on a single concern. This usually means that MSs tend to avoid interdependencies: if one MS has a hard requirement for other MSs, then the point is to understand if it makes sense to make them all part of the same unit. A monolithic application is split into several components that can run independently and may be implemented with different coding or programming languages. The resulting independent programs are executable by themselves, then these smaller components are grouped together to deliver all the functionalities of the monolithic application. The cloud infrastructure supporting the execution and the algorithms defining resources assignment to MSs should be optimized in order to keep the overall service reliable and performing.

In [6] we already presented a modeling technique that allows to give a general characterization of MBSA, in terms of performances and resource utilization. In this work, we focus on the problems related to scalability issues with respect to cloud architectures. As cloud architectures are the most likely hosting platforms for MS based applications, the interactions between scaling and cloud management and cost policies should be investigated, in order to understand the implications of design and management decisions. In normal operations on cloud systems, costs depend on the number of Virtual Machines (VM) that are used per time frame, that is generally defined as one hour: consequently, different management policies may cause different costs on the same platform with the same overall workload, depending on the usage patterns of available VMs. We propose a modeling approach that allows to evaluate the effects of different possible scaling policies, and that allows to evaluate the costs that derive from different VM usage strategies. The modeling approach is presented in two steps, to focus on the two problems and combine different scaling and usage strategies.

The rest of this paper is organised as follows: Section 2 presents the literature describing and evaluating MBSAs in cloud environments, in Section 3 the proposed approach is presented and the model exploitation is applied to different cases study, finally Section 4 draws conclusions and introduces the directions of future developments.

2 Related works

MBSA emerged as a solution to support a fast and agile development of large applications with very simple independent services with their own separate development and maintenance cycle and team. An application is then composed of a set of (not exclusively) integrated microservices, that interact via HTTP or socket based messaging, and is executed, in general, on a cloud system. A commonly agreed definition of MBSA has been presented for the first time in [1], while a more general point of view on this model and its implications has been presented in [4], that is a suitable reference for a first approach to the theme. A very comprehensive discussion on all the aspects of the architecture and the typical workload, and a comparison between a monolithic and a MBSA application in different implementations and cases are available in [20]. The problem of costs and development cycle management have been analyzed in [21] and [22], with a comparison between MBSA and monolithic and AWS Lambda based solutions. In [7] is discussed how mircoservices support scalability for both, runtime performance and development performance, via polyglot persistence, eventual consistency, loose coupling, open source frameworks, and continuous monitoring for elastic capacity management. The use of containers has been considered in [10] where scalability issues for Docker technology are evaluated. A similar study has been developed in [12], that identifies the challenges for a full development of containers based systems. In both works [8] and [5], the operating conditions are analyzed. The former describes a proposal for resilience testing, while the latter formulates a proposal for a decentralized autonomic behavior in MS infrastructures. From the applications point of view, among the works presented in literature, [15] and [3] are an interesting starting reference for readers, since giving a clear and systematic picture of the app scenario. In [17] authors present general autoscaling techniques in cloud environments. In [18] authors propose an automated approach for the selection and configuration of cloud providers for multi-cloud MS-based applications. [13] presents benchmark results to quantify the impacts of container, software defined networking, and encryption on network performance. In [14] authors present an approach to model the deployment costs, including compute and IO costs, of MS-based applications deployed to a public cloud. [19] proposes a novel architecture that enables scalable and resilient self-management of MSs applications on cloud. An interesting and current state-of-the-art review about cloud container technologies is reported in [16]. Finally, [2] is suggested as a more extensive reference list.

3 Modeling approach

The goal of our modeling process is to evaluate the performances, costs and energy consumptions related to the provisioning of a MBSA from both the service provider and infrastructure provider point of view. In particular, we divide the evaluation into two parts: first we consider the *autoscaling strategy*, then we focus on the *provisioning scheme*. The autoscaling strategies will be described using pseudo-code algorithms, while the provisioning schemes will be modeled with Fluid Stochastic Petri Nets (FSPN).
The main purpose of an autoscaling strategy is deciding when to increase or decrease the amount of resources used to support an application. In this work, we will consider a static strategy that, based on the current workload λ, decides how many instances of each MS are required to execute the application in a stable way. Each strategy is described by a set of tuples $S = \{C_i\}$ where each element $C_i = (\lambda_U, \lambda_L, n, \mathbf{m}, A)$ defines an infrastructure configuration. Given the current workload λ, the system must be in state C_i such that $\lambda_L \leq \lambda < \lambda_U$ (λ_U is the upper bound and λ_L is the lower bound). Whenever the workload exceeds λ_U, the system moves to configuration C_{i+1}. If workload becomes lower than λ_L, the system switches to configuration C_{i-1}. To avoid alternating behaviors, give two consecutive configurations C_i and C_{i+1}, we must have that: $\lambda_U^{(i)} > \lambda_L^{(i)}$, $\lambda_U^{(i+1)} > \lambda_U^{(i)}$, $\lambda_L^{(i+1)} > \lambda_L^{(i)}$ (configuration C_{i+1} must handle a larger workload than C_i), and $\lambda_U^{(i)} > \lambda_L^{(i+1)}$ (if the system returns to configuration C_{i+1} to C_i, it must not immediately return to C_{i+1})[4]. Element n represents the number of virtual machines over which the application is deployed in configuration C_i, and $\mathbf{m} = (m_1, \cdot, m_{N_{ms}}) \in \mathbb{N}^{N_{ms}}$ is a vector whose component m_j represents the number of instances on which MS j is currently replicated. Let us call $M = \sum_{j=1}^{N_{ms}}$ the total number of instances of MSs used in the considered configuration. Matrix $A = |a_{jk}| \in \mathbb{N}^{N_{ms} \times n}$ defines the allocation of microservices on the available VMs: a_{jk}, $1 \leq j \leq N_{ms}$, $1 \leq k \leq n$ defines the number of type j MSs currently running on VM k. For the set of configurations S to be valid, we need te following additional constraints: $n^{(i+1)} \geq n^{(i)}$ (the number of provisioned VMs can only increase), $\exists j, k : a_{jk}^{(i+1)} > a_{jk}^{i}$ (the number of instances of at least one MS must increase in configuration C_{i+1} compared to config C_i), $\forall j, \sum_{k=1}^{n} a_{jk} = m_j$ (all MSs instances must be allocated to a VM).

Following [6], we have generated MBSAs randomly. Each topology is composed by a random number N_{ms} of MSs that follows a Poisson distribution of parameter μ. MSs themselves, are characterized by their average *service demand* D_k, that is computed as the product of an average number of visits v_k, and the average time spent in service at each visit S_k, that is:

$$D_k = v_k \cdot S_k, \quad 1 \leq k \leq N_{ms}$$

Visits are randomly determined according to a rule that is based on the Zipf

[4] We have used the notation $\bullet^{(i)}$ to denote the $\bullet$ element of the tuple corresponding to configuration C_i.

law, which is defined using four parameters:

$$v_k = \frac{c}{(k + q + \beta \cdot \mathbf{u})^s} \tag{1}$$

where k is the index of the considered MS, c is the *scale parameter* (that defines the order of magnitude of the visits), s is the *shape parameter* (that defines the ratio between the more popular and less popular MSs), q is the *shift parameter* (that fine tunes the range of the visits), β is the *randomness parameters* (if $\beta = 0$, visits are determined entirely by the Zipf law, otherwise they are randomly modulated), and $\mathbf{u}$ is a random number in the $[0, 1]$ range. In this way, MSs characterized by a lower index k are more popular and receives more visits.
Service demands are instead randomly determined from instances of an Erlang distribution, summarized by a rate parameter γ and a number of stages equal to K_S.

3.1 First step: autoscaling strategies

The considered autoscaling strategies model and evaluate the effects of different allocation and consolidation[5] policies for MSs, in order to understand the impact of this choice on the usage profile of a set of VMs. In particular, the main point is the evaluation of the influence of allocation and consolidation on relevant indexes. Relevant indexes are, for the goals of our work, the required number of VMs as function of the workload and the utilization of the VMs.

Autoscaling strategies are characterized by two important features: the *initial allocation*, and the *consolidation* policies. The initial allocation policy defines the main logic according to which the different MSs are mapped onto available VMs when the system starts the operations. The consolidation policy defines how the MSs are mapped onto available VMs according to the dynamics of the workload, i.e. the growth or decrease of demand for the various instances for the different MSs.

Given a set of autoscaling strategies, it is possible to compare them by looking at the number of VMs needed and their utilization, as a function of the workload. We have then fixed a maximum arrival rate the system might have to serve, λ_{Max}, and used the strategies to generate the corresponding set of configurations S. In particular, we have generated $|S|$ configurations such that $\lambda_U^{(|S|-1)} \leq \lambda_{Max} < \lambda_U^{(|S|)}$.

In the following, we will consider three autoscaling strategies.

Strategy A: **one MS per VM**

The first strategy follows a trivial approach, used as a baseline for a comparison with the others: it consists in simply mapping one MS on one VM. The initial allocation, for a configuration composed by N_{ms} MSs characterized by demand vector $\mathbf{D}$, is described in Algorithm 1, where function $\texttt{eye}(N_{ms})$ returns a vector of size N_{ms} components, all equal to 1, and $\texttt{ones}(N_{ms})$ returns an identity matrix of size N_{ms}. Parameter $U_{\max}$ represents the maximum average utilization a VM is allowed to

[5] In the following, the term *consolidation* refers to the execution of two or MSs on a single VM, in analogy with its current use, that refers to the execution of two or more virtual machines on a single physical machine.

have. Basically, applying the utilization law, it determines the maximum workload the initial configuration is able to serve ($\lambda_U^{(0)}$), then it creates the configuration where the number of virtual machines $n = N_{ms}$, i.e. each MS runs as a single instance and it is allocated on a different VM.

Algorithm 1 autoScalingOneMsPerVM.init($\mathbf{D}, N_{ms}, U_{\max}$)

```
1: $\lambda_U^{(0)} = \dfrac{U_{\max}}{\max_{1 \le k \le N_{ms}} D_k}$;
2: **return** $C_0 = \Big(0, \lambda_U^{(0)}, N_{ms}, \mathtt{ones}(N_{ms}), \mathtt{eye}(N_{ms})\Big)$;
```

Algorithm 2, describes how the next configuration C_{i+1} can be determined from configuration C_i. The extra parameter $U_{\min}$ defines the minimum utilization the bottleneck server can have before the system can downgrade to the previous configuration. The algorithm first determines the bottleneck VM j in configuration C_i (line 1); then it increases the number of instances of the corresponding MS (line 2), and it determines using the utilization law the new values of $\lambda_L^{(i+1)}$ and $\lambda_U^{(i+1)}$ (line 3). Since in this strategies, each MS is allocate on a new VM, the new configuration will be characterised by $n^{(i+1)} = n^{(i)} + 1$ and $A^{(i+1)} = \mathtt{eye}(n^{(i)} + 1)$.

Algorithm 2 autoScalingOneMsPerVM.consolidation($C_i, \mathbf{D}, N_{ms}, U_{\min}, U_{\max}$)

```
1: $j = \operatorname{argmax}_{1 \le k \le N_{ms}} \left( \dfrac{D_k}{m_k^{(i)}} \right)$;
2: $\mathbf{m}^{(i+1)} = \mathbf{m}^{(i)}$; $m_j^{(i+1)} = m_j^{(i)} + 1$;
3: $\lambda_U^{(i+1)} = \dfrac{U_{\max}}{\max_{1 \le k \le N_{ms}} \left( \dfrac{D_k}{m_k^{(i+1)}} \right)}$; $\lambda_L^{(i+1)} = \dfrac{U_{\min}}{\max_{1 \le k \le N_{ms}} \left( \dfrac{D_k}{m_k^{(i+1)}} \right)}$;
4: **return** $C_{i+1} = \Big(\lambda_L^{(i+1)}, \lambda_U^{(i+1)}, n^{(i)} + 1, \mathbf{m}^{(i+1)}, \mathtt{eye}(n^{(i)} + 1)\Big)$;
```

Strategy B: **start with one MS per VM, but then try to consolidate MSs on less utilized VMs**

The second strategy set has the same allocation strategy of the first (Algorithm 1), with one VM for each MS, but it uses consolidation to limit the requests for more VMs when the workload increase: a new VM is requested only when all existing VMs are saturated. This is described in Algorithm 3. In this case, an extra parameter U_C is required, to control how consolidation is operated. In particular, the procedure works in this way: whenever a MS is replicated, it tries to start it on the VM that is currently less utilized. If after adding the new instance of the bottleneck MS, the VM remains with a utilization lower than U_C, consolidation is actually performed. Otherwise, if the machine with the smallest workload, after starting the new instance of the considered MS would jump to a utilization level greater than U_C, then consolidation is not performed, and the new instance of the

MS is started on a new VM. This allows to not saturate a VM with consolidation, and avoid Zeno behaviours that could continuously replicate MS, without actually increasing the capacity of the system. In order to determine which MS to replicate in a configuration, the algorithm first determines the most utilized VM h (line 1), and then the MS j that has the most impact on the workload of the node (line 2). Then it looks at the best candidate VM c where to place the replica of MS j, as the least loaded VM with this new replication scheme $\mathbf{m}^{(i+1)}$ (line 4). Consolidation occurs only if the selected VM c, after hosting MS j, will not exceed the U_C utilization threshold with the workload that caused the system to enter configuration C_{i+1}, that is $\lambda_U^{(i)}$ (line 5). In this case the number of VMs do not change, and the allocation matrix A increases of one unit the element a_{jc} corresponding to the starting of another instance of MS j on VM c (line 6). Otherwise, a new VM is started, and MS j is allocated in this newly-created VM (line 8). In this case, function `add1col` $\left(A^{(i)}\right)$ adds one extra column to matrix $A^{(i)}$. The new limits for the workload are again computed applying the utilization law to all VMs running in configuration C_{i+1} (line 10).

Algorithm 3 autoScalingConsMSonVMs.consolidation$(C_i, \mathbf{D}, N_{ms}, U_{\min}, U_{\max}, U_C)$

```
1:  $h = \operatorname*{argmax}_{1 \le l \le n^{(i)}} \left( \sum_{k=1}^{N_{ms}} \frac{a_{kl}^{(i)} \cdot D_k}{m_k^{(i)}} \right);$
2:  $j = \operatorname*{argmax}_{1 \le k \le N_{ms}} \left( \frac{a_{kh}^{(i)} \cdot D_k}{m_k^{(i)}} \right);$
3:  $\mathbf{m}^{(i+1)} = \mathbf{m}^{(i)};\ m_j^{(i+1)} = m_j^{(i)} + 1;$
4:  $c = \operatorname*{argmin}_{1 \le l \le n^{(i)}} \left( \sum_{k=1}^{N_{ms}} \frac{a_{kl} \cdot D_k}{m_k^{(i+1)}} \right);$
5:  if $\left( \sum_{k=1}^{N_{ms}} \frac{a_{kc} \cdot D_k}{m_k^{(i+1)}} + \frac{D_j}{m_j^{(i+1)}} \right) \lambda_U^{(i)} < U_C$ then
6:      $A^{(i+1)} = A^{(i)};\ n^{(i+1)} = n^{(i)};\ a_{jc}^{(i+1)} = a_{jc}^{(i)} + 1;$
7:  else
8:      $A^{(i+1)} = \texttt{add1col}\left(A^{(i)}\right);\ n^{(i+1)} = n^{(i)} + 1;\ a_{jn^{(i+1)}}^{(i+1)} = 1;$
9:  end if
10: $\lambda_U^{(i+1)} = \frac{U_{\max}}{\max_{1 \le l \le n^{(i+1)}} \left( \sum_{k=1}^{N_{ms}} \frac{a_{kl}^{(i)} \cdot D_k}{m_k^{(i)}} \right)};\ \lambda_L^{(i+1)} = \frac{U_{\min}}{\max_{1 \le l \le n^{(i+1)}} \left( \sum_{k=1}^{N_{ms}} \frac{a_{kl}^{(i)} \cdot D_k}{m_k^{(i)}} \right)};$
11: return $C_{i+1} = \left( \lambda_L^{(i+1)}, \lambda_U^{(i+1)}, n^{(i+1)}, \mathbf{m}^{(i+1)}, A^{(i+1)} \right);$
```

Strategy C: **start with MSs consolidated on VMs, and then try to consolidate MSs on less utilized VMs**

The third strategy set has the same consolidation strategy as the one presented in Alogrithm 3, but it uses a different initial allocation strategy to consolidate the MSs on the VMs even with a small workload. This technique, as described in Algorithm

4, requires an additional parameter α_{VM} that defines the ratio of initial number of VMs with respect to the number of MSs (line 1). The algorithm starts allocating all the MSs on different VMs (line 2), and then it consolidates them until the target number of VMs is reached (line 3). Consolidation is performed by first finding the two less loaded VMs h and l (line 4). Then the MSs running on the two VMs are consolidated on a single VM, and one VM is released (line 6). In this case, the notation $A_{:h}$ refers to the h-th column of matrix A, and function `rem1col`(A, l) removes column l from matrix A. At the end, the upper limit to the workload the configuration can handle is computed using the utilization law (line 8).

Algorithm 4 autoScalingConsMSonVMs.int$(C_i, \mathbf{D}, N_{ms}, U_{\max}, \alpha_{VM})$

```
1: $n^{(0)} = \lceil \alpha_{VM} \cdot N_{ms} \rceil$;
2: $n = N_{ms}$; $A = \texttt{eye}(N_{ms})$;
3: while $n > n^{(0)}$ do
4:     $[\mathrm{h,l}] = \left\{ h \neq l | \forall t \neq h, t \neq l : \sum_{k=1}^{N_{ms}} a_{kt} D_k \geq \sum_{k=1}^{N_{ms}} a_{kh} D_k \wedge \sum_{k=1}^{N_{ms}} a_{kt} D_k \geq \sum_{k=1}^{N_{ms}} a_{kl} D_k \right\}$
5:                 (with $1 \leq h, l, t \leq n$);
6:     $A_{:h} = A_{:h} + A_{:l}$; $A = \texttt{rem1col}(A, l)$; $n = n - 1$;
7: end while
8: $\lambda_U^{(0)} = \dfrac{U_{\max}}{\max\limits_{1 \leq l \leq n^{(0)}} \left( \sum_{k=1}^{N_{ms}} \dfrac{a_{kl}^{(i)} \cdot D_k}{m_k^{(i)}} \right)}$;
9: return $C_0 = \left(0, \lambda_U^{(0)}, n^{(0)}, \texttt{ones}(N_{ms}), A\right)$;
```

3.2 Evaluating the autoscaling strategies

We have generated several MBSA demands using the following parameters: $\mu = [5 \ldots 100]$, $c = 6$, $s = 1.5$, $q = 2$, $\beta = 1$, $\gamma = 25$ and $K_S = 4$. The thresholds used to generate the set of configurations S are respectively $U_{\min} = 0.65$, $U_{\max} = 0.8$, and $U_C = 0.8$. Each MBSA is characterized by different number of MS (N_{MS}) and different demand vectors $\mathbf{D}$. Figure 1 shows how the demands change for the different MSs for five MBSA with $\mu = 10$ (a) and $\mu = 50$ (b). As it can be seen, although the Zipf law makes the demand decrease as the id of the MS increases, the randomness created by having set parameter $\beta = 1$ does not make them a monotonic function.

Figure 2 shows the evolution of the number of VMs required to handle a given workload λ for topologies generated with a different average number of MSs $\mu = [5 \ldots 100]$. To show the differences that applications generated with the same μ have, the plot shows three different traces for $\mu = 10$. The assumption that MSs are deployed initially on different VMs makes the curves depend on the total number of MSs for very small workload. Instead, as the workload increases, the number of VMs tends to be proportional to the total demand of the system, $D = \sum_{i=1}^{N_{MS}} D_i$. Due to the Zipf assumption, D increases very slowly as function of μ. For these reasons,

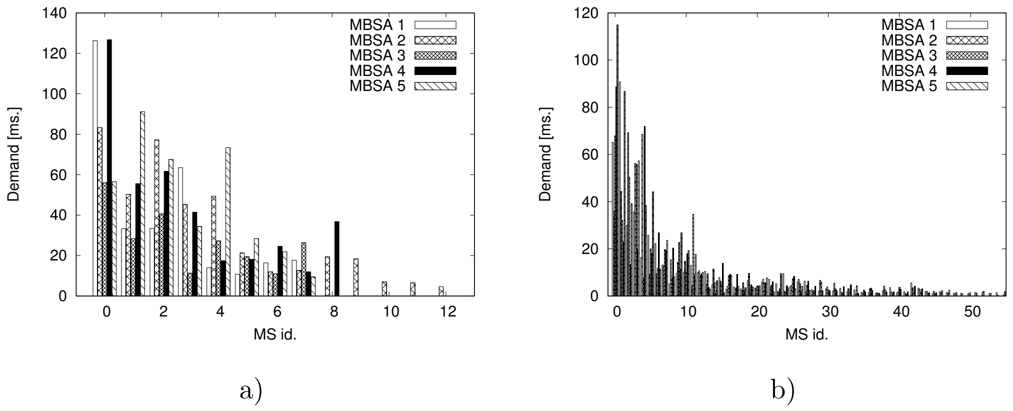

Fig. 1. Demands characterizing two generated MBSA: a) $\mu = 10$, b) $\mu = 50$.

in the following, we will focus only on two sample MBSAs generated respectively with $\mu = 10$ and $\mu = 100$.

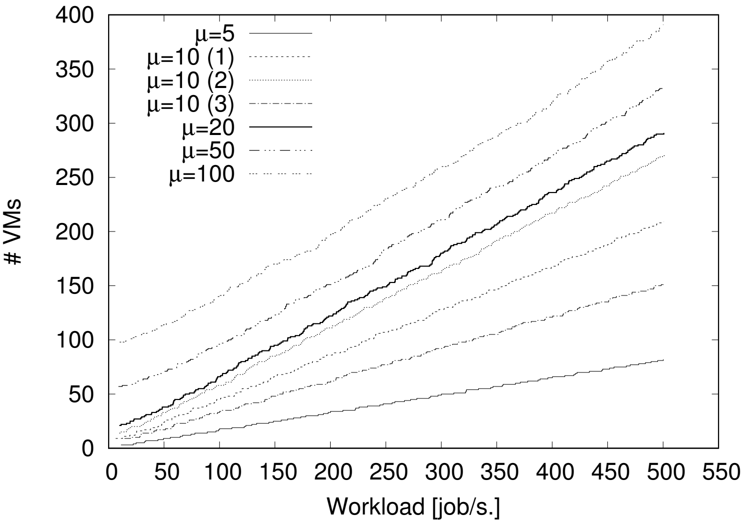

Fig. 2. Required VMs for Strategy A (no consolidation) as function of the workload, for different average number of MS μ. For the case $\mu = 10$, the value for three different traces is shown.

The three proposed scaling strategies are compared in Figure 3 and 4. When the workload is high compared to the capacity of the servers and a large number of instances is required to handle the requests, all policies behave more or less in the same way. Instead, when the traffic is low, consolidation policies are much more efficient with respect to the non-consolidating ones: this is particularly evident in Figure 3b, for the case with $\mu = 100$, and in the zoom provided in Figure 4 that considers a MBSA with $N_{MS} = 9$. However, from Figure 4, it is also clear that as soon as traffic increases a bit, the differences between the three policies become less and less evident, and with relatively high workloads (Figure 3a), strategy A that does not perform consolidation performs even better that some of the other policies.

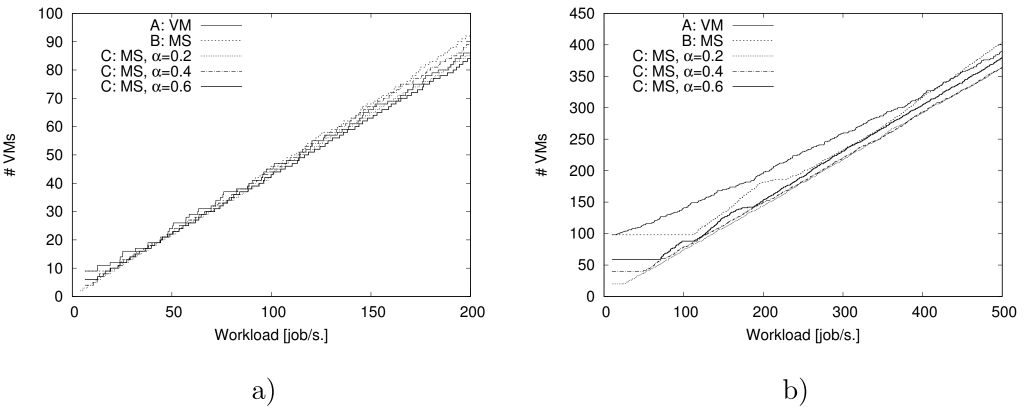

Fig. 3. Comparison of the different scaling strategies: a) $\mu = 10$, b) $\mu = 100$.

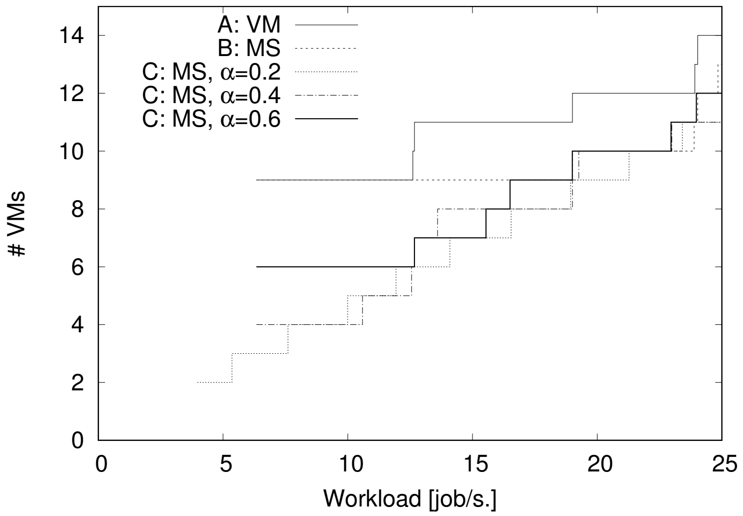

Fig. 4. Comparison of the different scaling strategies for a MBSA with $N_{MS} = 9$ and light workload.

The effect on the utilization of the VMs of the different policies is studied in Figure 5, where the minimum, average and maximum utilization per VM are shown. Utilization of a VM h is computed considering its total demand of the selected VM in the first configuration C_i that can handle the target workload λ, and applying the Utilization Law:

$$U_h(\lambda) = \lambda \sum_{k=1}^{N_{ms}} \frac{a_{kh}^{(i)} \cdot D_k}{m_k^{(i)}}, \qquad i : \lambda_U^{(i-1)} < \lambda \leq \lambda_U^{(i)} \tag{2}$$

As it can be seen, non-consolidating policies (Figure 5a and d) provide even a more evident sharing of the workload among the configurations. Instead, consolidation creates a large variability in the minimum utilization (Figure 5c and f): this is caused by the fact that when a new VM is started, it is usually assigned a MS that has already a large number of replica, and thus it receives a small amount of requests. This, however, is almost immediately corrected since the newly introduced VMs are targeted to host the consolidated replica of the next MS that becomes the bottleneck. For scenario B, the consolidation occurs only after the initial stage, this phenomenon on the minimum utilization occurs only when the workload reaches a certain level.

To summarize, from the previous analysis we can conclude that it is better to use consolidating policies (strategy C) when either the workload is very low, or the number of MSs is very high. Otherwise, the non-consolidating policy (strategy A) provides similar results in terms of required VMs, but a more uniform evolution in the utilization of the VMs, leading to more predictable performances.

3.3 Second step: cloud resource management policies

In the second step, the available cloud resources and their management policy are considered. As costs are dictated by the usage of resources in time and volume, the number of VMs in use are the most relevant cost factor. Two main cases are in the scope of our study: private clouds and public clouds.

In the case of private clouds, resources are owned by the organization that is running the application: consequently, the cost of a running VM does not depend on a fee connected to the type and number of used resources and the usage time according to a contract, but depends on the real expenses deriving by running the physical system that hosts the VMs. The minimization of costs is thus connected to power saving, rather than on the limitation of VMs usage.

In the case of public clouds, instead, costs are standardized according to the number of VMs used per time billing unit (e.g., one hour): the minimization of costs is thus connected to a better usage of the minimal number of VMs possible, but with the maximum exploitation of resources in the time billing unit.

The evaluation of the two different cases is performed by means of the FPN models depicted in Fig. 6 and Fig. 7.

Case I: **private cloud**

We will first consider the case of private clouds (Fig. 6). The current workload of the considered MS is expressed as the number of requests per second it has to serve. In the proposed model, this value corresponds to the marking of place `Load`. The workload evolves with time, and it might either increase or decrease: this is modeled by the two time-dependent fluid transitions `Increase` and `Decrease`. In our study, we will make them fire to make the marking of place `Load` follow the fluctuations of a publicly available workload trace. Marking of place `VMs` represents the number of currently deployed VMs. Let us call #`P` the marking of place `P`. The acquisition of a new resource is modeled by immediate transition `NewVM`: in particular it fires whenever the test arc that connects it to fluid transition `Load` detects that the workload has exceeded the upper threshold $\lambda_U(\#\texttt{VMs})$ for the configuration C_i with the maximum number of MSs instances, and a number of VMs $n^{(i)} = \#\texttt{VMs}$ equal to the one currently running (test arc that connects it to place `VMs`). In the same

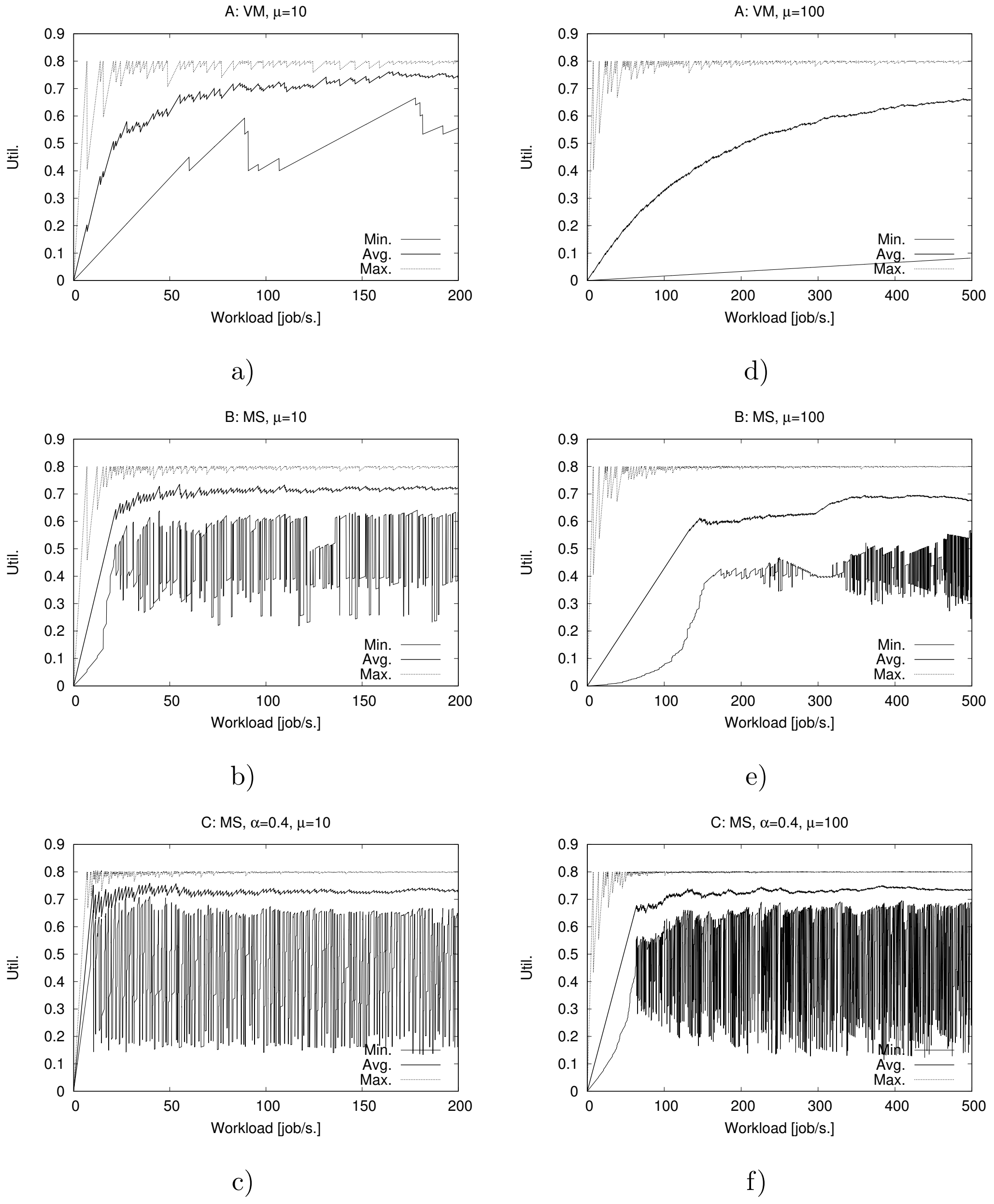

Fig. 5. Average, minimum and maximum utilization of the VMs: a,b,c) $\mu = 10$, d,e,f) $\mu = 100$; a,d) Strategy A - no consolidation, b,e) Strategy B - initially one VM per MS, then consolidation, c,f) start consolidated at $\alpha_{VM} = 0.4$, then continue consolidating resources.

way, immediate transition `Release` models the de-provisioning of a VM, and fires whenever it detects that the workload (connection with the inhibitor arc to place `Load`) is less than the threshold $\lambda_L(\#\texttt{VMs})$ of the configuration $C_{i'}$ with $n^{i'} = \#\texttt{VMs}$ virtual machines, and the least number of MSs instances (test arc that connects to place `VMs`). More formally:

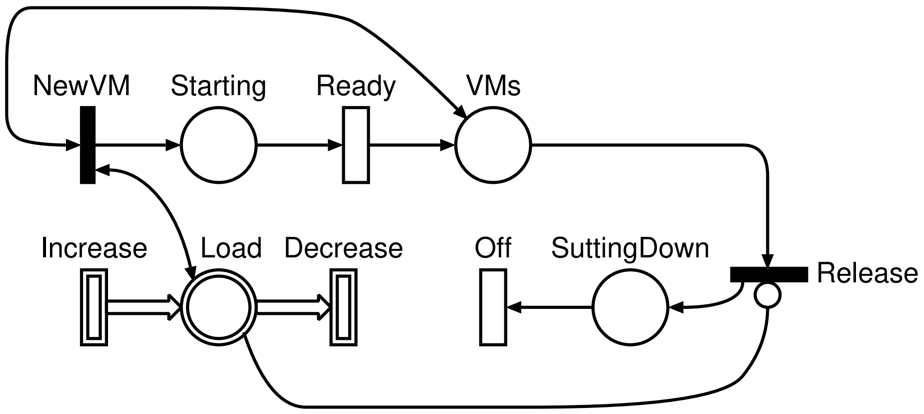

Fig. 6. Model for the description of the provisioning process in private clouds.

$$\lambda_U(n) = \{\lambda_U^{(i)} \in C_i | n^{(i)} = n \wedge n^{(i)} + 1 = n^{(i+1)}\}, \tag{3}$$

$$\lambda_L(n) = \{\lambda_L^{(i')} \in C_{i'} | n^{(i)} = n \wedge n^{(i')} = n^{(i'-1)} + 1\} \tag{4}$$

VMs are characterized by a startup time T_{up} and a shutdown time T_{down}. In this phases, VMs are running, but cannot be used to serve any of the incoming traffic. Since they are running, they consume energy: note that the instances of the MSs running over them are considered to be inactive and do not share the workload with the other instances. Startup phase is modeled by deterministic transition `Ready` (characterized by firing time T_{up}) and place `Starting`. Shutdown is modeled by deterministic transition `Off` (with firing time T_{down} and place `SuttingDown`). Both transitions are infinite server, since more VMs can be starting or shutting down at the same time.

Since users in private cloud pay for the energy required to run the VMs, the corresponding strategy aims at starting up the VMs as late as possible, and to shut them down as early as possible. Note also that the presence of startup times can lead to moments in which the system is unstable, that is number or requests arriving per second is greater than the one that the system can serve.

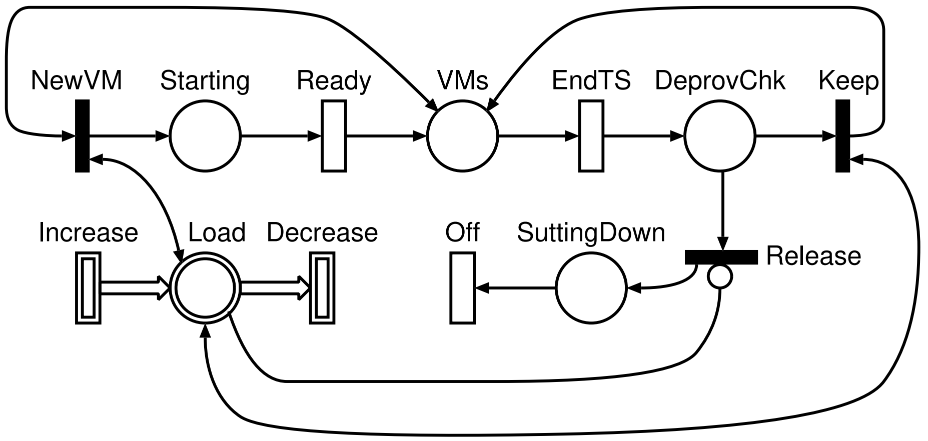

Fig. 7. Model for the description of the provisioning process in public clouds.

Case II: **public cloud**
Next, we describe VM provisioning when using public clouds (Fig. 7). Workload evolution and provisioning of VMs works exactly as for private clouds (Places `Load`, `Starting` and `VMs`, transitions `Increase`, `Decrease`, `NewVM` and `Ready`). Deprovisioning instead, accounts for the fact that once a VM has been acquired, it is charged in a time-quantized way: for example, even if a VM is used for just a few minutes, it is charged for an entire hour. For this reason, once a VM has been started, it can be used to improve the performance of the application, even if not strictly necessary to support the current workload. The VM could then be released at the end of the billing period if the workload has not increased in the meantime. This process is modeled by the infinite server deterministic transition `EndTS`, whose firing time T_{bill} corresponds to the length of the billing interval applied by the public cloud provider. As soon as a VM is ready, every T_{bill} it is checked whether to be kept or released: transition `EndTS` fires and moves the token corresponding to the VM in place `DeprovChk` to check whether it could be released or it should be kept. If the current workload is less than $\lambda_L(\#\texttt{VMs})$ (the inhibitor arc that connects to fluid place `Load`), then the VM could be released and immediate transition `Released` fires. If instead, the workload is greater than $\lambda_L(\#\texttt{VMs})$, immediate transition `Keep` fires (thanks to the test arc that connects it to fluid place `Load`). In this case, the token is immediately returned to place `VMs`, keeping the number of provisioned VMs constant. When a VM is released, a token is moved to place `ShuttingDown` where it remains for the shutdown time modeled by transition `Off`: this behaviour is identical to the one seen in private clouds.

3.4 *Evaluation of the cloud resource management policies*

We can now use the previously presented models to evaluate the related performance metrics. Interesting indexes are the energy cost (private cloud) or billing cost (public cloud case) that characterize a policy, the response time of MSs, and the stability of the system as of the workload of the system evolves. In particular, we feed the sets of configurations $S = \{C_i\}$ defined in Sec. 3.1 to define the thresholds of the test and inhibitor arcs of the FPNs presented in Sec. 3.3.

Fluid transitions `Increase` and `Decrease` are programmed to mimic a realistic variable stream of requests. In particular, they follow the traffic data from the Olympic Web site from February 9, 1998 through February 16, 1998 as presented in [11], and scale such trace with an appropriate constant to make it in the workload range intended for the type of MBSA considered in this work. Figure 8 shows (on the right axis) the number of requests per second $\Lambda(T)$ at time T used in our experiments. Let us call #`P(T)` the marking of place `P` (either fluid or discrete) at time T. The marking of the FPN models evolves such that:

$$\#\texttt{Load}(T) = \Lambda(T) \tag{5}$$

We start focusing on the private cloud case modeled in Figure 6. Figure 8 also shows the number of VM required to handle the considered workload (marking #`VMs(T)`) when the setup and de-provisioning times of the VMs are supposed to be negligible ($T_{up} = T_{down} = 0$). The considered MBSA has been generated with the parameters given in 3.1 and $\mu = 10$. Following the discussion in 3.2, strategy A uses the most number of VMs, while strategy B has a lower adaptation speed when the requests reduce.

Next we consider the effects of provisioning and release times by setting $T_{up} = 40$

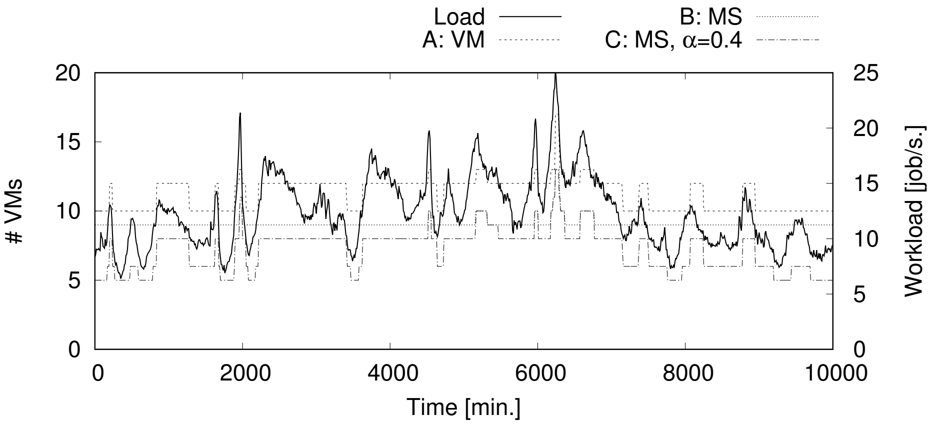

Fig. 8. The system workload coming from the Olympic Web site from February 9, 1998 through February 16, 1998 used as a guideline to define a variable workload (right axis), and the minimum number of VMs required according to the three considered autoscaling scenarios (left axis) in the private cloud scenario.

min. and $T_{down} = 2$ h. Such values have been slightly enlarged compared to actual durations that can be experienced in common cloud solutions, to emphasize their effect. Results for the different autoscaling scenarios are shown in Figure 9 for the time range $T \in [5000, 8000]$. The curve called *Target* refers to the case when the setup and shutdown time are negligible (as in Figure 8). Curve *Active* instead corresponds to the number of VMs currently running ($\#\texttt{VMs}(T)$), while line *Current* refers to the total number of VMs that are currently consuming energy ($\#\texttt{Starting}(T) + \#\texttt{VMs}(T) + \#\texttt{SuttingDown}(T)$). The presence of startup and shutdown time creates a delay which in some cases might make a VM available to support the MSs only after the workload has already reduced. For this reason, policies with a lower hysteresis, such as strategy B, provide a better support to the variable workload.

To further investigate the effects of the autoscaling strategies and the provisioning and de-provisioning times, we compute the average response time as function of time. In particular, we assume the MBSA as a separable queuing network, and we compute its average system response time $R(T)$ as:

$$R(T) = \sum_{h=1}^{n^{(i)}} \frac{\displaystyle\sum_{k=1}^{N_{ms}} \frac{a_{kh}^{(i)} \cdot D_k}{m_k^{(i)}}}{1 - \#\texttt{Load}(T) \displaystyle\sum_{k=1}^{N_{ms}} \frac{a_{kh}^{(i)} \cdot D_k}{m_k^{(i)}}}, \qquad i = \#\texttt{VMs}(T) \tag{6}$$

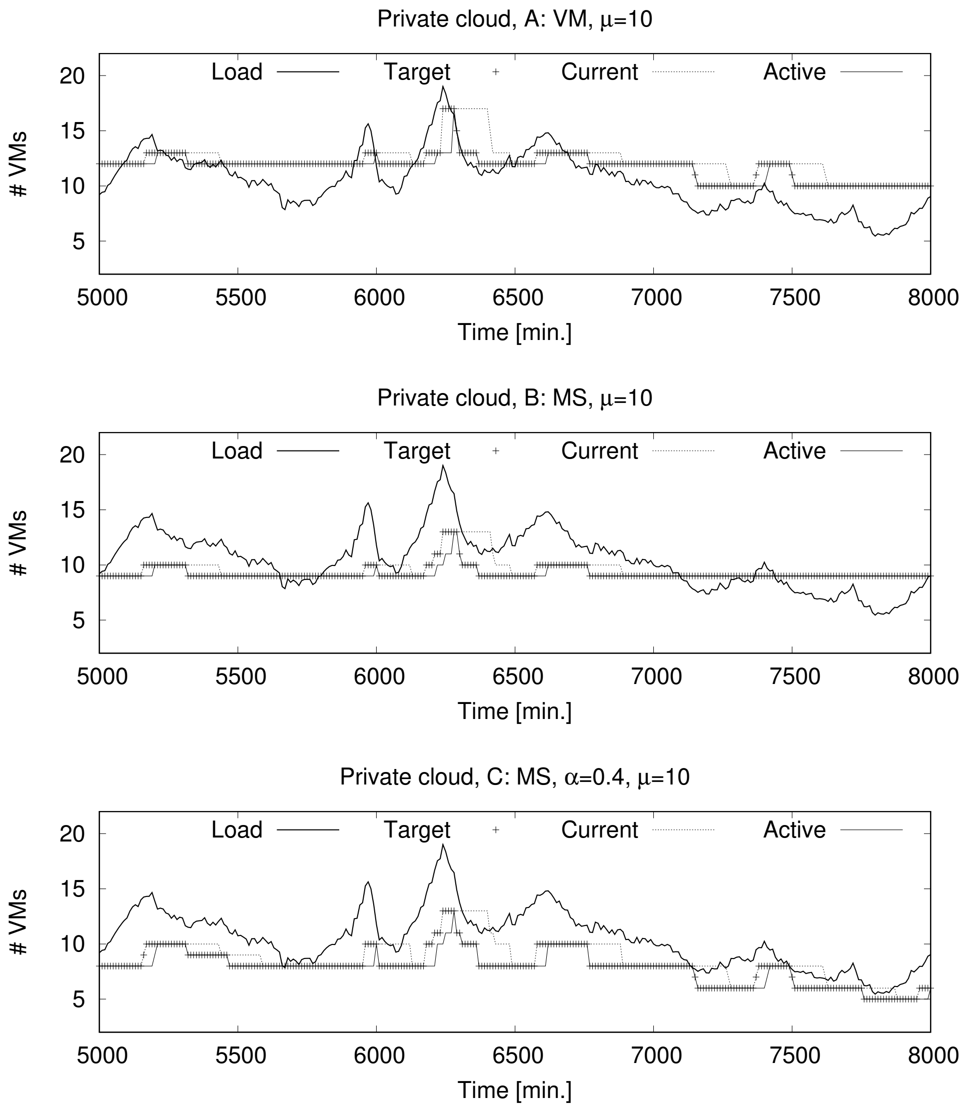

Fig. 9. Effects of VMs startup and shutdown times on the number of active and currently running VMs for the considered autoscaling scenarios.

where the numerator computes the demand of each VM, and the denominator accounts for one minus its utilization. Results are shown in Figure 10. The policy that replicates each MS on a proprietary VM (strategy A) is the one that gives the users the best response time. On the contrary, the full consolidation policy (strategy C), is affected by the higher load VMs are experiencing providing poor performances, which in some cases might lead to system instability (shown in the Figure with vertical lines).

We then focus on the power consumption, applying the simple rule given in [9]:

$$P_{VM} = P_{Idle} + U_{VM} \cdot (P_{Max} - P_{Idle}) \tag{7}$$

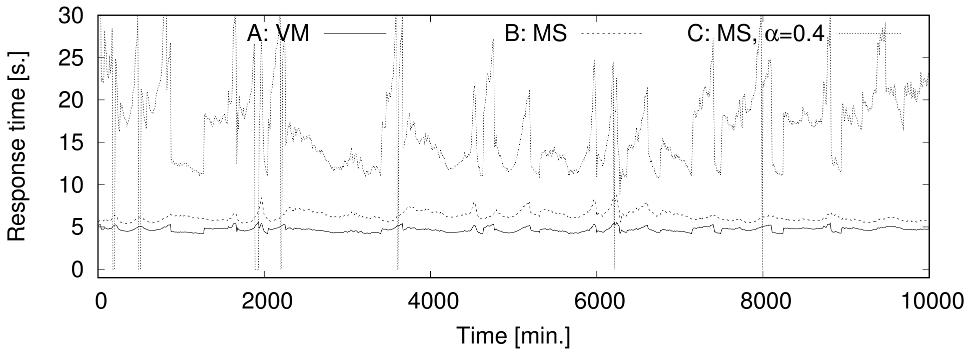

Fig. 10. Response time of a MBSA when run on a private cloud.

where the power consumed by a VM is approximated as a constant contribution (P_{Idle}), plus a term that depends on the utilization U_{VM} and P_{Max} - the maximum power that a VM can require. In this context we set $P_{Idle} = 16.5$ Watt and $P_{Max} =$ 34.5 Watt, we suppose that VMs starting or shutting down requires P_{Idle} Watt, and estimate the power consumption $P(T)$ of the MBSA at time T as:

$$\begin{aligned} P(T) &= u \cdot P_{Idle} + \#\mathtt{Load}(T) \cdot (P_{Max} - P_{Idle}) \sum_{h=1}^{n^{(i)}} \sum_{k=1}^{N_{ms}} \frac{a_{kh}^{(i)} \cdot D_k}{m_k^{(i)}}, \\ i &= \#\mathtt{VMs}(T), \\ u &= \#\mathtt{Starting}(T) + \#\mathtt{VMs}(T) + \#\mathtt{SuttingDown}(T) \end{aligned} \tag{8}$$

Results are shown in Figure 11. It is interesting to see that with the given values of P_{Idle} and P_{Max}, the consolidated case, due to its higher utilization, experiences a higher power consumption with respect to the non-consolidate strategy, even if it uses a lower number of VMs.

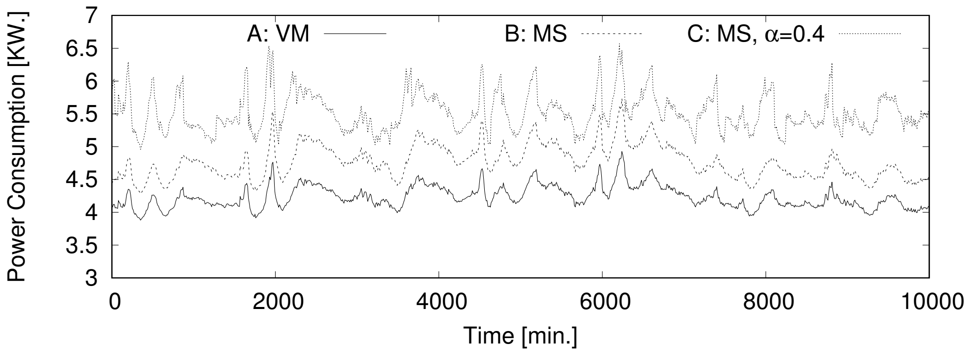

Fig. 11. Power consumption of a MBSA when run on a private cloud.

Finally we study the public cloud case modeled in Figure 7. Figure 12 shows the number of VMs required to support the MBSA. In particular, curve *Active* refer to the number of VMs that would be available using the same strategy as private clouds, while *Pay Opt.* shows the number of VMs that can be used without increasing the cost, by exploiting the time-slot based billing policy usually applied by providers. In this study, we have set $T_{bill} = 8$ h., again slightly larger than the one commonly used by cloud providers, to emphasize its effect. As it can be seen, in many cases keeping the VMs active until the next billing period, can leave the system ready to accomodate new workload fluctuations and generally give more resources than the one actually needed, allowing to provide a better quality of service.

Figure 13 shows the time that must be purchased from the provider (identified

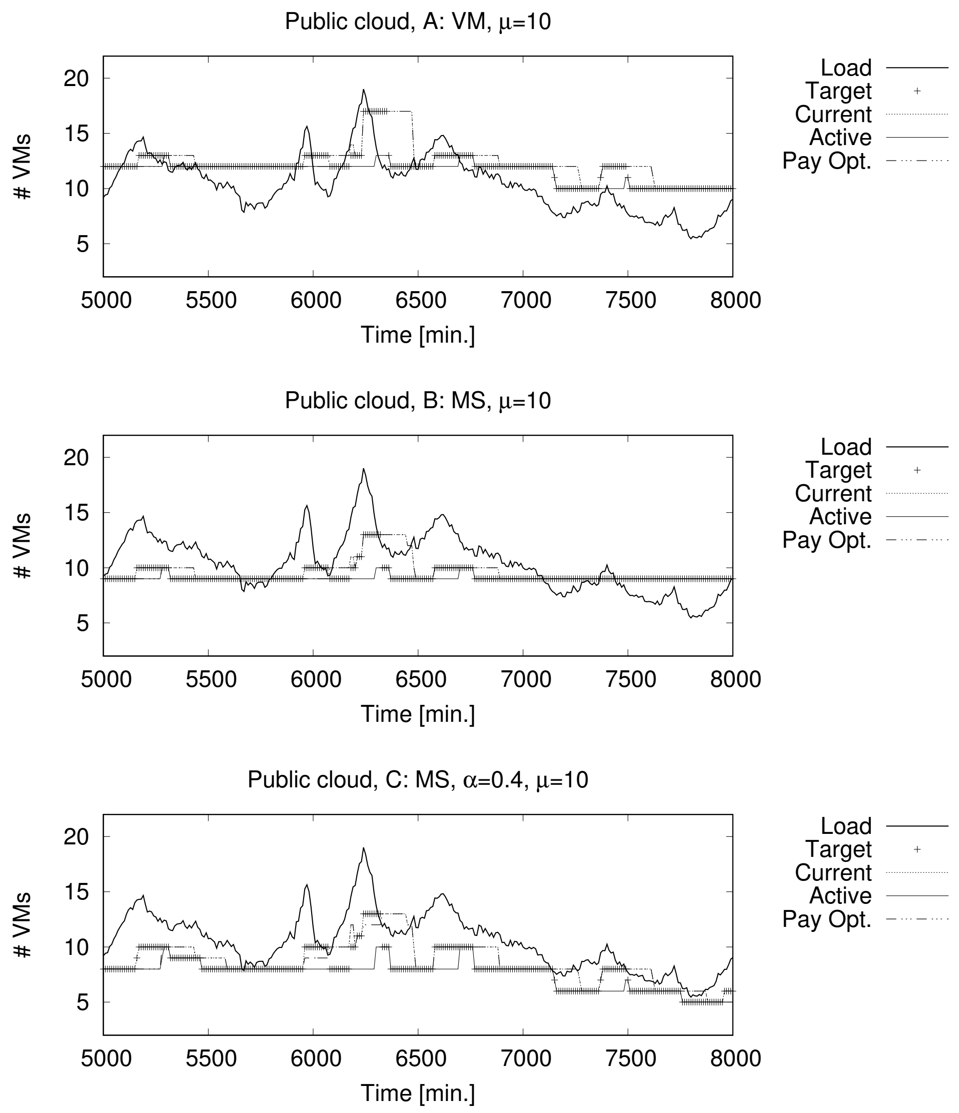

Fig. 12. L5.

by the term *pay*) in order to provide the required service. The minimum time that could be purchased, if charged at the seconds level, is instead shown by curves identified by *run*. As expected, actual purchased VM time grows as a step function, due to quantization of billing periods. As expected, the full consolidation policy, strategy C, is the one that provides the least cost in term of VMs running hours, and the non-consolidating policy is the one that results to be the most expensive.

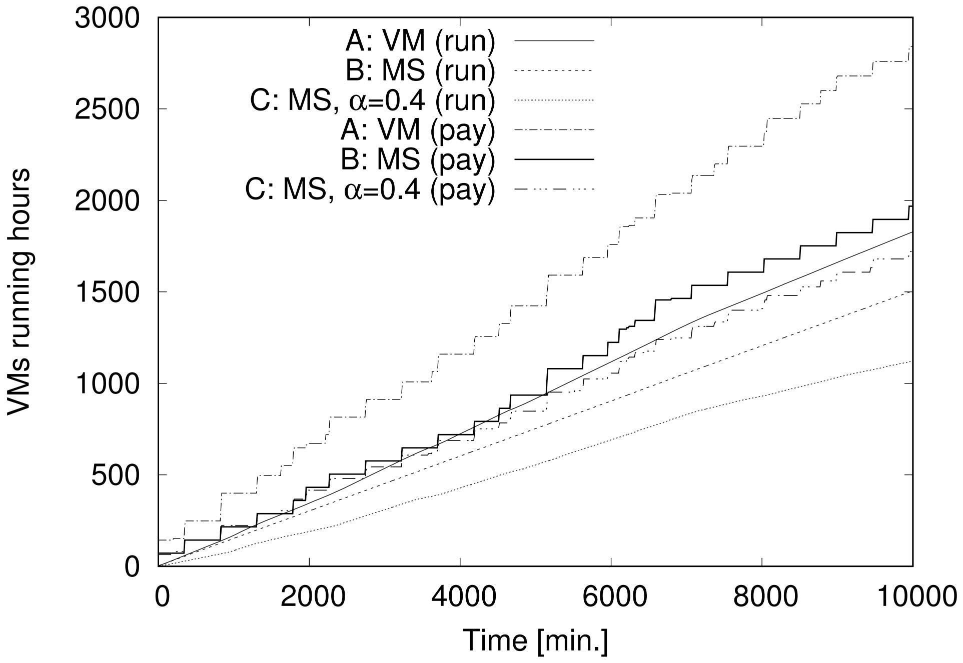

Fig. 13. VM running hours for the different autoscaling strategies in public clouds.

4 Conclusions and future work

In this paper, we propose a framework able to evaluate the performances, costs and energy consumptions related to the provisioning of a MBSA. The resulting model allows to study such a systems from both the service provider and infrastructure point of view. In particular, the autoscaling processes are described by pseudo-code algorithms that lead to evaluate different strategies. It emerges that consolidating strategy works better if the load is rather low or the number of MSs is relevantly high. Otherwise, non-consolidating policy has the advantage to supply more predictable behaviors of the architecture.
The infrastructure analysis has been performed by utilization the provisioning schemes with a FSPN where the available cloud resources and their management policy are considered. In order to evaluate the cost factors affected by resources utilization, two cases have been compared: private clouds and public clouds. The derivation of interesting indexes, as the energy cost for the private cloud and billing cost for the public cloud case together with the workload of the VMs and the execution time performed by the MSs, supplies an accurate analysis of MBSAs evolution. In the case of private clouds, the costs are mainly related to power concerns whereas

in public clouds the number of VMs used per time billing unit is the dominant factor.
We also evaluated a MBSA loaded with a realistic variable stream of requests, the traffic data from the Olympic Web site from February 9, 1998 through February 16, 1998. The proposed approach can be easily adapted to describe new scaling strategies and infrastructure cases, providing a flexible tool able to take into account many features of these complex systems.

5 Acknowledgments

This work is partially supported by EUBra- BIGSEA (690116), a Research and Innovation Action (RIA) funded by the European Commission under the Cooperation Programme, Horizon 2020 and the Ministrio de Cincia, Tecnologia e Inovao (MCTI), RNP/Brazil (grant GA-0000000650/04), and the European Commission under the grant ANTAREX H2020 FET-HPC-671623.

References

[1] *Microservices (a definition of this new architectural term)*, https://martinfowler.com/articles/microservices.html, accessed: 2017-01-25.

[2] Alshuqayran, N., N. Ali and R. Evans, *A systematic mapping study in microservice architecture*, in: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, 2016, pp. 44–51.

[3] Butzin, B., F. Golatowski and D. Timmermann, *Microservices approach for the internet of things*, in: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–6.

[4] Esposito, C., A. Castiglione and K. K. R. Choo, *Challenges in delivering software in the cloud as microservices*, IEEE Cloud Computing **3** (2016), pp. 10–14.

[5] Florio, L. and E. D. Nitto, *Gru: An approach to introduce decentralized autonomic behavior in microservices architectures*, in: *2016 IEEE International Conference on Autonomic Computing (ICAC)*, 2016, pp. 357–362.

[6] Gribaudo, M., M. Iacono and D. Manini, *Performance evaluation of massively distributed microservices based applications*, in: *Proc. of the 2017 ECMS*, 2017.

[7] Hasselbring, W., *Microservices for scalability: Keynote talk abstract*, in: *Proc. of the 2017 ASMTA*, 2016.

[8] Heorhiadi, V., S. Rajagopalan, H. Jamjoom, M. K. Reiter and V. Sekar, *Gremlin: Systematic resilience testing of microservices*, in: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 57–66.

[9] Ho, T. T. N., M. Gribaudo and B. Pernici, *Characterizing energy per job in cloud applications*, Electronics **5** (2016).

[10] Inagaki, T., Y. Ueda and M. Ohara, *Container management as emerging workload for operating systems*, in: *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[11] Iyengar, A. K., M. S. Squillante and L. Zhang, *Analysis and characterization of largescale web server access patterns and performance*, World Wide Web **2** (1999), pp. 85–100.
URL https://doi.org/10.1023/A:1019244621570

[12] Kang, H., M. Le and S. Tao, *Container and microservice driven design for cloud infrastructure DevOps*, in: *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 202–211.

[13] Kratzke, N., *Automated setup of multi-cloud environments for microservices applications*, in: *Proc. of the 2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, 2016.

[14] Leitner, P., J. Cito and E. Stöckli, *Modelling and managing deployment costs of microservice-based cloud applications*, in: *Proceedings of the 9th International Conference on Utility and Cloud Computing*, UCC '16 (2016), pp. 165–174.
URL http://doi.acm.org/10.1145/2996890.2996901

[15] Melis, A., S. Mirri, C. Prandi, M. Prandini, P. Salomoni and F. Callegati, *Crowdsensing for smart mobility through a service-oriented architecture*, in: *2016 IEEE International Smart Cities Conference (ISC2)*, 2016, pp. 1–2.

[16] Pahl, C., A. Brogi, J. Soldani and P. Jamshidi, *Cloud container technologies: a state-of-the-art review*, IEEE Transactions on Cloud Computing **PP** (2017), pp. 1–1.
URL http://ieeexplore.ieee.org/document/7922500/

[17] samuel Kounev, *Stochastic models for self-aware computing in data centers: Tutorial*, in: *Proc. of the 2017 ASMTA*, 2017.

[18] Sousa, G., W. Rudametkin and L. Duchien, *About microservices, containers and their underestimated impact on network performance*, in: *Proc. of the CLOUD COMPUTING 2015: The Sixth International Conference on Cloud Computing, GRIDs, and Virtualization*, 2015.

[19] Toffetti, G., S. Brunner, M. Blöchlinger, F. Dudouet and A. Edmonds, *An architecture for self-managing microservices*, in: *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, AIMC '15 (2015), pp. 19–24.
URL http://doi.acm.org/10.1145/2747470.2747474

[20] Ueda, T., T. Nakaike and M. Ohara, *Workload characterization for microservices*, in: *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[21] Villamizar, M., O. Garcs, H. Castro, M. Verano, L. Salamanca, R. Casallas and S. Gil, *Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud*, in: *2015 10th Computing Colombian Conference (10CCC)*, 2015, pp. 583–590.

[22] Villamizar, M., O. Garcs, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano and M. Lang, *Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures*, in: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 179–182.