Adaptive Resource Management for Balancing Availability and Performance in Cloud Computing

Ravi Jhawar and Vincenzo Piuri
Dipartimento di Informatica – Università degli Studi di Milano, 26013, Crema, Italy
Email: {firstname.lastname}@unimi.it

Keywords: Availability, Cloud computing, Dynamic adaption, Fault tolerance management, Performance, Resource management, Security

Abstract: Security, availability and performance are critical to meet service level agreements in most Cloud computing services. In this paper, we build on the virtual machine technology that allows software components to be cheaply moved, replicated, and allocated on the hardware infrastructure to devise a solution that ensures users availability and performance requirements in Cloud environments. To deal with failures and vulnerabilities also due to cyber-attacks, we formulate the availability and performance attributes in the users perspective and show that the two attributes may often be competing for a given application. We then present a heuristics-based approach that restores application's requirements in the failure and recovery events. Our algorithm uses Markov chains and queuing networks to estimate the availability and performance of different deployment contexts, and generates a set of actions to re-deploy a given application. By simulation, we show that our proposed approach improves the availability and lowers the degradation of system's response time compared to traditional static schemes.

1 INTRODUCTION

The increasing demand for flexibility and scalability in obtaining and releasing computing resources in a cost-efficient manner has resulted in a wide adoption of the Cloud computing paradigm. The effective combination of existing technologies and modern business models in Cloud computing provides a suitable alternative to the users to deploy applications with diverse and dynamically changing requirements. While the benefits are immense, due to high complexity, even carefully engineered Cloud infrastructures are subject to a large number of failures and vulnerable to various types of cyber-attacks (e.g., server crashes, denial of service attacks). These vulnerabilities and failures evidently have a significant impact on the users applications and, as a consequence, there is a pressing need to address users availability and security issues (Samarati and De Capitani di Vimercati, 2010).

The traditional way to increase the availability of software is to employ fault tolerance techniques at development time. This approach requires users to build their applications by taking the system architecture into account. Unfortunately, the low-level architectural details are not widely available to the users because of the abstraction layers of Cloud computing. An alternative to the traditional approach is to offer fault tolerance as a service to users applications (Jhawar et al., 2012b). In this approach, a third party designs an appropriate fault tolerance policy based on users high level requirements and transparently applies it on the applications. To realize the notion of fault tolerance as a service, the Fault Tolerance Manager (FTM), presented in (Jhawar et al., 2012b), uses the virtualization technology to apply fault tolerance mechanisms at the granularity of virtual machine instances. For example, to increase availability, FTM replicates the entire virtual machine in which the application tasks are deployed by taking into account the failure characteristics and recovery behavior of the system. The constraints and placement techniques described in (Jhawar et al., 2012a) can then used to satisfy the deployment requirements of the chosen fault tolerance policy.

In this paper, we extend the concept of fault tolerance policy management, embedded in FTM, and present a solution concerning two important aspects of the service that were not analyzed previously. The first aspect is based on the observation that the fault tolerance policy that is initially selected and applied by FTM on an application may not be satisfied when system changes such as server crashes or network congestion happen. For example, a fault tolerance policy may require three replicas of an application to ensure a specified level of availability; however, if a replica fails at runtime, overall availability requirements are not satisfied. To avoid such situations, we need to take into account the current resource status of the system and adapt the fault tolerance service to ensure that the policy conditions of the application are satisfied. In this direction, we present a solution that dynamically identifies the system changes that affect users applications and responds to the changes by adapting the allocation to the new working status of the Cloud by means of a heuristic approach. We realize our solution as an *online controller* that, instead of computing an allocation from scratch when failures happen, uses the monitoring information (e.g., application workload, bandwidth availability, resource status) and virtualization technology constructs to satisfy users requirements by applying fewer actions. The online controller can be integrated within the FTM as a complementary component that ensures users requirements at runtime.

The second aspect considered in this paper is based on the observation that high availability and performance of an application may often be competing attributes. For example, availability can be improved by increasing the number of application replicas but that may diminish its performance due to additional processing and communication required to maintain consistency. In fact, Brewer's theorem states that consistency, availability, and partition tolerance are the three commonly desired properties by a distributed system, but it is impossible to achieve all three (Gilbert and Lynch, 2002). Our solution takes into account the availability and performance requirements while applying the fault tolerance service on an application. In particular, we design the online controller to balance both availability and performance attributes while generating a new configuration for a given application.

Existing solutions within this framework have not considered the aspect of balancing the application's availability and performance.

The remainder of the paper is as follows. Section 2 presents some preliminary concepts and our assumptions on users applications and the Cloud infrastructure. Section 3 provides a detailed discussion on the availability and performance metrics and their inter-relationship. Section 4 presents an approach to realize the online controller. Section 5 reports our simulation study results. Section 6 summarizes the related work and Section 7 outlines our conclusions.

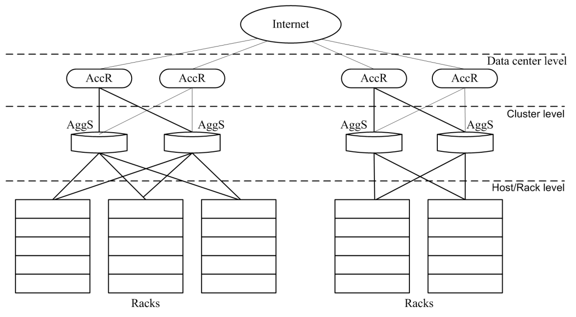

Figure 1: An example of Cloud infrastructure showing various deployment levels

2 SYSTEM MODEL

We consider the Cloud computing infrastructure built by inter-connecting large-scale, geographically distributed, data centers. Each data center (DC) consists of thousands of hosts that are organized into racks and clusters, and each host contains multiple processors, storage disks, memory modules and network interfaces. In terms of the network architecture: physical hosts are first connected via high-speed rack switches, which are in turn connected to (primary and backup) aggregation switches (AggS). The subsystem under an AggS can be viewed as a cluster. An AggS connects tens of racks to redundant access routers (AccR), and each AccR connects different data centers via the Internet backbone (Gill et al., 2011). Figure 1 illustrates an example of a Cloud infrastructure with two data centers, having hosts arranged as racks and clusters.

Let $\mathcal{H}$ be the set of all hosts in the system that is partitioned into a set $\mathcal{C}$ of clusters. We represent the resource characteristics of each host $h \in \mathcal{H}$ using a multi-dimensional vector $\overrightarrow{h}$, where each dimension represents the amount of *residual* resources (i.e., resources not yet allocated) of a specific type (e.g., CPU, memory) available on that host. For simplicity, we specify residual resource capacity of hosts using normalized values between 0 and 1. For example, $\overrightarrow{h} = (0.6, 0.5)$ implies that 60% of CPU, 50% of memory on host h is available for use. The service provider supplies computing resources to its users in the form of virtual machine (VM) instances of a given size. Let $\mathcal{V}$ be the set of VM instances. Since resource dimensions of VM instances are same as that of physical hosts, resource capacity of the VM can also be represented using multi-dimensional vector $\overrightarrow{v}$ and normalized values. For example, $\overrightarrow{v} = (0.4, 0.3)$ implies that VM $v \in \mathcal{V}$ will consume 40% of CPU and 30% of memory from a physical host. Note that a threshold on the usage of resources of physical hosts can be defined (e.g., (Jhawar et al., 2012a)), and heterogeneity of hardware resources can be made transparent to the users (e.g., Amazon EC2 service).

A user can deploy its applications using the Cloud-based infrastructure delivery service offered by the service provider. For the sake of generality, we do not model users applications making specific architectural assumptions and consider it to be a composition of a set of tasks. That is, let $\mathcal{A}$ be the set of all user applications in the system, then an application $A \in \mathcal{A}$ is a group of tasks $A = \{T_1, \ldots, T_m\}$. To achieve high scalability, the user can deploy each task of her application using an individual VM instance v. Furthermore, to improve fault tolerance, the user can engage with the FTM, running on the envisioned Cloud, by specifying its high-level availability goals. For example, the user can specify the *desired availability* Avail^d as 99.5%. Based on users requirements and system's resource status, the FTM selects an appropriate fault tolerance policy and associates each task $T_i \in A$ with a set of replicas $R(T_i) = \{t_{i,1}, \ldots, t_{i,|r|}\}$. The FTM then defines an allocation function $alloc : \mathcal{V} \rightarrow \mathcal{H}$ that allocates each replica $t_{i,j} \in T_i$, deployed in VM instances $v_{i,j} \in \mathcal{V}$, on a physical host in the infrastructure $h \in \mathcal{H}$. We note that there is a one-to-one relationship between tasks and VM instances.

The *alloc* function is invoked each time a request to deploy an application arrives, and it performs an allocation incrementally on the current resource status of the Cloud. We denote the specific output of the *alloc* function, that describes the exact location of each task (and its replicas) of an application, as a *configuration*. To determine a configuration, the *alloc* function may consider only a subset of the infrastructure. For example, *alloc* may consider only the hosts, network links, and switches within a given cluster to deploy an application. We denote the subsystem chosen within the infrastructure as a *deployment level DL*.

To meet users *availability* requirements, the allocation function must take into account the failure behavior of various infrastructure components. With respect to the failure behavior, we define a partial ordered hierarchy $(DL, \preceq_{DL})$, where DL denotes the deployment level and $\preceq_{DL}$ defines the relationship between different deployment levels. For example, $C_1 \preceq_{DL} DC_1$ indicates that data center DC_1 is a larger subsystem or deployment level when compared to cluster C_1. A transitive closure $\preceq^*_{DL}$ that indicates the "contains-in" relationship also exists on $\preceq_{DL}$. For example, $h_1 \preceq^*_{DL} C_1 \preceq^*_{DL} DC_1$ indicates that host h_1 is part of cluster C_1 that in turn exists in data center DC_1. Intuitively, the availability increases with increasing deployment level. That is, availability for an individual host is smaller than the availability of a cluster, which is still smaller than the availability of a data center. Therefore, the allocation function can take into account the *mean time between failures* for each deployment level MTBF_{DL} and map tasks of a given application based on users desired availability Avail^d.

To meet users *performance* requirements, the allocation function must take into account the amount of resources allocated to each task (size of VM instances) of the application and network latency between individual replicas of a task. The latency between hosts depends on how far they are from one another. For example, hosts in the same rack have a lower network latency than hosts across different clusters. Hence, if $L(DL)$ denotes the maximum latency between two hosts in the deployment level DL, then the allocation function *alloc* can decide suitable DL for each task replica based on users *desired performance* Perf^d, where Perf^d can be expressed in terms of expected response time.

3 AVAILABILITY AND PERFORMANCE METRICS

In this section, we discuss the availability and performance metrics in the users perspective, that is, we model the infrastructure parameters (e.g., failures and network congestion) that have an impact on the availability and performance of users applications. We then discuss the relationship between the two metrics.

3.1 Availability model

We consider a task replica, deployed on the physical host h, to have failed when h is completely nonoperational (i.e., h experiences a crash fault). In this context, an application is available if at least one replica of each of its tasks can be readily executed at a given point in time. Availability of an application can then be measured as the fraction of time for which it is available over a specified period of time. A fault tolerance scheme can improve the availability of a given application provided at least two replicas are used for each of its tasks. However, only considering two replicas may not be sufficient to avoid single points of failures. For example, if both (or all) the replicas are allocated on the same physical host, then the host failure may result in the complete failure of the application. This implies that the allocation function *alloc* must also take into account the location of individual replicas in the infrastructure to correctly realize a fault tolerance policy. In other words, a *configuration* is influenced both by the number of replicas of application tasks and their locations in the Cloud. We use $\text{nR}(T_i)$

to denote the number of replicas of task T_i currently available.

If $DL^{max}(T)$ is the highest deployment level within which the application task T must be allocated, then failures at $DL^{max}(T)$ or higher will cause the application to fail. For example, if task T of an application has two replicas allocated on different hosts of the same cluster C, then $DL^{max}(T)$ is cluster C and its failure results in unavailability of the application.

Availability of a given application can be estimated by measuring the MTBF of its configuration (based on the failure and repair properties of the hosts involved in the allocation). This behavior can be represented for an application using a Markov chain that considers all the tasks and their replicas (Jhawar and Piuri, 2012). In this paper, we extend such approach to represent the failure behavior of various deployment levels. As typically considered in the literature (Jung et al., 2010), failures at deployment levels are modeled as a Poisson process with rate $\lambda_{DL}=1/MTBF_{DL}$. Since application tasks are mapped on the infrastructure, its failure arrival process is also a Poisson process with rate $\sum_{DL}\lambda_{DL}$. This implies that a failure event affects the deployment level DL with probability

$$\frac{\lambda_{DL}}{\sum_{DL}\lambda_{DL}}$$

and causes the application having DL^{max} lower than DL to fail ($\forall T_i \in A$, $DL^{max}(R(T_i)) \preceq^{*}_{DL} DL$). Hence, the MTBF for an application A in a given configuration is

$$MTBF_A = \Big(\sum_{\forall DL,\ \exists T_i \in A,\ DL^{max}(R(T_i)) \preceq^{*}_{DL} DL} MTBF_{DL}^{-1}\Big)^{-1}$$

Given the users desired availability for its application Avail^d, similarly to (Jung et al., 2010), the FTM can first calculate the lower-bound of desired mean time between failures using the system's recovery policy:

$$MTBF_A^d = \frac{Avail^d \cdot MTTR}{1 - Avail^d}$$

where MTTR is the mean time to repair value. Then, based on the desired and estimated failure values and system's properties, it can define the deployment level for each application task to ensure users goals are satisfied. The identified deployments levels can be provided as placement constraints to the *alloc* function as follows.

- To limit a VM instance $V_i \in \mathcal{V}$ hosting the application task $T_i \in A$ on being allocated on a specified deployment level DL, the FTM can define the set $Restr=\{(R(T_i),DL_j)\}$ for each task, and ensure that the allocation function $alloc{:}\mathcal{V}\to\mathcal{H}$ satisfies the following condition:

$$\forall (R(T_i),DL_j) \in Restr \implies alloc(V_i) \in DL_j, DL_j \subseteq \mathcal{H}$$

  This condition is analogous to defining the maximum deployment level $DL^{max}(R(T_i))$ tasks.

- To avoid single points of failures, FTM can define the set $Distr=\{(t_i,t_j)|t_i,t_j \in R(T)\}$ specifying the task replicas that cannot be deployed on the same host and ensure that the allocation function $alloc{:}\mathcal{V}\to\mathcal{H}$ satisfies the following condition:

$$\forall t_i,t_j \in R(T), h \in \mathcal{H} : (t_i,t_j) \in Distr \implies alloc(v_i) \neq alloc(v_j)$$

This implies that FTM can translate users high-level availability requirements to low system relevant conditions by defining allocation conditions. Note that an application can meet its availability demands as long as the aforementioned conditions and values are satisfied. Furthermore, by means of such conditions, three deployment scenarios for *alloc* are possible: task replicas can be placed on the physical hosts that belong to the *i)* same cluster, *ii)* different clusters in the same data center, and *iii)* different data centers. The first configuration offers least failure independence as replicas cannot execute the fault tolerance protocol upon a single switch failure. The second configuration offers higher failure independence, and the third configuration offers high level of failure independence.

3.2 Performance model

We consider three parameters to influence the performance of an application: *i)* number of replicas for each task $nR(T_i)$, *ii)* amount of resources allocated to each task, and *iii)* network latency between task replicas. In general, the amount of processing and communication necessary to maintain a consistent state (as defined by the fault tolerance policy) increases as the number of replicas increase. In other words, the response time of the application increases as the number of replicas for each of its tasks increase. The mean response time also increases due to the network latency that increases as the distance between task replicas increase. Finally, the processing and memory capacity allocated to an application decides the amount of workload that its tasks can handle.

To quantify the performance of a given configuration, similarly to (Jung et al., 2010), (Qian et al., 2011), we use the layered queuing network formalism (Franks et al., 2009) as our application model.

This queuing network model allows us to predict the response time of the application and resource requirements of task replicas for a given configuration and workload. We represent application tasks using first-come first-served (FCFS) queues and resource requirements (size of the VM instance) using the processor sharing queues. The FTM can measure the parameters of the model such as the response time, whenever a request arrives, by calculating the delay between the incoming request and the outgoing response. Using the queuing network, similarly to availability, the FTM can then define a set of performance conditions that allow it to ensure that the allocation performed by the *alloc* function satisfies users desired performance requirements Perf^d. We discuss two such conditions:

- The first condition ensures that application tasks obtain the amount of CPU resources required to perform their tasks with optimal response time. Based on Perf^d and the queuing model discussed above, FTM can define the minimum size of the VM instances $\overrightarrow{v}$ for each task. The allocation function *alloc* then maps the VMs only on the hosts with the sufficient residual resources capacity $\overrightarrow{h}$

$$\forall h \in \mathcal{H} : \sum_{t_{i,j} \in R(T_i), T_i \in A, v_{i,j} \in \mathcal{V} | alloc(v_{i,j}) = h} \overrightarrow{v} \leq \overrightarrow{h}$$

- The second condition defines the maximum allowed network latency between two task replicas deployed in VM instances $v_i, v_j \in \mathcal{V}$. In particular, FTM can define a set $MaxL = \{(v_i, v_j, L_{max}) | v_i, v_j \in \mathcal{V}\}$ to specify the acceptable network latency L_{max} between VM instances v_i and v_j. The condition can then be used to ensure that the allocation function satisfies the following condition:

$$\begin{aligned} \forall v_i, v_j \in \mathcal{V} : \quad (v_i, v_j, L_{max}) \in MaxL \\ \implies Latency(alloc(v_i), alloc(v_j)) \leq L_{max} \end{aligned}$$

Note that $L(DL)$ denotes the maximum latency between two hosts in the deployment level DL (see Section 2). This implies that L_{max} should be greater than $L(DL)$ after allocation. We choose the number of replicas for a task dynamically (as discussed in Section 4) building on the initial configuration to satisfy users performance requirements. Lastly, note that the response time degradation for a configuration can be measured as the difference between the estimated mean response time and the desired mean response time.

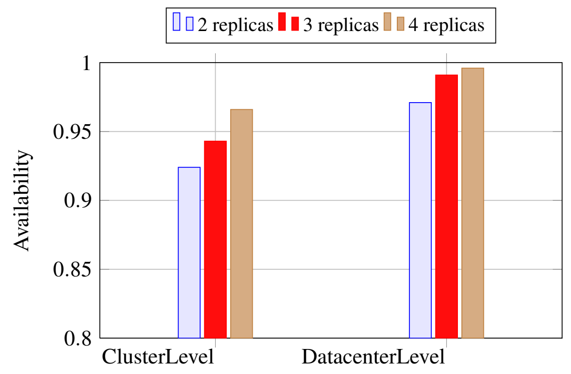

Figure 2: Availability at different deployment levels with varying number of replicas

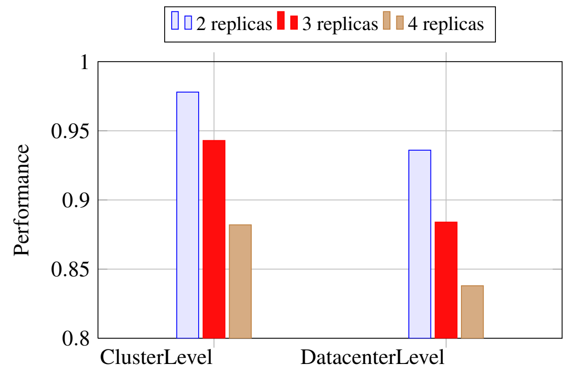

Figure 3: Performance in terms of response time at different deployment levels with varying number of replicas

3.3 Relationship between availability and performance

Since input parameters and availability values of hardware and system software are typically vendor-confidential, we derive this data from the tables published in (Kim et al., 2009)(Tang et al., 2007)(Smith et al., 2008). Based on this data and our modeling approach defined in the previous sections (using Markov chains and queuing networks), we show the relationship between the availability and performance parameters. In particular, we estimate the availability Avail^e and performance Perf^e of a given application (with a single task) having 2, 3 and 4 replicas, deployed in *i)* different clusters within a data center and *ii)* different data centers in the Cloud. Here, we do not discuss the scenario where task replicas are deployed within a cluster since their behavior only depends on the availability and performance attributes of the physical hosts. A detailed description of the simulation environment setup and our evaluation methodology is presented in Section 5.

Figure 2 illustrates how availability of the application changes for different configurations. We observe that the availability increases as the number of replicas of the application increase, and availability when the replicas are placed in different data centers is greater than the availability when replicas are placed in different clusters in the same data center. This implies that, availability of an application can be improved also by changing the location of its replicas.

Figure 3 illustrates how the performance of an application changes for different configurations. We assume that sufficient CPU capacity is allocated to each task, so the performance values reported here largely depends on the network latency. For the sake of clarity, the results are presented using normalized values between 0 and 1. We can observe that the performance decreases as the number of replicas of the application increase, and performance of an application when its replicas are placed in different clusters within a data center is greater than the performance when its replicas are placed in different data centers.

In the practice, we can see that there is a strict dependency between the two parameters, and therefore, the configuration choice for a given application must take into account both performance and availability metrics.

4 ONLINE CONTROLLER

In the previous section, we discussed the availability and performance models of users applications. Based on the models, we showed that availability and performance are competing parameters, and the allocation function that deploys users applications on the Cloud infrastructure must balance the two parameters while generating configuration solutions. We also presented an approach to translate high-level user requirements to low-level conditions, and discussed how the *alloc* function uses the models to generate configuration solutions while satisfying users requirements. Note that the specific output of the *alloc* function that describes the exact location of each task (and its replicas) in the Cloud infrastructure is denoted as configuration.

The Cloud computing environment is highly dynamic in terms of task activation, bandwidth availability, component failures and recovery. Due to the dynamic nature, the current configuration of a given application may become obsolete, and its performance and availability goals may not be satisfied. For example, replication level of a given task may be reduced due to a server crash, violating the availability goals. Hence, static deployment strategies that perform only initial allocation (such as *alloc* function) may not provide satisfactory results at runtime, and an adaptive approach to resource management is necessary.

One method to respond to system changes is to recompute the allocation from *scratch* using the *alloc* function. However, this method is rather naive and may not scale well during runtime. In this section, we present a heuristics-based approach that minimizes the performance and availability degradation of users applications due to various system changes. Our heuristic is realized as the online controller and introduced in the envisioned Cloud environment. The online controller uses the system's monitoring information (e.g., application workload, server's failure behavior, processor and bandwidth usage), and redeploys the applications as a response to the events that may violate the application's performance or availability goals. In particular, it generates a new configuration for users applications by creating new task replicas in case of host failures and by migrating individual tasks on (other working hosts) orthogonally across different deployment levels in the system to satisfy the overall performance and availability requirements.

In our context, the online controller is integrated in the FTM to provide fault tolerance support also during runtime. The activities required to change the current allocation status and re-deploy users applications are realized using the virtualization technology constructs. That is, by treating the task replicas as individual tasks, the online controller generates the new configuration in terms of the following actions:

- Launch(t,h): Due to system failures, the controller may identify that new replicas of a given task must be created. To realize this function, it instantiates a VM v, hosting the task replica $t \in T$, on the physical host $h \in \mathcal{H}$ using the Launch(t,h) action.
- Migrate(t,h_i,h_j): As a response to performance or availability degradation, the online controller may have to change the current location of a subset of task replicas. For example, to respond to network congestion in cluster C_1, the online controller may want to move task t_1 (initially hosted in C_1) to another cluster C_2. This function can be realized using the Migrate(t,h_i,h_j) action by specifying that VM instance deployed on host $h_i \in \mathcal{H}$, containing a task replica $t \in T$, must be moved to host $h_j \in \mathcal{H}$.
- Delete(t,h): Due to performance overhead, the online controller may need to reduce the replication level of a task. This action can be specified using the Delete(t,h) construct that removes the VM instance, hosting task replica $t \in T$, from host $h \in \mathcal{H}$.

We now define a mapping function $map : \mathcal{V} \rightarrow \mathcal{H}$

```
1:  RECONFIGURE
2:  INPUT    $alloc{:}\mathcal{V}\rightarrow\mathcal{H}$, $T_i{\in}A$, $\mathcal{H}$, $Restr$, $Distr$, $MaxL$
3:  OUTPUT   Set containing actions $Action$
4:  $Action{:=}\emptyset$
5:  /* If real availability is lower than the desired availability*/
6:  if Avail$^r$< Avail$^d$ then
7:    /*Identify the application tasks with replica failures*/
8:    for each $T_i \in a$ with nR$(T_i)$ < $|T_i|$ do
9:      /*Create task replicas in the original deployment level $DL$*/
10:     /*without violating the performance goals*/
11:     while (Avail$^e$≥ Avail$^d$ ∨ nR$(T_i)$ ≥ $|T_i|$) ∧ (Perf$^e$≥ Perf$^d$) do
12:       /*Include the launch action in $Action$*/
13:       $\forall t_{i,j}{\in}T_i$, $map(t_{i,j}){\in}DL$,
14:       $Action := Action{\cup}\{\textbf{Launch}(t_{i,j}, map(t_{i,j}))\}$
15:     end while /*Expected availability or replication level is met*/
16:   end for
17:   /*If expected availability is still lower than the desired one*/
18:   while Avail$^e$< Avail$^d$ do
19:   /*Move task replicas to the higher deployment levels $DL$*/
20:     if $\forall T_i{\in}a$, $t_{i,j}{\in}T_i$, $DL$, $map(t_{i,j}){\in}DL$ s.t. Perf$^e$≥ Perf$^d$ then
21:       /*Change in configuration by migrating task is possible*/
22:       $Action = Action \cup\{\textbf{Migrate}(t_{i,j}, alloc(v_{i,j}), map(t_{i,j}))\}$
23:     else
24:       /*Increase number of replicas to improve availability*/
25:       /*Traverse from highest deployment level to lowest*/
26:       $\forall T_i \in a, DL$, Perf$^e$≥ Perf$^d$,
27:       $Action = Action \cup \{\textbf{Launch}(t_{i,j},\ map(t_{i,j}))\}$
28:   end while
29: /* If real performance is lower than the desired performance*/
30: if Perf$^r$< Perf$^d$ then
31:   /*Identify the application tasks with affected response time*/
32:   for each $T_i \in a$ with $L(T_i) > L_{max}$ do
33:     /*Delete task replicas in the original deployment level $DL$*/
34:     /*without violating availability goals*/
35:     while (Perf$^e$≥ Perf$^d$ ∨ $L(T_i) \leq L_{max}$) ∧ (Avail$^e$≥ Avail$^d$) do
36:       /*Include the delete action in $Action$*/
37:       $\forall t_{i,j}{\in}T_i$, $map(t_{i,j}){\in}DL$,
38:       $Action := Action \cup \{\textbf{Delete}(t_{i,j},\ map(t_{i,j}))\}$
39:     end while /*Expected performance or latency obtained*/
40:   end for
41:   /*If expected performance is still lower than the desired one*/
42:   while Perf$^e$< Perf$^d$ do
43:   /*Move task replicas to the lower deployment level $DL$*/
44:     if $\forall T_i{\in}a$, $t_{i,j}{\in}T_i$, $DL$, $map(t_{i,j}) \in DL$ s.t. Avail$^e$≥ Avail$^d$ then
45:       /*Change in configuration by migrating task is possible*/
46:       $Action = Action \cup\{\textbf{Migrate}(t_{i,j}, alloc(v_{i,j}), map(t_{i,j}))\}$
47:     else
48:       /*Decrease the number of replicas to improve performance*/
49:       /*Traverse from lowest deployment level to highest*/
50:       $\forall T_i \in a, DL$, Avail$^e$≥ Avail$^d$,
51:       $Action = Action \cup \{\textbf{Delete}(t_{i,j},\ map(t_{i,j}))\}$
52:   end while
53: return $Action$ /*and call $alloc$ to schedule the actions*/
```

Figure 4: Pseudo-code algorithm for generating a new configuration plan

that behaves similarly to *alloc* but performs only a tentative search. That is, the mappings generated by *map* do not reflect on the infrastructure and must be explicitly *committed* using *alloc*. For example, using $map(v) = h$, the residual resource capacity, *estimated* performance and availability can be computed, and other placement conditions can be verified; but, the actual allocation can be performed using *alloc* (that gives *real* performance and availability values). Note that a task, deployed in a VM instance v, can be allocated on a host $h{\in}\mathcal{H}$ when the conditions specified in the *Restr*, *Distr*, *MaxL* sets are satisfied, and the host has sufficient resources to accommodate the task. We implement this problem as a bin-packing problem where the hosts represent the bins and VMs represent the items. The allocation function *alloc* can be realized using a packing algorithm that satisfies additional constraints such as the one described in (Jhawar et al., 2012a), (Machida et al., 2010), (Hermenier et al., 2011) (Seiden, 2002). The *alloc* function is invoked whenever there is a change in the application (e.g., addition of tasks) to perform an incremental allocation on the infrastructure. Once the application is deployed and initial configuration generated, the online controller is invoked to ensure the applications requirements at runtime.

Figure 4 depicts the pseudo-code of the algorithm that computes the set of actions that, when *committed*, generates a new configuration for a given application. It takes the current configuration, system status, application tasks and the sets specifying allocation conditions as input, and generates the sequence of actions that brings the system to a new feasible configuration state. The algorithm is invoked when a failure or performance degradation event happens. The algorithm consists of two main conditions, one concerning availability violation due to system failures (lines 5–28) and other concerning performance degradation (lines 29–52). If the real availability of an application is less than the desired one, we first identify the task replica failures and tentatively launch new replicas at the same deployment level using the *map* function. Note that the launch action is performed only until the current replication level is same as the original level and performance goals are not violated (lines 8–16). When addition of replicas does not satisfy the requirements, the algorithm tries to move task replicas to a higher deployment level using the Migrate action (note that the availability increases with increasing deployment levels). This action allows the online controller to generate the new configuration without increasing the resource consumption costs. If the performance condition conflicts by moving tasks to higher deployment levels, additional replicas must be created to improve the availability. To create new replicas, we start from higher deployment levels and move gradually to lower levels, creating the replicas at the level where availability and performance goals are fulfilled. These actions are realized using the migrate and launch actions (lines 18–28). When users availability requirements are satisfied, the algorithm realizes the actions in *Action* using the *alloc* function. In contrast, when real performance is less than desired performance, instead of launching new replicas, VM instances are deleted, and instead of moving higher in the hierarchy, migration takes place to lower deployment levels. These actions are based on the observation that decrease in the deployment or replication level, improves the application performance. Note that the online controller is invoked only when an application experiences failures or performance degradation, and therefore, it is suitable for long-running tasks; short-running tasks are practically managed by the FTM during initial deployment.

5 EXPERIMENTAL EVALUATION

In this section, we report the simulation results of the experiments we conducted to evaluate the online controller. In particular, we validate the controller in terms of *i)* the time required to compute a new configuration using the algorithm in Figure 4, *ii)* increase in overall availability of an application, and *iii)* improvement with respect to the performance in varying system contexts.

Setup. The hardware failure rates are provided by many companies in the form of tables (Tang et al., 2007). However, the task of attributing the cause of failures and estimating the mean time to failure for software components (e.g., hypervisor) is difficult. To this aim, we derive the input parameter values from (Kim et al., 2009) (e.g., 2.654e+003 and 3.508e-001 as mean time to failure and recovery respectively for virtualized hosts) and use the ORMM Markov analysis tool (Jensen, 2011) to obtain the output measures. To make the results applicable for systems with different MTBF and MTTR values, we normalize all times to MTBF, and vary the MTTR over a range from 0.01 to 2.0. This indicates variation in repair times from 10% to 200% of actual MTBF, hence providing different availability values. Similarly, the parameters for performance are obtained using the layered queueing network solver (Franks et al., 2009). We note that Markov analysis tool and queueing network solver are used offline, and output parameter values are used to configure the online controller simulator written in C++. The simulation is executed on a machine having Intel i7-2860QM 2.50GHz processor, with 16GB of memory, running Windows 7 operating system.

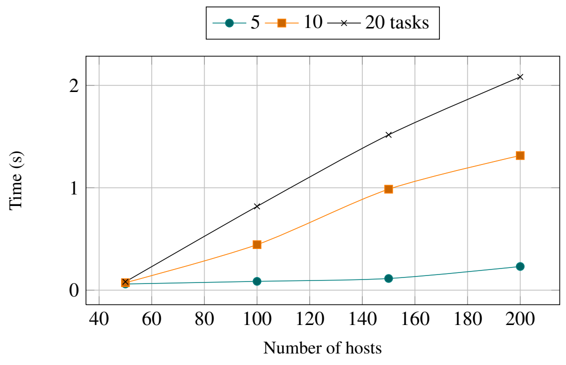

Figure 5: Time to compute the new configuration solution wrt the number of hosts, for different number of tasks

We configure the Cloud infrastructure by randomly initializing the hosts with different amounts of residual resources. This forms the basis for our online controller to manage VM instances of a given application on the current resource status of the Cloud. The utilization of hosts are updated after launch, migrate and delete actions, providing results on incremental resource management. We also initialize varying network latency between deployment levels, MTBF and relative MTTR rates. For example, network latency between VM instances vary depending on the deployment configuration (if replicas share a host, rack, cluster, or a data center). The network latency within a host is considered 0; if latency between two hosts in a rack is x, latency is set to $1x$, $1.5x$ and $2.5x$ for different racks, clusters, and data centers respectively. We select applications with different number of tasks (see below) and randomly choose replicated task sets. The simulation results presented here are the mean values of ten executions of each configuration.

Processing time evaluation. We study the amount of time it takes for the online controller to compute the new configuration for applications with 5, 10 and 20 tasks, on an infrastructure containing 50 to 200 hosts. Figure 5 illustrates how the processing time varies for different contexts. For smaller size instances of applications and infrastructure, the solution can be computed in the order of a few milli-seconds. When the application contains 20 tasks, and infrastructure has 200 hosts, the processing time is about 2 seconds. In particular, we observe higher processing time when cluster level failures affect multiple task replicas. Although computing new configuration has acceptable scalability, note that the amount of time to actually reconfigure the system may be larger due to several system parameters (e.g., the time to migrate a VM may be in the order of minutes, particularly when the VM size is large, and the target host is connected via Internet).

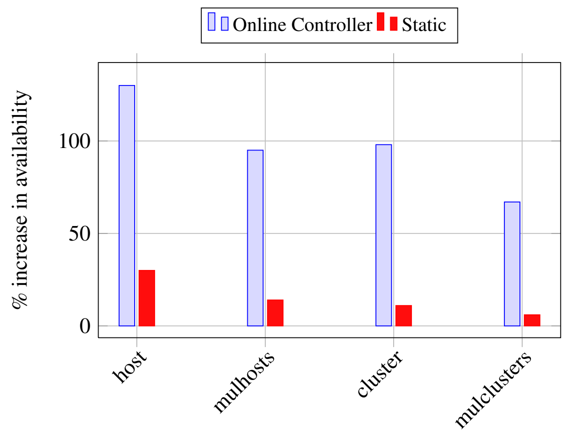

Figure 6: Percentage increase in the availability due to reconfiguration

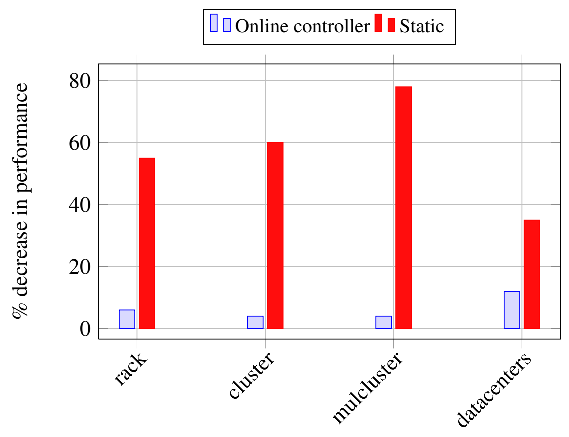

Figure 7: Percentage change in term of response time/performance degradation due to reconfiguration

Availability and performance evaluation. We allocate the tasks, as defined by the *alloc* function, using the first-fit bin-packing strategy. This allocation is considered *static*, and simulations are then compared against the static scheme. To evaluate the increase in availability, we introduce failures following the MTBF values at different deployment levels (hosts and clusters) in the infrastructure. Similarly, to evaluate the performance, we randomly select a deployment level (hosts and clusters) and assume an increase in the network latency connecting those resources. For each failure and change in network latency, the online controller is invoked to compute the new configuration.

We calculate the percentage increase in the availability of an application with 10 tasks, comparing static approach and our proposed approach, for different failure levels. A cluster failure implies assuming all the hosts in that cluster have failed. Figure 6 shows the difference between the availability levels. In case of single host failures (that have least MTBF values), the online controller is estimated to improve an application's availability by 120 percent (when compared to static allocation). The increase in availability is about 95 percent in case of multiple hosts and single cluster failures, whereas, in case of multiple cluster failures (with higher MTBFs), application's availability is estimated to improve by 70 percent when compared to static deployment methods.

Similarly, we calculate the change in the response time of an application by increasing the network latency at different levels in the infrastructure. Figure 7 illustrates that online controller approach can significantly reduce the performance degradation among applications when changes in the network latency affects multiple clusters within a data center. On the contrary, the percentage improvement in performance due to disruptions at data center level is marginal. Since, we regenerate or migrate the task replicas in the event of failures or performance degradation as opposed to the static scheme, the results cannot be consistently compared. Nevertheless, the simulation results clearly show that our approach of online controller can provide high levels of graceful service degradation to the users.

6 RELATED WORK

Virtualization technology is an important enabler of the Cloud computing paradigm. It allows a service provider to address concerns related to scalability and issues with heterogeneous computing resources in data centers (Hermenier et al., 2011), (Machida et al., 2010), (Qian et al., 2011) (Cully et al., 2008). The FTM framework uses the virtualization technology to deliver fault tolerance as a service. The overall conceptual framework for FTM and an outline of the required approach is presented in (Jhawar et al., 2012b). Similarly to FTM, other fault-tolerant real-time systems such as Mars (Kopetz et al., 1989) also use static replication of processing tasks to ensure reliability. In the present paper, we extend the concept of fault tolerance policy management, embedded in FTM, with an online controller to meet users requirements after their applications are deployed in a given configuration. Our approach uses the constraints defined in (Jhawar et al., 2012a) to impose restrictions on the allocations to be made on the infrastructure.

Availability and response time are often used as standard service level agreement metrics in Cloud computing services (Buyya et al., 2011) (De Capitani di Vimercati et al., 2012). The well-known Brewer's theorem states that consistency, availability, and partition tolerance are the three commonly desired properties by a distributed system, but it is impossible to achieve all three (Gilbert and Lynch, 2002). Several studies have represented availability and performance attributes using analytical models and demonstrated very accurate results in a time-efficient manner. (Kim et al., 2009) presented availability models for virtualized and non-virtualized servers using hierarchical analytical models and demonstrated encouraging results with the use of virtualization. Similar model is used by Jhawar et al. (Jhawar and Piuri, 2012) to model component failures at different levels in data centers, correlation between failures, and impact boundaries.

Dynamic creation of replicas to deal with system failures has been used before. For example, VMWare High Availability (HA) (VMware, 2007) allow a virtual machine on a failed host to be re-instantiated on a new machine and (Pu et al., 1988) uses regeneration of new data objects to account for reduction in redundancy in the Google File System. The work most relevant to the proposal in this paper is by Jung et. al (Jung et al., 2010) (Jung et al., 2008) that examines how virtualization can be used to provide enhanced solutions to the classic problem of ensuring high availability while maintaining performance of multi-tier web services. Software components are restored whenever failures occur and component placement is managed using information about application control flow and performance predictions. Our work is different from the existing systems in the way we handle system failures to create replicas and orthogonally migrate them in various deployment levels in the Cloud. Moreover, other approaches that generate new configurations at runtime do not take into account the placement constraints as we do in this paper.

The performance impact of resource allocation on web applications has been studied in (Urgaonkar et al., 2005), but it does not combine availability requirements and regeneration of failed components. Several works on dependability have highlighted the necessity to trade-off between availability and performance (Shin et al., 1989) (Sahai et al., 2002).

7 CONCLUSIONS

In this paper, we have highlighted that adaptive resource management is critical for fault tolerance of applications in Cloud computing. We extended the concept of fault tolerance policy management, embedded in FTM (that provides fault tolerance as a service), with an online controller to dynamically change the replication levels and deployment configurations in the event of system failures (e.g., server crashes and security exploits resulting in the denial of service). First, we formulated availability and performance of applications using Markov chains and layered queuing networks, and showed that the two attributes may be competing with each other in a given configuration. Then, using the models, we presented the online controller that realizes a heuristics-based algorithm to restore application's requirements at runtime. Finally, we reported our simulation results and showed that the online controller can significantly improve the availability and lower the degradation of system response times compared to traditional static schemes. Our future work will extend the models to a larger scale and perform case studies on specific software architectures in Cloud computing environments.

ACKNOWLEDGMENTS

This work was supported in part by the Italian Ministry of Research within PRIN project "GenData 2020" (2010RTFWBH), and by Google, under the Google Research Award program.

REFERENCES

Buyya, R., Garg, S. K., and Calheiros, R. N. (2011). Sla-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions. In *Proc. of the 2011 International Conference on Cloud and Service Computing*, pages 1–10, Washington, DC, USA.

Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. (2008). Remus: high availability via asynchronous virtual machine replication. In *Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, San Francisco, California.

De Capitani di Vimercati, S., Foresti, S., and Samarati, P. (2012). Managing and accessing data in the cloud: Privacy risks and approaches. In *Proc. of the 7th International Conference on Risks and Security of Internet and Systems*, Cork, Ireland.

Franks, G., Al-Omari, T., Woodside, M., Das, O., and Derisavi, S. (2009). Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering*, 35(2):148–161.

Gilbert, S. and Lynch, N. (2002). Brewer's conjecture and the feasibility of Consistent, Available, Partition-tolerant web services. *SIGACT News*, 33(2):51–59.

Gill, P., Jain, N., and Nagappan, N. (2011). Understanding network failures in data centers: measurement, analysis, and implications. *ACM Computer Communication Review*, 41(4):350–361.

Hermenier, F., Lawall, J., Menaud, J.-M., and Muller, G. (2011). Dynamic Consolidation of Highly Available Web Applications. Technical Report RR-7545, INRIA.

Jensen, P. A. (2011). Operations Research Models and Methods – Markov Analysis Tools. Available at www.me.utexas.edu/jensen/ormm/excel/markov.html.

Jhawar, R. and Piuri, V. (2012). Fault tolerance management in iaas clouds. In *Proc. of 2012 IEEE First AESS European Conference on Satellite Telecommunications*, pages 1–6, Rome, Italy.

Jhawar, R., Piuri, V., and Samarati, P. (2012a). Supporting security requirements for resource management in cloud computing. In *Proc. of the 15th IEEE International Conference on Computational Science and Engineering*, Paphos, Cyprus.

Jhawar, R., Piuri, V., and Santambrogio, M. (2012b). Fault tolerance management in cloud computing: A system-level perspective. *IEEE Systems Journal*, PP(99).

Jung, G., Joshi, K., Hiltunen, M., Schlichting, R., and Pu, C. (2010). Performance and availability aware regeneration for cloud based multitier applications. In *Proc. of 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 497–506, Chicago, IL, USA.

Jung, G., Joshi, K. R., Hiltunen, M. A., Schlichting, R. D., and Pu, C. (2008). Generating adaptation policies for multi-tier applications in consolidated server environments. In *Proc. of the 2008 International Conference on Autonomic Computing*, pages 23–32, Washington, DC, USA.

Kim, S., Machida, F., and Trivedi, K. (2009). Availability modeling and analysis of virtualized system. In *Proc. of 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 365–371, Shanghai, China.

Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., and Zainlinger, R. (1989). Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. *IEEE Micro*, 9(1):25–40.

Machida, F., Kawato, M., and Maeno, Y. (2010). Redundant virtual machine placement for fault-tolerant consolidated server clusters. In *Proc. of Network Operations and Management Symposium*, pages 32–39, Osaka, Japan.

Pu, C., Noe, J., and Proudfoot, A. (1988). Regeneration of replicated objects: a technique and its eden implementation. *IEEE Transactions on Software Engineering*, 14(7):936–945.

Qian, H., Medhi, D., and Trivedi, T. (2011). A hierarchical model to evaluate quality of experience of online services hosted by cloud computing. In *Proc. of IFIP/IEEE International Symposium on Integrated Network Management*, pages 105–112, Dublin, Ireland.

Sahai, A., Machiraju, V., Sayal, M., Moorsel, A. P. A. v., and Casati, F. (2002). Automated sla monitoring for web services. In *Proc. of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management: Management Technologies for E-Commerce and E-Business Applications*, pages 28–41, London, UK.

Samarati, P. and De Capitani di Vimercati, S. (2010). Data protection in outsourcing scenarios: issues and directions. In *Proc. of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 1–14, Beijing, China.

Seiden, S. S. (2002). On the online bin packing problem. *ACM Journal*, 49(5).

Shin, K., Krishna, C. M., and hang Lee, Y. (1989). Optimal dynamic control of resources in a distributed system. *IEEE Transactions on Software Engineering*, 15(10):1188–1198.

Smith, W. E., Trivedi, K. S., Tomek, L. A., and Ackaret, J. (2008). Availability analysis of blade server systems. *IBM Systems Journal*, 47(4):621–640.

Tang, C., Steinder, M., Spreitzer, M., and Pacifici, G. (2007). A scalable application placement controller for enterprise data centers. In *Proc. of 16th International conference on World Wide Web*, pages 331–340, Alberta, Canada.

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. (2005). An analytical model for multi-tier internet services and its applications. *SIGMETRICS Performance Evaluation Review*, 33(1):291–302.

VMware (2007). White paper: Vmware high availability concepts, implementation and best practices.