PERFORMANCE EVALUATION OF MASSIVELY DISTRIBUTED MICROSERVICES BASED APPLICATIONS

Marco Gribaudo
DEIB
Politecnico di Milano
via Ponzio 51
20133, Milano, Italy
marco.gribaudo@polimi.it

Mauro Iacono
DMF
Università degli Studi della Campania "Luigi Vanvitelli"
viale Lincoln 5
81100 Caserta, Italy
mauro.iacono@unicampania.it

Daniele Manini
DI
Università degli Studi di Torino
corso Svizzera 185
10129, Torino, Italy
manini@di.unito.it

KEYWORDS

cloud infrastructures; data center performances; performance modeling; containers; microservices

ABSTRACT

Microservice-based software architectures are a recent trend, stemming from solutions that have been designed and experimented in big software companies, that aims to support devops and agile development strategies. The main point is that software architectures, similarly to what happens in SOA, are decomposed into very elementary tasks, that can be developed, maintained and deployed in isolation by small independent teams, and that compose an application by means of simple interactions. The resulting architecture is advocated to be more maintainable, less prone to failures, more agile, but obviously impacts on performances. In this paper we provide a simulation based approach to explore the impact of microservice-based software architectures in terms of performances and dependability, given a desired configuration. Our approach aims at giving a first approximation estimation of the behavior of different classes of microservice-based applications over a given system configuration, to characterize the infrastructure from the point of view of the service provider under a randomly generated realistic overall workload: to the best of our knowledge, there is not any other analogous decision support tool available in literature.

I. INTRODUCTION

Microservice-based software architectures are a technical solution that emerged from the industry sector to face the challenges that the market of cloud applications has created. Competition requires a continuous update and upgrade of applications that become larger and larger and serve simultaneously a very big number of users and requests, while the design, maintenance and deployment of larger and larger code bases results in more and more complex development and management cycles, exposing applications to a higher risk of fault propagation or more extended consequences of erroneous behaviors, due to coding errors. Microservice-based architectures support the decomposition of complex monolithic applications into a (high) number of very simple services, each responsible of elementary actions within the logic of the execution flow of an application, and interacting by means of simple (generally HTTP based or socket based) communication protocols.

A first advantage of this choice is a decoupling of microservices, each of which may be developed, maintained and administered by a different team, and each of which may be managed in a continuous integration mode, typical of agile development paradigms. Moreover, the small size of a microservice allow a small code base (based on an independent stack, seamlessly with respect to the stack used by the other microservices), a smaller team, and an easy integration of new team members with a shorter training.

A second advantage is the potential improvement of application resilience and scalability, as, being each service run independently, faults will not result in a crash of the whole application, and a different number of replicas of each microservice may be executed when and if needed, depending on the workload. Together with the use of containers, the increase in workload may be mitigated: anyway, in general, lower performances are reasonably expected with respect to monolithic applications.

While there is a simplification of the development cycle of microservices, the general structure of the application becomes more complex, because of the fact that planning and general design must suffer a lack of control, and, potentially, of optimization, on what is delegated to each microservice development team. The overall management of the execution environment and of the infrastructure, that is the most of what is left to the global level, heavily affects the general performances of the application, but with a high level of decentralization of responsibilities and control leverages. In this paper we investigate, with a stochastic simulation based

Proceedings 31st European Conference on Modelling and
Simulation ©ECMS Zita Zoltay Paprika, Péter Horák, Kata Váradi,
Péter Tamás Zwierczyk, Ágnes Vidovics-Dancs, János Péter Rádics (Editors)
ISBN: 978-0-9932440-4-9/ ISBN: 978-0-9932440-5-6 (CD)

modeling approach, the general behavior of microservice-based applications with respect to performances, dependability and scalability, on a given infrastructure. The goal is to obtain a conceptual tool to provide methodological guidelines for the management of the infrastructure.

This paper is organized as follows: next Section provides related works, while Section III gives an overall introduction to microservice-based software architectures and applications; Section IV describes the simulation approach and the modeling framework; Section V offers the results of the test experiments that have been performed; finally, conclusions close the paper.

II. RELATED WORKS

At the state, there is not a wide academic literature about microservices, while there is plenty of good technical references related to implementation, cases and practical issues. Microservices architectures, as they are intended in the currently agreed definition, have been introduced in [1]. For a fast and readable introduction to the main themes about microservices in the cloud we suggest the reader to refer to [2], while for a systematic mapping of existing literature about the microservice architecture we suggest [3], to which we also redirect the readers for a more extensive reference list. In [4] the authors discuss, with a quite complete and solid analysis of all the aspects related to the executing architecture, the workload characterization of microservice architectures, with an experimental approach that benchmarks a monolithic application versus two different microservice versions, one based on a monothread support and one based on a multithread support, with very interesting results. In [5] and [6] the analysis focus instead on the costs and the benefits of the deployment of monolithic versus microservices and of monolithic versus AWS Lambda versus microservices architectures, considering both cloud customer or cloud provider operated systems. In [7] the scalability of the Docker container is evaluated, in different conditions, considering it as a new type of system workload. A similar study has been developed in [8], that identifies the challenges for a full development of containers based systems. For what concerns the operating condition, [9] describes a proposal for resilience testing, while [10] formulates a proposal for a decentralized autonomic behaviour for microservice architectures. Finally, for what concerns applications, the web offers a lot of proposals and descriptions: we rather prefer here to refer to a couple of peer reviewed papers, [11] and [12], as a starting point for readers, for their clear and systematic presentation.

III. MICROSERVICE ARCHITECTURES

A microservices based software architecture is an application composed of a number of software services that may be independently deployed and that directly interact with one another with lightweight mechanisms [4]. Each microservice is executed in a separate process. In general, a microservice is executed as a native process of the host, by means of a container, that is an abstraction layer capable of virtualizing resources with a low, but non negligible, impact on

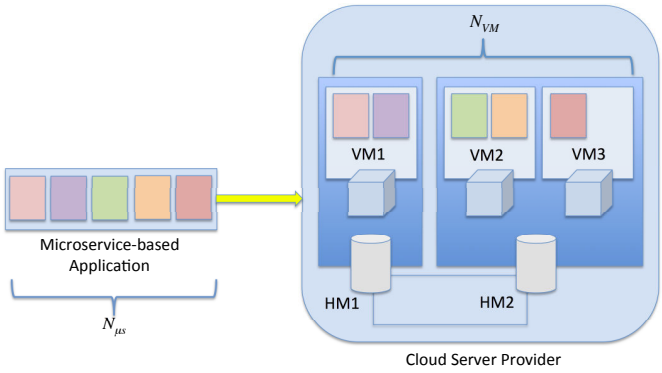

Fig. 1. General schema of a microservices based architecture

performances, and providing isolation. The technology stack may or may not allow multithreading, introducing a further element of complexity for the evaluation of performances: a multithreading solution may be more efficient by exploiting pooling, but suffers e.g. from blocking caused by I/O request waits; a single thread solution does not, but processes need more resources than threads. The use of containers allows a microservices based application to be seamlessly migrated as a whole, as it is commonly used for an agile deployment from the development to the production environment, that is generally cloud based. However, in the practice an application spans over a large number of different containers, that may theoretically get up to the number of microservices, to allow the enactment of agile development processes. This has an impact on performances as well, because the interactions between the microservices need a virtualized network between containers. Finally, the number of active containers is also a leverage to scale performances up and down when needed to fit the dynamics of the workload. As described in [2], Fig. 1 shows a graphical representation of the relations between microservices (μs), virtual machines (VM) and nodes (HM - Hardware Machines). As usual in cloud computing, each node can host several VMs and partition resources among them. Moreover, each VM can host several microservices that are used to implement the application.

Each container is executed within the operating system (OS) of which it virtualizes the resources, as a process, and provides its services by means of a client-server logic. When run in cloud environments, containers are executed within VMs. Consequently, containers are executed on the OS provided by a VM that in turn is executed on the computing nodes by means of the host OS or a hypervisor, eventually together with other VMs. The computing resources of the node, that is generally a multiprocessor and/or multicore architecture, are so managed in order to map each thread of a microservice to a core and each process to a processor.

Within the cloud infrastructure, VMs are managed according to the internal policies that provide elasticity and power management features. VMs may be migrated, shelved or launched when needed, similarly to what happens to threads

within containers and to containers within a VM. A correct estimation of the best policies for the provider, consequently, requires models that may allow to understand the overall effects of the interactions within and between the different levels, and evaluate the role of the various available parameters. We already dealt with performance modeling of cloud architectures [13] [14] and multithreaded applications [15]: in the following we will focus on the microservice architecture, including all the architectural details of the whole cloud stack that are relevant for performance evaluation.

IV. SIMULATION APPROACH AND MODEL

As, to the best of our knowledge, there is not at the moment a general simulation approach for microservice architectures, nor there are extensive characterizations of their parameters available, the simulation approach adopted in this paper is designed to produce a first approximation glance on the general behavior of these systems without a single reference scenario: the goal is to provide an estimation of performance and dependability for different configurations. Consequently, the approach is based on a parametric generation of a large number of different possible microservices applications, defined as oriented graphs, with a parametric random resource usage and fault probability per microservice, that are mapped onto a parametric architecture, in which the number of servers (in a cloud), the number of VMs per server, the number of containers per VM can be varied: moreover, a fault probability is assigned to every component of the architecture. The workflow of the approach is depicted in Fig. 2. A set of applications is generated, according to chosen parameters, by a *scenario generator*, that instantiates a simulation per case. Simulations are run by an event based *simulator* that has been specificly designed for this paper. The simulator produces performance and dependability metrics, and the results of simulations are then processed by a *statistics* processor that produces an overall performance profile of the given system configuration(s), described as a function of the different parameters and the scenarios.

More in details, in this work we used Montecarlo simulation to generate several random application topologies and study their performance and availability, and proper maximum entropy probability distributions, to avoid biasing due to the lack of assessed models, or the Zipf distribution in analogy to web traffic characterizations.

In each simulated scenario (see Fig. 1) an application is split in $N_{\mu s}$ microservices, and it can be executed by users at rate of λ_u requests per second. Microservices are allocated within proper containers in VMs executed on the top of host machines provided by a cloud infrastructure, the total number of available VMs, according to the contract, is denoted with N_{VM}. We defined most parameters with stochastic numbers generated with probability distributions. In this case study, we assume that the number of microservices $N_{\mu s}$ has a Poisson distribution with parameter $\lambda_{\mu s}$. Next, we consider that the number of VMs on which microservice containers are deployed N_{VM} is a fraction of β of $N_{\mu s}$, to represent that

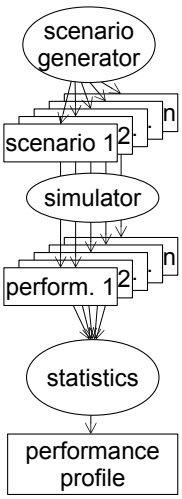

Fig. 2. The workflow of the simulation approach

subsets of containers can be allocated on the same VM. In particular, we define:

$$N_{VM} = \lceil \beta \cdot N_{\mu s} \rceil \tag{1}$$

Each microservice can be invoked a random number of times during the execution of the application. We assume this random number to be geometrically distributed, with an average v_i for each microservice i. We assume that not all the microservices are equally used in serving a request. Some can be essential, and will consequently be called several times during the execution (e.g. verification of user identity), while some other ones might be required only in special circumstances. We thus assign to each microservice i, $1 \leq i \leq N_{\mu s}$, a random average number of executions v_i, according to a popularity level. In particular, we assume that microservices with a lower index i are more popular (i.e. have a larger average number of executions) than services with a high index. Popularity follows a modified Zipf distribution, characterized by 4 parameters: c (the scale parameter), s (the shape parameter), q (the shift parameter) and α (the randomness parameters). Let $\mathbf{u}_i$ be a random number, uniformly distributed in the range $[0,1]$. We compute, for each randomly generated scenario, the average number of calls to a microservice i as:

$$v_i = \frac{c}{(i+q+\alpha \cdot \mathbf{u})^s} \tag{2}$$

We assume that the execution time for each microservice i is exponentially distributed with average S_i. In each scenario for any micorservice i the value of S_i is randomly defined according to an Erlang distribution with k_S stages, and average λ_S. In order to consider VM faults, Mean Time To Failure (MTTF) and Mean Time To Repair (MTTR) parameters are sampled for both the VMs (infrastructure faults) and the microservices (software faults). In particular, such parameters are $MTTF_{VM}$, $MTTR_{VM}$, $MTTF_{\mu s}$ and $MTTR_{\mu s}$. For each

random scenario generated, we sample the previous parameters from four different Erlang distributions, each one characterized by its number of stages ($k_{MTTF_{VM}}, \ldots, k_{MTTR_{\mu s}}$), and average time ($\lambda_{MTTF_{VM}}, \ldots, \lambda_{MTTR_{\mu s}}$).

A. VMs allocation

Many allocation policies can be described in our model. In particular, for each VM j we denote with $\mathbf{M}_j \subseteq \{1, \ldots, N_{\mu s}\}$ the set of microservices that are executed over it. The set $\mathbf{M}_j$ forms a partition: $\bigcup_{1 \leq N_{\mu s}} A_j = \{1, \ldots, N_{\mu s}\}$, and $A_j \cap A_k = \emptyset$, $\forall 1 \leq j,k \leq N_{VM}$. In this work we have considered two policies, addressed in the following as *case I* and *case II*. The *case I* policy defines an elementary strategy according to which containers are cyclically assigned to VMs taken from a list obtained as a random permutation of available VM number identifiers. When the application execution starts, a container is assigned to the first VM in list, then the next is assigned to the following VM in list, and so on; if the VM list is over, containers are assigned starting again from the top of the list. In this way each VM has at least one container, and services are assigned to computing resources in a random way.
The *case II* strategy instead tries to compact the containers according to their demand. The two containers with the smallest requirements are merged together on the same VM, creating a new single "equivalent" container whose demand is the sum of the ones of the services that are combined. The equivalent container replaces the two merged services, and the process is repeated until there is just one equivalent container per VM. In this way, the average utilization of the VMs is maximized, creating a more balanced system.
Other strategies that can be easily included in the model can for example be based on the actual load, overall utilization, or containers can be assigned to VMs taking into account their task and requirements.

B. Performance indexes

Given these scenario and parameters we are able to evaluate system performance by computing the indexes we report in the following. The load introduced in the cloud infrastructure by any microservice i is easily derived as:

$$D_i = v_i \cdot S_i \tag{3}$$

By counting the number of containers assigned to each VM we can compute VMs load and utilization. Hence, for any VM j we have the load defined as

$$D_j = \sum_{i \in \mathbf{M}_j} D_i \tag{4}$$

Remembering that λ_u is the rate at which users requests for the application arrives to the system, the utilization U_j for any VM j, when the system is stable, can be computed as:

$$U_j = \lambda_u \cdot D_j \tag{5}$$

In the same way, one particular configuration is stable only if Equation 5 is strictly less then one for all VMs, or equivalently if:

$$\lambda_u < \frac{1}{\max\limits_{1 \leq j \leq N_{VM}} D_j} \tag{6}$$

Throughput X_i of a container i is computed as $X_i = \lambda_u \cdot v_i$; the average response time of a VM j and the average system response time R as:

$$R_j = \frac{D_j}{1 - U_j} \qquad R = \sum_{k=1}^{N_{VM}} R_j \tag{7}$$

The availability of the application $A = A_{VM} \cdot A_{\mu s}$ is computed as the product of the availabilities of the VMs (A_{VM}) and of the microservices ($A_{\mu s}$) used during one application execution. Since each VM j and each microservice i is characterized by its own mean time to failure and mean time to repair, we can define their corresponding availability as:

$$A_{VM_j} = \frac{MTTF_{VM_j}}{MTTF_{VM_j} + MTTR_{VM_j}} \tag{8}$$

$$A_{\mu s_i} = \frac{MTTF_{\mu s_i}}{MTTF_{\mu s_i} + MTTR_{\mu s_i}} \tag{9}$$

However, since not all microservices are required during each application execution, the fault of a machine will not always cause a failure. Let us call p_{VM_j} and $p_{\mu s_i}$ respectively the probabilities that VM j or microservice i are used during a call to the application. Then effective availability $\hat{A}$ can be computed as:

$$\begin{aligned} \hat{A}_{VM_j} &= 1 \cdot (1 - p_{VM_j}) + A_{VM_j} \cdot p_{VM_j} \\ &= 1 - (1 - A_{VM_j}) \cdot p_{VM_j} \end{aligned} \tag{10}$$

$$\hat{A}_{\mu s_i} = 1 - (1 - A_{\mu s_i}) \cdot p_{\mu s_i} \tag{11}$$

Due to the geometric assumption, and since each microservice i is called an average of v_i times, we have:

$$p_{\mu s_i} = Pr\left\{ Geom\left(\frac{1}{v_i + 1}\right) > 0 \right\} = \frac{v_i}{v_i + 1} \tag{12}$$

The probability that VM j is used during a call to the application must instead consider the fact that at least one microservices i allocated over it is required, which could be expressed as:

$$p_{VM_j} = 1 - \prod_{i \in \mathbf{M}_j} \left(1 - p_{\mu s u_i}\right) \tag{13}$$

The final values of A_{VM} and $A_{\mu s}$ can then be computed as:

$$A_{VM} = \prod_{j=1}^{N_{VM}} \hat{A}_{VM_j}, \qquad A_{\mu s} = \prod_{i=1}^{N_{\mu s}} \hat{A}_{\mu s_i} \tag{14}$$

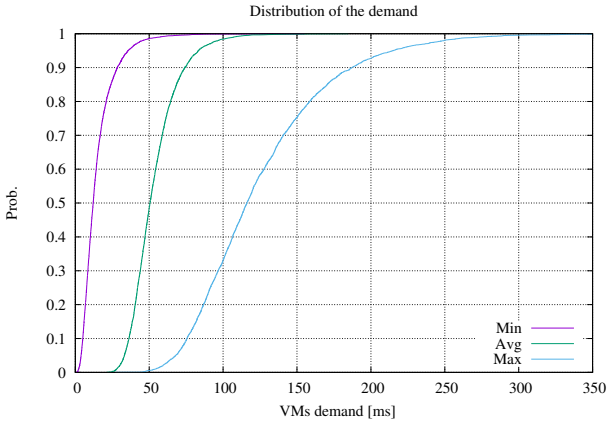

Fig. 3. Probability distribution of VMs demand

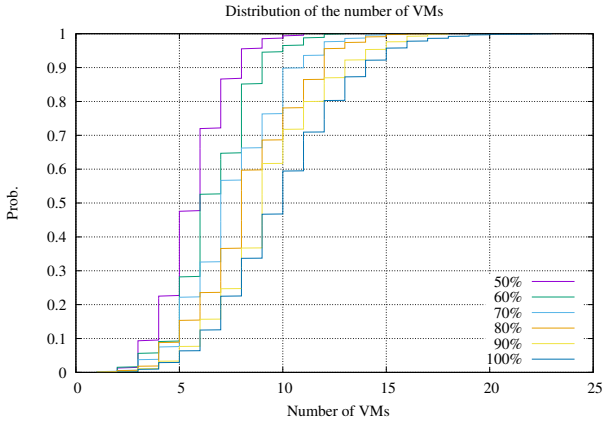

Fig. 4. Probability distribution of the number of VMs

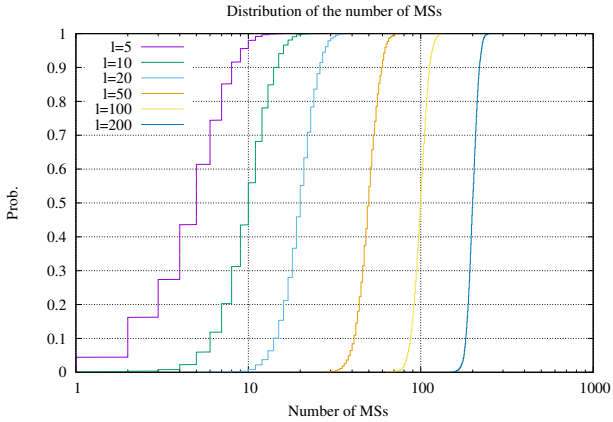

Fig. 5. Probability distribution of the number of microservices

V. EXPERIMENTS

In the following we report the results obtained from a set of experiments. For each configuration, we have generated between 5000 to 100000 random applications (depending on the fraction of stable cases), and we have collected both average values and distributions. We show the outcome in three different classes: infrastructure, performance, and availability. In all the following case studies we have set the parameters corresponding to the popularity and the average service time as reported in Table I.

TABLE I
COMMON PARAMETERS FOR ALL THE EXPERIMENTS

c	6	q	2
α	1	s	1.5
λ_S	100 ms.	k_s	4
$\lambda_{MTTF_{VM}}$	1000 h.	$\lambda_{MTTR_{VM}}$	2 h.
$\lambda_{MTTF_{\mu s}}$	500 h.	$\lambda_{MTTR_{\mu s}}$	6 min.

A. Infrastructure

The figures of this section plot indexes related to the system structure: their purpose is to show the main features that the applications generated with the considered set of parameter distributions have. Fig. 3 shows minimum, average, and maximum of the demand load on each VMs. The parameters setting is $\lambda_u = 1$ call / sec., $\lambda_{\mu s} = 10$ ms., $\beta = 66\%$, the VM selection policy is Case I. As we can see, the popularity mechanism creates a few microservices that are heavily loaded: for this reason the average demand is more skewed towards the minimum.

The probability distribution of the number of VMs for different values of β is reported in Fig. 4: the savings that could be made in number of VMs with a lower value of β become important only when the application is composed by a large number of microservices.

The probability distribution of the number of microservices is showed in Fig. 5 as function of the $\lambda_{\mu s}$: the Poisson distribution provides a good way to generate meaningful topologies when only a rough idea on the average number of microservices is available.

B. Performance

In this section performance indexes are reported. We first analyze the system response time versus the user demand. Fig. 6 shows the probability distribution of the response time with increasing values of λ_u, $\lambda_{\mu s} = 10$, and $\beta = 66\%$, the VM selection policy is Case I. As expected the system reacts more slowly when microservices execution demand is higher. Moreover, the probability distributions are defective, since as the load increases, there is a higher chance of obtaining unstable topologies that are excluded from the output.

In Fig. 7 mean response time and average percentage of stable runs are compared in case I and II versus the user load λ_u, with $\lambda_{\mu s} = 10$ and $\beta = 66\%$. Since mean values are considered, confidence intervals are reported. Case II outperforms the response time of Case I, with the exception of the case with the highest value of λ_u. The reason lies in the fact that with this value the model has a relevant number of unstable runs.

After the evaluation versus the user load, we studied the indexes as a function of the number of VMs. Fig. 8 reports

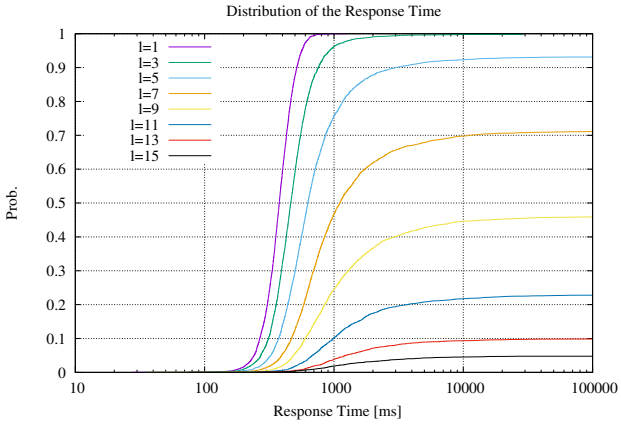

Fig. 6. Probability distribution of the response time

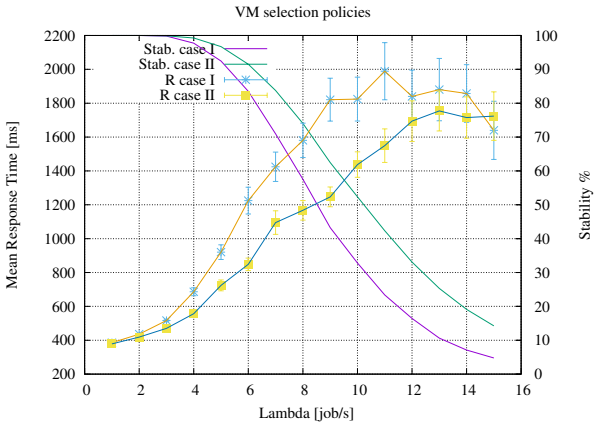

Fig. 7. Comparison of VMs policies

the probability distribution of VMs utilization with different values of β.Parameters are $\lambda_u = 12$, $\lambda_{\mu s} = 10$, the VM selection policy is Case I. When the percentage of VMs is lower, each VM get more microservices to be executed and hence the overall VMs utilization is higher.

Fig. 9 shows the distribution of the minimum, average, and

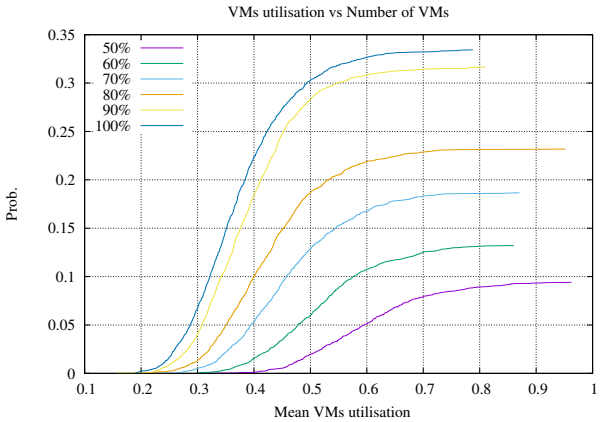

Fig. 8. Probability distribution of VMs utilization

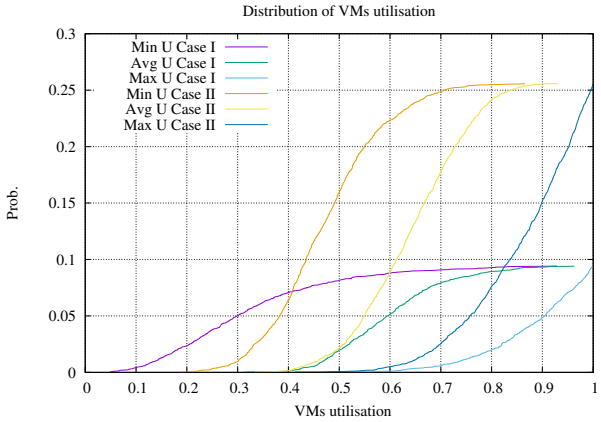

Fig. 9. VMs utilization with different policies

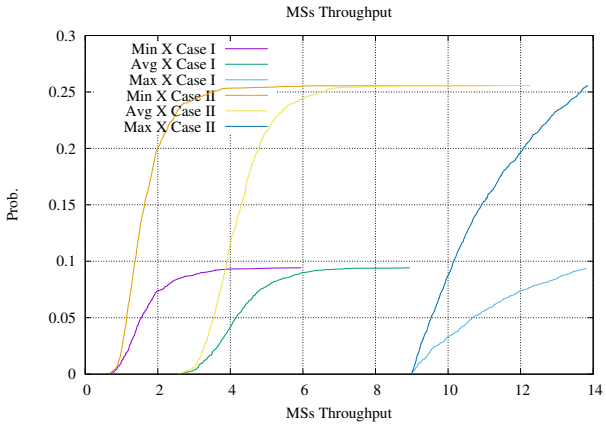

Fig. 10. MSs throughput

maximum VMs utilization in each topology, with different policies. Parameters are $\lambda_u = 12$, $\lambda_{\mu s} = 10$, and $\beta = 50\%$. The utilization in Case I is lower, but as it is showed in Fig. 7 this policy is less unstable, indeed the VM selection is random and it can happen that some VM get higher load then the others.

The distribution of the minimum, average and maximum throughput of the microservices in each simulation run is reported in Fig. 10. As it can be seen, although both cases has the same throughput (since this parameter is determined by the application and not by its deployment on the infrastructure) Case II policy performs better than the Case I, having a larger number of configuration in which the system is stable.

C. Availability

Finally, we evaluated the system from the availability point of view. Fig. 11 shows the overall, microservices, and VMs unavailability, with $\lambda_u = 1$, $\lambda_{\mu s} = 10$, and $\beta = 66\%$. The Erlang distribution used to generate the MTTF and MTTR are all characterized by 10 stages. The other parameters are reported in Table I. As it can be seen, in this scenario most of the faults are caused by software error in microservices rather than problems with the VMs.

The probability distribution of the availability versus VMs and microservices MTTFs are reported in Fig. 12 and Fig.

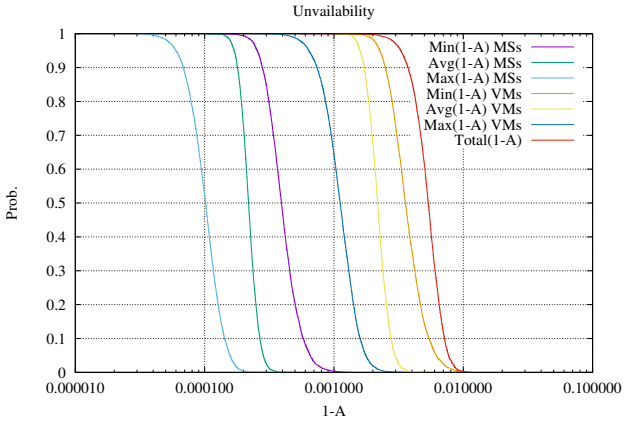

Fig. 11. Total, microservices, and VMs unavailability

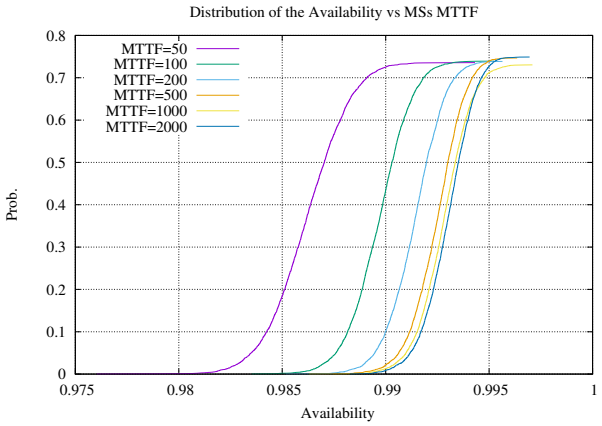

Fig. 12. Probability distribution of the availability versus microservice MTTF

13 respectively, with $\lambda_u = 8$, $\lambda_{\mu s} = 20$, and $\beta = 66\%$, Case II policies is used. As expected, the higher the MTTF, the higher the probability to have better availability.

VI. CONCLUSIONS

In this paper we proposed an approach for performance evaluation of infrastructures that support the execution of a mix of microservice-based software applications. Our approach, to the best of our knowledge, is the first parametric simulation approach that allows providers to model such architectures in a general case of an aggregated heterogeneous tunable workload mix, to support decisions in the design, maintenance and management of microservice-based infrastructures. Future works include further parameterization of the simulation approach, the exploration of massive real workloads for a better modeling approach, the extension of the simulation support for more infrastructural configurations, and a more accurate validation campaign when sufficient data will be available about traces from real, production infrastructures. Finally, the simulator will be extended in order to include energy issues.

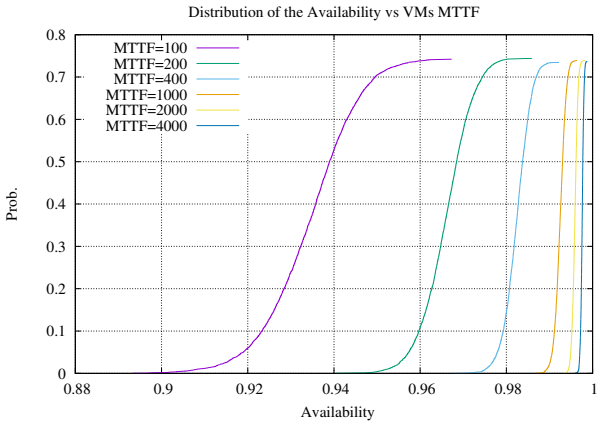

Fig. 13. Probability distribution of the availability versus VM MTTF

REFERENCES

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

[2] C. Esposito, A. Castiglione, and K. K. R. Choo, "Challenges in delivering software in the cloud as microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 10–14, Sept 2016.

[3] N. Alshuqayran, 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)*, Nov 2016, pp. 44–51.

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

[5] M. Villamizar, 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)*, Sept 2015, pp. 583–590.

[6] M. Villamizar, 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)*, May 2016, pp. 179–182.

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

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

[9] V. Heorhiadi, 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)*, June 2016, pp. 57–66.

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

[11] A. Melis, 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)*, Sept 2016, pp. 1–2.

[12] B. Butzin, 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)*, Sept 2016, pp. 1–6.

[13] A. Castiglione, M. Gribaudo, M. Iacono, and F. Palmieri, "Modeling performances of concurrent big data applications," *Software: Practice and Experience*, vol. 45, no. 8, pp. 1127–1144, 2015.

[14] M. Gribaudo, M. Iacono, and D. Manini, "Three layers network influence on cloud data center performances," 2016, pp. 621–627.

[15] D. Cerotti, M. Gribaudo, M. Iacono, and P. Piazzolla, "Modeling and analysis of performances for concurrent multithread applications on multicore and graphics processing unit systems," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 2, pp. 438–452, 2016, cpe.3504.