Model Based Control for Multi-cloud Applications

Marco Miglierina, Giovanni P. Gibilisco, Danilo Ardagna, Elisabetta Di Nitto
Politecnico di Milano, Italy
(miglierina|gibilisco|ardagna|dinitto)@elet.polimi.it

***Abstract*—The advent of cloud computing has offered to developers a new appealing paradigm to deploy their applications without capital investments. Resources can now be acquired on-demand in a flexible, scalable and rapid way. However, cloud providers lack of native mechanisms to guarantee the Quality of Service required by specific application domains. High availability can be achieved by replication of critical components. Since outages could affect the entire cloud provider, replication can be effective only by using multiple providers.**

In this paper we tackle the above problem and present an approach to guarantee availability requirements of cloud-based applications by exploiting replication on multiple clouds to reduce unavailability, still limiting costs. More precisely, we propose: *i)* an approach to model, at design time, the application, its availability requirements and the characteristics of the used clouds, and *ii)* a self-adaptive technique responsible, at runtime, of both in-cloud scaling policies and traffic routing among different cloud providers, by means of a control-theoretical approach.

We integrated the modeling approach in the Palladio Bench IDE and developed a runtime self-adaptation controller in Matlab. The controller has been evaluated against different workload conditions, costs variations and service failures in simulated scenarios. The controller has been able to provide the desired availability minimizing costs.

***Index Terms*—Cloud computing, multi-cloud applications, availability, control theory, non-functional requirements, self-adaptive software.**

I. Introduction

The advent of cloud computing is promoting a new way of building applications and services and offering them to the public. The possibility of acquiring computational, storage and communication resources only when they are needed results in a business model that emphasizes flexibility, scalability, and performance. The fact that cloud providers let users pay only for the resources they use makes the cloud a very interesting option also from the economic point of view.

However, cloud solutions lack native mechanisms to guarantee the Quality of Service (QoS) required by specific application domains. In such domains, high availability requirements are usually satisfied by replication of critical components. While the acquisition of a large number of resources to support replication is certainly not a technical problem in the cloud, this may affect costs significantly since replicated components are usually left unused and exploited only in case of failure of the primary resource. The fact that cloud resources can be acquired in a matter of minutes could help to solve the over provisioning problem due to replication, but it is not sufficient to fully solve the availability problem. Many cloud providers offer in their Service Level Agreements (SLAs) an availability value of 99.95% for users applications. This value is not enough for business critical applications. Indeed, [1] shows that availability values experienced by users of cloud based services are much lower than those declared by cloud providers. The study reports that, in the analyzed period, the European region of Amazon EC2 showed an average availability of 96.32% and the Windows Azure service showed an even lower availability equal to 95.39%.

A possible solution to this problem could be to exploit different cloud providers and migrate the application from one cloud to another when the availability requirements are not fulfilled by the first one. An application that uses multiple cloud providers in such a way is called *multi-cloud application*.

In this paper we present a technique to achieve high availability by exploiting multiple clouds, while minimizing infrastructural costs. More precisely, we propose: *i)* a design time approach to model the application, its availability requirements, and the characteristics of the target cloud infrastructures; and *ii)* a runtime self-adaptive technique responsible for both in-cloud scaling policies and traffic routing among different cloud providers, by means of a control-theoretical approach.

We have developed the modeling approach as an extension to the integrated modeling environment Palladio Bench, and the runtime self-adaptation controller as a Matlab system. We have then evaluated our proposal by analyzing the runtime behavior of the controller against different workload conditions, costs variations, and service failures. Results show that our controller is capable of guaranteeing the desired availability, minimizing the total cost of cloud resources provisioning. Our work fits in the view depicted in [2].

The rest of the paper is structured as follows: Section II presents some related work that deals with control of component based applications. Section III gives an high level overview of our approach. Section IV describes the model developed for the description of multi-cloud applications and used by the controller of Section V. Section VI presents the preliminary evaluation of our approach. Section VII summarizes the results and points to some directions for future work.

II. Related Work

Several approaches to the general problem of modeling, analyzing and enforcing QoS requirements of software systems have been presented in the literature. [3] provides an overview of these approaches. Most of them target the design phase. Some, like the one presented in [4], focus on the modeling of QoS requirements and in supporting developers in the

creation of documentation. Others (see [5]) help developers by optimizing in an automatic or semi automatic way the architecture of an application.

A control-based approach similar to the one presented in this paper can be found in [6]. This is one of the first works where a control theoretical approach was used to solve problems of self-adaptation in software system models. Authors derive from a Discrete Time Markov Chain (DTMC) that describes an application, a closed formula that defines the explicit dependency of reliability on control variables and measured reliabilities. The controller uses this formula to compute the actual availability of the system with regard to changes in the measured reliabilities. The controller then minimizes a cost function defined over the control variables using as a constraint the fact that the closed formula, updated with estimated parameters, must be equal to the desired availability.

An approach to cloud service provisioning is presented in [7] where authors build an integer programming problem from deadline and budget constraints and solve it to get scaling decisions. In [8] authors propose a cost-aware cloud provisioning engine that exploits replication and migration to reconfigure applications in order to minimize costs and guarantee throughput. Authors of [9] address the challenges of minimizing the total amount of resources while meeting performance requirements for applications.

Differently from the other approaches, our work starts from the assumption that the target application is capable of migrating between different cloud providers. This can be seen as a restrictive assumption at present but, as stated in [10], the research in the area of cloud migration is very active. There are many works in this direction, especially within European projects, like Reservoir, mOSAIC, cloud4SOA, REMICS, and ALERT projects [11], [12]. Another interesting work dealing with cloud interoperability is presented in [13]. All this work in the field shows that our assumption is likely to be fulfilled in the near future.

III. Overview of the Approach

Our approach is based on control theory, a branch of engineering that deals with dynamic systems with inputs. In particular, we adopt a *closed-loop controller* approach that uses the model of the system and feedback from the measured output. As shown in Figure 1, we use an extended Discrete Time Markov Chain (DTMC), defined during design time, as our system model and data from a monitoring system as feedback. Using such information, our controller is able to make decisions on the actions to execute in order to fulfill the availability requirements defined at design time. Such decisions are made at runtime and have an impact on the actual configuration of the system. The controller keeps alive the model by updating its parameters with estimates based on the monitoring data.

DTMCs [14] are a common formalism to describe systems from the availability viewpoint. They are graphs where nodes represent *states* and edges model *transitions*, i.e., state changes, with a probability attached to them. A state describes

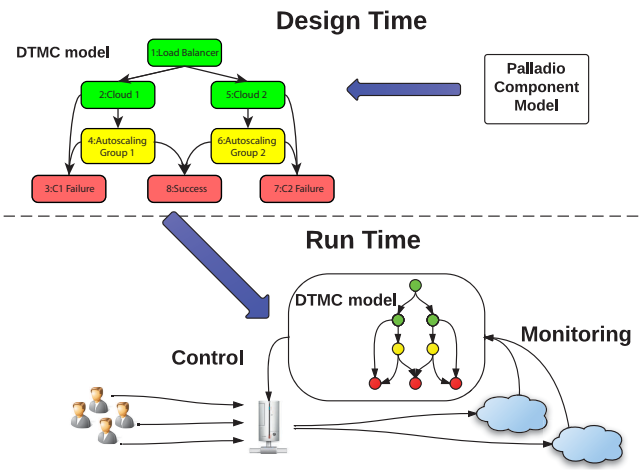

Fig. 1. Overview of the approach

some information about a system at a certain moment of its behavior. Transitions specify how the system can evolve from one state to another. The successor of state, say, s is chosen according to a probability distribution that only depends on the current state s, and not on, e.g., the path fragment that led to state s from some initial state. States (or transitions) can be augmented with rewards, numbers that can be interpreted as bonuses, or, dually, as costs. The idea is that whenever a state s (or transition) is chosen, the reward associated to s is earned.

Our extended DTMC is derived from more abstract design artifacts in which all characteristics of a multi-cloud application are described. Such artifacts are built using our extension of Palladio Bench [15]. This is an IDE based on Eclipse that provides different tools for allowing developers to build separate diagrams describing some characteristics of the system to be. The tool then automatically integrates all these diagrams and generates models of the entire system to analyze some QoS properties at design time. Our Palladio extension allows the automatic generation of a DTMC of the application from the diagrams built in Palladio at design time.

Due to space limitations, in this paper we do not focus on the way the extended Palladio generates the DTMC nor on the monitoring and actuation aspects, while we describe our model and controller.

IV. Design-Time Modeling

This section introduces the DTMC model that has been developed to model the availability of an application deployed on multiple clouds. Compared to existing approaches in the literature, our DTMC allows to model explicitly some cloud-specific concepts that are relevant to our analysis.

More precisely, in the model nodes represent *Physical* and *Logical* Nodes (see Figure 2). Physical Nodes correspond to some concrete resource, e.g, a physical server or a pool of virtual machines offered by a cloud provider. Logical Nodes represent entites that are relevant for the DTMC analysis but do not have a specific correspondence to concrete elements in the cloud. Examples of Logical Nodes can be the ones

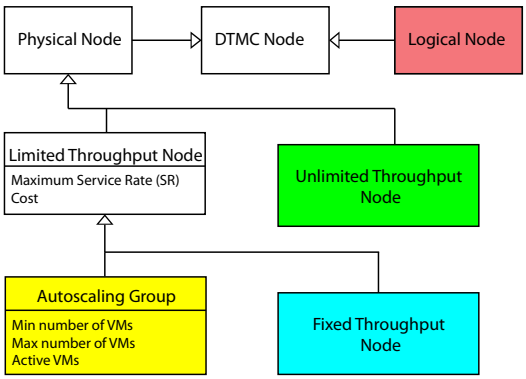

Fig. 2. Types of nodes in the DTMC

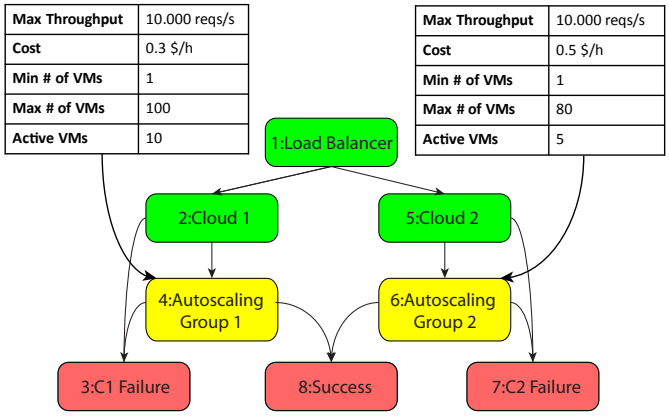

Fig. 3. Case study extended model

representing the success or the failure state of the application. In turn, Physical Nodes can either represent elements in the concrete system to which we want to associate a limit in the throughput and a cost, or elements that, from the perspective of the model, have unlimited throughput and no cost associated. In the figure these are presented by *Limited Throughput Nodes (LTN)* and *Unlimited Throughput Nodes (UTN)*. In our examples we have modeled load balancers as UTNs because usually they are managed by cloud providers in an automatic fashion and no control is possible on them.

Of course, LTNs represent the most important part of the model. They can be further classified in *Autoscaling Groups* or *Fixed Throughput Nodes*. The first ones represent entities capable to perform autoscaling, i.e., to change dynamically their processing capacity. In this case the *Maximum Service Rate (SR)* is given by *Number of running machines* $\times$ *single machine maximum service rate (sr)*. *Fixed Throughput Nodes* are very useful if the designer wants to model an hybrid cloud architecture where some nodes have a fixed capacity.

As in [6], in our DTMC we add *control variables* and *measured availabilities* as labels to transitions. Measured availabilities represent factors originating from the infrastructure and are external to the application. In control theory terminology these factors are called disturbances and can be measured by monitors. Examples of these factors are blackouts or outages of application components due to a failure of the middleware managing data centers (which may lead to world wide outages), or failures of a local computing resource. Control variables represent alternative choices, made according to certain probabilities. These probabilities define the rate at which requests are routed through connected nodes.

In a DTMC model, *rewards* or *costs* can be introduced. In our model rewards are attached to states and model the cost generated by a request traversing that node. Recalling the distinction of nodes just presented, one can note that only computing resources represent nodes with a positive cost while logical nodes have zero cost. This is due to the fact that they are not mapped, as a first approximation, to any physical resource consumption that leads to an increase in the cost of the system.

We require that the developer annotates the nominal cost of using the resource modeled by the node. Instance pricing is usually constant and available on the provider web site. Though, we took into consideration the fact that prices could change, like Amazon spot instances[1].

The next two parameters that will be presented are used only in autoscaling nodes since they model specific features of the cloud environment. Each autoscaling node is labeled with a *minimum* and a *maximum number of running instances*; these two parameters represent, respectively, the minimum and maximum number of machines that can run simultaneously on the resource modeled by the node. These parameters can be used, for example, to model the fact that the designer may decide to fix the number of machines running on each region of a cloud provider (e.g., at least two machines always running). The last parameter that the designer has to specify in order to build a complete instance of the model represents the availability constraint of the application which can be a fixed value or a function of time.

The model of the case study described in Section III has been automatically derived from Palladio Bench by our tool and is reported in Figure 3.

Red circles represent logical nodes. The green circle represents a load balancer as a physical node that has zero cost and an infinite processing capacity. Yellow circles represent Autoscaling groups. The application presented in this model is composed of two replicas of a service deployed on top of two cloud providers represented by states 4 and 6. Requests entering the system are directed to one of the two services by the load balancer according to probabilities $C0$ and $1 - C0$. When reaching one of the two cloud providers, represented in states 2 and 5, requests can be lost because of a failure in the cloud infrastructure and go directly to failure states 3 and 7. If a request reaches one of the services, it is processed. The processing of a request could lead to a successful execution of the service, state 8, or to a failure due to the overloading

[1]http://aws.amazon.com/ec2/spot-instances/

of the processing resource, represented by transitions to states 3 and 7. The table attached to node 4 represents the values of the parameters that characterize that specific node. For the sake of simplicity only labels of the relevant nodes have been represented in the image.

V. Run-Time Control

The proposed DTMC model is meant to be kept alive at run-time, so that whenever some controller modifies it, changes take effect on the actual implementation.

The controller we define in this paper is a dual layer controller. The first layer controller is responsible for managing one autoscaling group, controlling the number of running machines. Therefore, there are as many first layer controllers as the number of nodes modeling autoscaling groups. The second layer controller is a load balancer in charge of distributing the incoming traffic among nodes. The cooperation between these two layers of controllers aims at guaranteeing system availability, while minimizing costs.

Both controllers work at discrete time, that is, monitoring data is aggregated and delivered from monitors at constant time intervals (steps).

A. The Autoscaling Controller

The objective of this controller is to manage the number of running machines in an autoscaling group so that the average percentage of CPU utilization is equal to a *setpoint*.

What we need for a controller is a feedback loop. So we need data from "sensors", i.e., we assume that we can check how the system is behaving in response to controller's decisions. First of all, we define a *sliding observation window*, which is the time span (or number of steps) used to calculate statistics from data collected by sensors (i.e., the IaaS monitoring system). The statistics we gathered are the following:

- the *incoming workload*, that is the number of incoming requests to the node;
- the *successful requests*, that is the number of requests successfully processed by the node;
- the *average CPU load*, that is the average percentage of CPU utilization computed over all running machines in the node;
- the *number of running machines*.

From this data, the success rate is then estimated as $\frac{\textit{successful requests}}{\textit{incoming workload}}$. The success rate will be our parameter of availability.

Given estimated data from the monitoring system, we first need to find a control formula where the error observed between the desired behavior and the actual one, can be reduced (and asymptotically eliminated) at each control step acting on the control variables.

We can identify two main working conditions:

1) the number of machines is sufficient to satisfy the entire incoming traffic (average CPU usage $\leq$ 100%, availability = 100%).
2) the number of machines is not sufficient to satisfy the entire incoming traffic (average CPU usage = 100%, availability $<$ 100%);

In order to control the system we make use of a feedback loop methodology used in [6]. The only control variable we have is the number of machines we want to have running at the next step, i.e., $n(k+1)$.

As for the first working condition, all incoming traffic is satisfied, therefore availability is 100%, and we want to reach the CPU setpoint u. Therefore, the controller should find $n(k+1)$ so to satisfy the following equation

$$u(k+1) - \hat{p}(k+1|k) = \alpha(u(k) - p(k)) \tag{1}$$

where $\hat{p}(k+1|k)$ is a function of $n(k+1)$ and represents the expected CPU usage at the next step. α is a parameter in the range $(0,1)$ and determines how fast is the convergence to the solution, that is, in the next step we expect the absolute error to be reduced by a factor α.

The relation between $\hat{p}(k+1|k)$ and $n(k+1)$ can easily be found given the equation

$$p(k) = \frac{AR(k)}{SR(k)} \tag{2}$$

from [16], where SR and AR are the total maximum service rate and the total arrival rate respectively. We point out that Equation 2 holds only in the first working condition. Furthermore, we recall that SR is given by the contribution of all virtual machines having each a maximum service rate sr:

$$SR(k) = sr(k)n(k) \tag{3}$$

From Equations 2 and 3 we get:

$$p(k+1) = \frac{AR(k+1)}{sr(k+1)n(k+1)} \tag{4}$$

We assume that time intervals are small enough to consider the maximum service rate of a machine and the arrival rate to remain constant. If this does not hold, prediction can be taken in consideration, but it is out of the scope of this paper. Therefore our expected CPU utilization is:

$$\hat{p}(k+1|k) = \frac{AR(k)}{sr(k)n(k+1)} \tag{5}$$

that after some algebra becomes:

$$\hat{p}(k+1|k) = p(k)\frac{n(k)}{n(k+1)} \tag{6}$$

Using this result with Equation 1 we analytically obtain the control formula to be used in the first working condition:

$$n(k+1) = \frac{n(k)p(k)}{u(k+1) - \alpha(u(k) - p(k))} \tag{7}$$

As for the second working condition, the incoming traffic is higher than the total service rate, therefore the total CPU usage is equal to 100% and we are not able to use control formula 7 anymore. However, in this working condition it is easy to evaluate the maximum service rate SR since it is equal

to the total throughput, i.e., the traffic actually satisfied over time. Using Equations 3 and 4, setting $p(k+1)$ equal to our setpoint $u(k+1)$ and given assumptions similar to the ones made before, after some algebra we obtain the exact number of machines $\bar{n}$ required to satisfy the incoming traffic:

$$\bar{n} = \frac{AR(k)n(k)}{SR(k)u(k+1)} \tag{8}$$

In order to deal with noise, we use in this case as well a convergence rate to the desired setpoint:

$$n(k+1) - \bar{n} = \alpha(n(k) - \bar{n}) \tag{9}$$

which gives the final control formula:

$$n(k+1) = \alpha n(k) + (1-\alpha)\bar{n} \tag{10}$$

We know from [6] that exponential convergence to the setpoint is ensured for equations of the kind of 1 and 9 with rate α.

B. The Load Balancer Controller

The second layer controller is responsible for setting the controllable variables of the DTMC model, so to distribute the traffic. The approach used for this layer is a cloud extension of the work in [6].

This controller aims at distributing traffic among nodes guaranteeing availability and minimizing costs. The model is iteratively updated at run-time using monitor data. Model's parameters that require to be continuously estimated are the maximum service rate SR_i for each node i, and the *cost per request* K_i, which measures the "virtual" cost of a request traversing node i. In fact, costs are usually associated to the number of running virtual machines: cost per instance hour. However, when the autoscaling controller is stable, the number of machines is directly related to the number of incoming requests. The number of incoming requests when all machines of node i are working at the desired CPU usage level u_i, is equal to $SR_i \times u_i$. It follows $K_i = (c_i \times n_i)/(SR_i \times u_i)$ where c_i is the cost per machine and n_i is the number of running machines. The cost of a request is therefore "virtual", because the cost of a virtual machine is indeed spread over the processed requests.

The *set point* at this layer is the minimum *success rate* of the system. We decided to allow the developer to set a minimum because even though he would obviously always like to have 100%, for some applications he might want to make a trade-off between costs and availability.

In this case, the problem cannot be solved analytically anymore. The load balancer controller is in charge of solving a non-linear constraint minimization problem.

Since we deal with probabilities, the first constraint is that controllable variables must be chosen in the range $(0, 1)$. Also, since we are dealing with a DTMC, the sum of the outgoing arcs must be 1. This last constrain can be avoided by allowing only two outgoing arcs on load balancers and setting the value of one of the arcs equal to one minus the other. If we want to have a load balancer with three or more outgoing arcs, it is enough to put two or more binary load balancers in cascade.

Then we need a constraint on the success rate, which has to be greater or equal to the set point. To do this, we must obtain a formula that describes the explicit dependency of system availability on control variables and measured nodes availabilities. First of all, given the transition matrix $\mathbf{A}$ of our DTMC model with self loops removed (i.e., no ones on the diagonal), i is the row of the matrix relative to the input node, j is the row of the matrix relative to the output node (i.e., the success state), we can write the following dynamic system

$$\mathbf{x^T}(k+1) = \mathbf{x^T}(k)\mathbf{A} + \mathbf{b^T}$$

where $\mathbf{x}$ is a vector as long as the number of nodes, and b is the input vector, as long as $\mathbf{x}$, with all 0s except for the ith element which is 1. If $\mathbf{b}$ is constant the system is going to stabilize and the values of x are going to be the *workload ratio* arriving at each node:

$$\mathbf{x^T} = \mathbf{x^T}\mathbf{A} + \mathbf{b^T} \;\Rightarrow\; \mathbf{x^T}(I - \mathbf{A}) = \mathbf{b^T} \;\Rightarrow$$
$$\mathbf{x^T} = \mathbf{b^T}(I - \mathbf{A})^{-1}$$

The jth element of $\mathbf{x}$ is going to be the success rate as a function of the control variables and nodes availabilities, which will be used to estimate the availability. Since we are dealing with models whose structure is constant in time, the success rate function is always the same and can be computed at design time.

Now we can write the availability constraint function as

$$u(k+1) - \hat{s}(k+1|k) \leq \beta \cdot \max(0, u(k) - s(k)) \tag{11}$$

where u is the set point, $\hat{s}$ is the estimated availability, using the average availabilities of the nodes and letting $\hat{s}$ become a function only of the control variables. γ is a parameter in the range $(0, 1)$ that will affect the convergence rate to the solution. Finally, s is the system availability measured at step k. Using Equation 11 the controller is allowed to let $\hat{s}$ be greater than the set point u.

Now we define the *cost function* that has to be minimized. The first objective, as we said, is to minimize costs. However, we also need to discourage the controller from overloading a node with more requests than the ones it is actually estimated to be capable of processing. Whenever a migration of requests for economic reasons is required, the workload has to be gently distributed on the cheaper node letting it the time to scale up without overloading it, that is, without loosing requests and affect availability. The *cost function* we defined is the following

$$J = \mathbf{x^T} \cdot \mathbf{K} + W \left\| \max(0, AR(k)\mathbf{x} - \mathbf{SR}(k)) \right\| \tag{12}$$

where $\mathbf{x}$ is the previously calculated *workload ratio* array that, once availabilities are substituted with the average availabilities measured for each node, depends only on the control variables. $\mathbf{K}$ is the vector containing the *cost per request* values. W is a large number used to discourage that a solution is found where the component $\|\max(0, AR(k)\mathbf{x} - \mathbf{SR}(k))\|$ is greater than zero, that happens when one or more nodes are going to be overloaded.

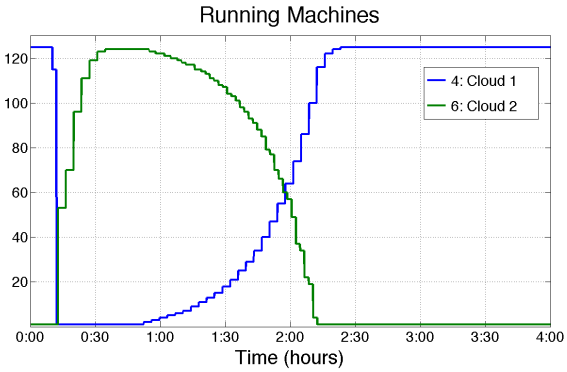

Fig. 4. Number of running machines over time

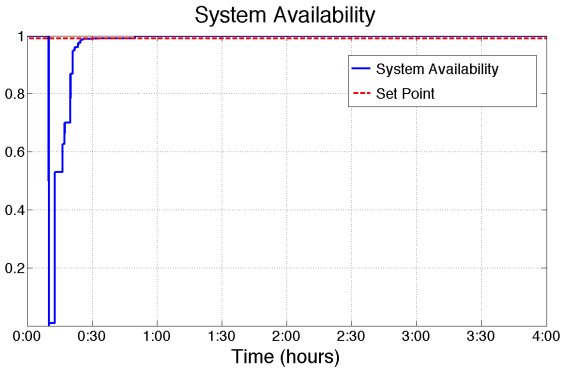

Fig. 5. System availability over time

When the availability constraint is not satisfied, because, for example, an entire autoscaling group failed, the controller will not find a minimal solution of J without overloading a node, but in this case it turns out to be the desired behavior for the following reasons:

- All requests going to the failed node would be lost anyway
- The overloaded node will scale much faster in order to cope with the new workload since the arrival rate AR in the Equation 8 will be very high.

The *autoscaling controller* and the *load balancer controller* work simultaneously. In order to prevent oscillations due to controllers coordination, it was sufficient to set a policy that inhibits the *load balancer controller* whenever any cloud is not able to satisfy the incoming traffic, giving time to the *autoscaling controller* to make it scale up.

VI. Experimental Results

We evaluated our approach by means of the two clouds scenario described in Section IV whose model is represented in Figure 3. Evaluation on a real cloud infrastructure is part of our future work. As for an initial analysis of the approach we developed a tool able to simulate generic IaaS (Infrastrucutre as a Service) cloud infrastructures by means of Matlab. The tool allows the user to test the behavior of a multi-cloud application as we described in Section IV, against different workload conditions, costs variations, and service failures.

We simulated 4 hours of activity with simulation steps of 1 second and sliding windows of 60 seconds. We kept arrival rate, service rate and costs constant. Cloud 1 was set to be the cheapest choice, by running all machines with the same maximum service rate and the same CPU setpoint (80%) and different cost per hour. The autoscaling controller had α set to $1/3$. The load balancer controller had β set to $1/3$. Machines startup time was set to 100 seconds, an average value for Linux machines on Amazon cloud according to [17]. The desired system availability is kept constant to 99%. We induced a single failure event by simulating an outage of Cloud 1 between 00:10 and 00:50 and observed the results illustrated in Figures 4 and 5.

The entire traffic was initially delivered only to Cloud 1. At 00:10 Cloud 1 availability changes abruptly from 100% to 0%. Traffic is instantly migrated to Cloud 2 and the autoscaling controller manages the scale up activity in 20 minutes. This time is dependent on the choice of parameters α and β, the smaller the faster but more sensitive to noise, and on machines startup times. At time 00:50 Cloud 1 recovers and, since it remains the cheapest solution, requests are migrated back from Cloud 2 to Cloud 1. The difference between the speed of migrations has to be noticed. While in the first part the migration to Cloud 2 is forced by the outage of Cloud 1, generating a large number of lost requests (see availability drop at time 00:10 in Figure 5), and has to be actuated by the controller very fast, in the second part the migration to Cloud 1 is graduated by the controller so that no request is lost in the migration. This behavior is guaranteed by the component $W \left\| \max(0, AR(k)\mathbf{x} - \mathbf{SR}(k)) \right\|$ in Equation 12. When Cloud 1 fails, the controller cannot find a solution without having this component different from zero, therefore overloading occurs. In the second part of the simulation, all solutions are found where the component is equal to zero, so overloading is prevented, and the autoscaling controller is given the time to make Cloud 1 scale up.

VII. Conclusion and Future Work

This paper presented a novel approach for modeling availability in multi-cloud applications and controlling their deployment configuration in order to guarantee availability constraints. Results show that our controller is capable of coping with cloud failures and re-establishing the desired availability in a short time. Results also show that the controller is capable of reducing operative costs by scaling the number of provisioned resources and balancing incoming traffic among the available cloud providers. In particular even during the migration of service between cloud providers for economical reasons the availability value of the entire system is not affected.

Future research will first go through different improvements on the resolution of constraint optimization problems so to cope with more challenging scenarios where the adopted technique had some issues in finding the optimal solution.

Another improvement to be investigated is the estimate of future parameters. In our approach the average of monitoring data in the observation window is used to estimate parameters while more advanced techniques, like the Kalman Filter, could lead to more precise estimation of the underlying system state. Finally, we plan to test our approach in a real cloud scenario and considering industrial use cases.

ACKNOWLEDGEMENTS

This research has been partially supported by the European Commission, Grant no. FP7-ICT-2011-8-318484 MODAClouds project.

The authors would like to thank Prof. Carlo Ghezzi, Prof. Alberto Leva and Dr. Antonio Filieri for valuable discussions.

REFERENCES

[1] Bitcurrent, "Cloud performance from the end user," http://www.bitcurrent.com/, Tech. Rep., 2011.

[2] D. Ardagna, E. Di Nitto, P. Mohagheghi, S. Mosser, C. Ballagny, F. D'Andria, G. Casale, P. Matthews, C. S. Nechifor, D. Petcu, A. Gericke, and C. Sheridan, "MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds," in *Modeling in Software Engineering (MISE), 2012 ICSE Workshop on*. IEEE, Jun. 2012, pp. 50–56. [Online]. Available: http://www.i3s.unice.fr/~mosser/_media/research/mise_icse_2012.pdf

[3] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: a survey," in *Software Engineering, IEEE Transactions on*, vol. 30, no. 5, may 2004, pp. 295 – 310.

[4] C. Vicente-Chicote, B. n. Moros, and A. Toval, "REMM-Studio: an Integrated Model-Driven Environment for Requirements Specification, Validation and Formatting," in *Journal of Object Technology, Special Issue TOOLS EUROPE 2007*, vol. 6, no. 9, Oct. 2007, pp. 437–454. [Online]. Available: http://www.jot.fm/issues/issue_2007_10/paper22/

[5] M. L. Drago, "Quality driven model transformations for feedback provisioning," Ph.D. dissertation, Politecnico di Milano, 2012.

[6] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio, "Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, nov. 2011, pp. 283 –292.

[7] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, oct. 2010, pp. 41 –48.

[8] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "Kingfisher: Cost-aware elasticity in the cloud," in *INFOCOM, 2011 Proceedings IEEE*, april 2011, pp. 206 –210.

[9] P. Xiong, Z. Wang, S. Malkowski, Q. Wang, D. Jayasinghe, and C. Pu, "Economical and robust provisioning of n-tier cloud workloads: A multi-level control approach," in *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ser. ICDCS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 571–580. [Online]. Available: http://dx.doi.org/10.1109/ICDCS.2011.88

[10] D. Petcu, "Portability and interoperability between clouds: challenges and case study," in *Proceedings of the 4th European conference on Towards a service-based internet*, ser. ServiceWave'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 62–74. [Online]. Available: http://dl.acm.org/citation.cfm?id=2050869.2050876

[11] B. Di Martino, D. Petcu, R. Cossu, P. Goncalves, T. Máhr, and M. Loichate, "Building a mosaic of clouds," in *Proceedings of the 2010 conference on Parallel processing*, ser. Euro-Par 2010. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 571–578. [Online]. Available: http://dl.acm.org/citation.cfm?id=2031978.2032056

[12] N. Loutas, V. Peristeras, T. Bouras, E. Kamateri, D. Zeginis, and K. Tarabanis, "Towards a reference architecture for semantically interoperable clouds," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, 30 2010-dec. 3 2010, pp. 143 –150.

[13] R. Teckelmann, C. Reich, and A. Sulistio, "Mapping of cloud standards to the taxonomy of interoperability in iaas," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 522–526.

[14] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[15] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," in *J. Syst. Softw.*, vol. 82, no. 1. New York, NY, USA: Elsevier Science Inc., Jan. 2009, pp. 3–22. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2008.03.066

[16] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984.

[17] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, june 2012, pp. 423 –430.