Autonomic SLA-driven Provisioning for Cloud Applications

Nicolas Bonvin, Thanasis G. Papaioannou and Karl Aberer
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

***Abstract*—Significant achievements have been made for automated allocation of cloud resources. However, the performance of applications may be poor in peak load periods, unless their cloud resources are dynamically adjusted. Moreover, although cloud resources dedicated to different applications are virtually isolated, performance fluctuations do occur because of resource sharing, and software or hardware failures (e.g. unstable virtual machines, power outages, etc.). In this paper, we propose a decentralized economic approach for dynamically adapting the cloud resources of various applications, so as to statistically meet their SLA performance and availability goals in the presence of varying loads or failures. According to our approach, the dynamic *economic fitness* of a Web service determines whether it is replicated or migrated to another server, or deleted. The economic fitness of a Web service depends on its individual performance constraints, its load, and the utilization of the resources where it resides. Cascading performance objectives are dynamically calculated for individual tasks in the application workflow according to the user requirements. By fully implementing our framework, we experimentally proved that our adaptive approach statistically meets the performance objectives under peak load periods or failures, as opposed to static resource settings.**

***Index Terms*—cost-efficiency, replication, migration, net benefit, performance elasticity, web services**

I. Introduction

With the emergence of the cloud computing paradigm, avoiding high capital investment for infrastructure becomes viable. Lower operational expenses are expected for renting cloud resources on demand by the application providers. Although achievements in automated cloud resource provisioning were enough for the first wave of "best effort" application deployments, adaptive resource allocation for satisfying the performance and availability objectives of mission-critical application remains an open issue. With static resource allocation (based on resource planning and over-provisioning), a cluster system would be likely to leave 50% of the hardware resources (i.e. CPU, memory, disk) idle, thus baring unnecessary operational expenses without any profit (i.e. negative value flows). Moreover, as described in [1], the performance of multiple identical virtual machines may greatly vary, and thus might drastically reduce the performance of a distributed application. On the other hand, as clouds scale up, software and hardware failures of any type (e.g. software stales, virtual machines go "wonky" (i.e unstable), hardware or rack or even datacenter failures, etc.) are unavoidable and often *spatially correlated* [2]. Resource redundancy should be employed to increase service reliability and availability, yet in a cost-effective way. Another concern is that, as the size of the cloud increases, its administrative overhead becomes unmanageable.

In this paper, we focus on cost-effective autonomic resource allocation, so as to adaptively satisfy service level agreements (SLAs) for performance and availability statistical guarantees against load variations and software / hardware failures. We propose a middleware ("Scattered Autonomic Resources", referred to as *Scarce*) that performs supple sharing to avoid stranded and underutilized computational resources and dynamically adapts to changing conditions, such as failures, load variations or "wonky" server (or virtual machines). As our framework works indifferently on top of virtualized and/or physical servers, henceforth, we use the terms *server* and *virtual machine* interchangeably, unless stated otherwise. Our middleware simplifies the development of online applications composed by multiple independent components (e.g. web services) following the Service Oriented Architecture (SOA) principles. We consider a *virtual economy*, where components are treated as individually rational entities that rent computational resources from servers, and migrate, replicate or exit according to their economic *fitness*. This fitness expresses the difference between the utility offered by a specific application component and the cost for retaining it in the cloud. The server rent price is an increasing function of the utilization of server resources. Moreover, components of a certain application are dynamically replicated to *geographically-diverse* servers according to the availability requirements of the application. A preliminary version of our work without offering any performance guarantees was presented in [3].

Our approach combines the following unique characteristics :

- Adaptive adjustment of cloud resource allocation in order to statistically satisfy response time or availability SLA requirements.
- Cost-effective resource allocation and component placement for minimizing the operational costs of the cloud application.
- Detection and removal or replacement of stale cloud resources.
- Component replication and migration for accommodating load variations and for supple load balancing.

- Decentralized self-management of the cloud resources for the application.
- Geographically-diverse placement of clone component instances, as shown in [3].

Having implemented a full prototype of our approach, we experimentally prove that it effectively accommodates load spikes, it satisfies compliance to the SLA response-time requirements, it cost-effectively utilizes the cloud resources, and it provides a dynamic geographical replica placement without thrashing. We finally reveal the *trade-off* between cost-effectiveness and meeting strict SLA requirements; the latter may necessitate a more conservative (over-provisioning) resource allocation approach.

The remainder of this paper is organized as follows: in Section II, we present a motivating example application. In Section III, we describe our economic approach for autonomic component replica management. Section IV describes how SLAs are propagated. In Section V, we describe how the components dynamically adapt their resources to honor their SLA. Subsection V-A describes how "wonky"and cloud resources are detected and removed. In Section VI, we present our experimental results. In Section VII, we overview the related work and, finally in Section VIII, we conclude our work.

II. Motivation

Elastic platforms are becoming more and more popular and start to be a viable alternative to host distributed applications and web applications in particular. In a typical cloud infrastructure, a user can rent virtual machines (VM) and allocate dedicated resources (such as CPU cores, RAM, disk space, etc.) to them in order to closely match the application needs. Moreover, through the cloud infrastructure application programming interface (API), the user is able to programmatically increase or decrease the resources allocated to a virtual machine, and can also start new VMs or stop unused ones. The virtualization technologies that have greatly contributed to the success of cloud computing also come with some drawbacks. In effect, a user does not have full control over the underlying infrastructure. For example, in today's public cloud infrastructures, such as EC2 or Rackspace Cloud, a user does not have the possibility to choose on which physical server a VM will be started. Moreover, a VM may migrate during its lifetime from one physical server to another. Also, a user has no control over the virtual machines that are collocated on the same physical server. This may have a large performance impact, if for example, a collocated VM performs an I/O intensive work. Figure 1 shows an architectural view of a distributed application hosted by a cloud computing infrastructure. The application consists of 5 replicated components (*Comp 1* to *Comp 5*) and spans 4 VMs and 3 physical servers. In such elastic infrastructures, each physical server hosts several VMs. As the application owner has usually no control on where its VMs are hosted, the application responsiveness may suffer from an overloaded VM of another customer hosted at the same physical server as the one employed by the application VM.

A well-engineered cloud application should be able to detect slow or "wonky" VMs and react accordingly. A desirable reaction would be to first gradually redirect the traffic for the components of the "wonky" VM to their replica components elsewhere. Second, if the overall performance of the application has suffered considerably, a new VM should be started in order to take over the redirected traffic. At some point, the "wonky" VM will not receive any traffic and can safely be removed, thus saving rental costs.

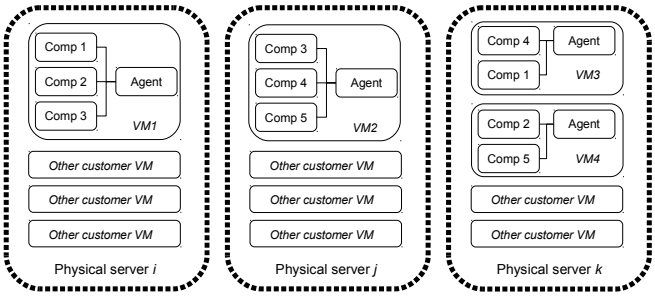

Fig. 1. Overview of a distributed application composed by 5 components (i.e. *Comp 1* to *Comp 5*) deployed on a cloud computing infrastructure.

III. Scarce: the quest of autonomic applications

A. The approach

We consider applications formed by many independent and stateless components that interact among each other to provide a service to the end user, as in the Service Oriented Architecture (SOA) paradigm. A component is self-managing, self-healing and is hosted by a server (or a virtual machine), which in turn is allowed to host many different components. A component can stop, migrate or replicate to a new server according to its load or availability. The approach to maintain high availability is explained in Section III-E.

B. Server agent

The server agent is a special component that resides at each server and is responsible for managing the resources of the server according to our economic-based approach, as shown in Figure 2. Specifically, this agent is responsible for starting and stopping the components of the various applications at the local server, as well as checking the "health" of the services (e.g. by verifying if the service process is still running, or by firing a test request and checking that the corresponding reply is correct). The agent knows the properties of every service that composes the application, such as the path of the service executable, and its minimum and maximum replication factor. This knowledge is acquired when the agent starts, by contacting another agent (referred to as "bootstrap agent"). Any running agent participating in the application cluster can act as a bootstrap agent.

During the startup phase, the agent also retrieves the current routing table from the bootstrap agent. A routing table consists

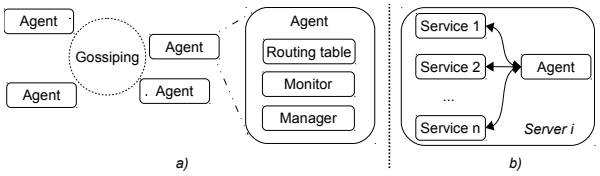

Fig. 2. *a)* Agents communicate using a gossiping protocol *b)* A server hosts many services and an agent.

of a mapping between services and servers (*cf.* Section III-C). The number of replicas of a service and their placement are handled by a distributed optimization algorithm autonomously executed by the agents.

In an untrustworthy environment, where a server agent may be malicious, the functionality of decision making could be implemented directly in the component itself. While being robust to strategic behaviors of server agents, this approach tends to waste resources, as every component would have to perform the tasks of a server agent (i.e. maintaining the routing table, gossiping, etc.).

We assume that a server belongs to a rack, a room, a datacenter, a city, a country and a continent. Note that finer or coarser geographical granularity could also be considered, especially in a cloud environment where the rack and room information may not be available. A label of the form "continent-country-city-datacenter-room-rack-server" is attached to each server in order to precisely identify its geographical location. For example, a possible label for a server located in a data center in London could be "EU-UK-LON-D1-C03-R11-S07".

C. Routing table

Instead of using a centralized repository for locating services, such as a UDDI registry (uddi.xml.org), each server keeps locally mappings (i.e. a routing table) between components and servers. This routing table (e.g. Table I) is maintained by a gossiping algorithm (see Figure 2), where each agent contacts a random subset [$log(N)$ where N is the total number of servers] of remote agents and exchanges information about the services running at their respective server.

TABLE I

component	*servers*
component 1	server A, server B
component 2	server B, server C
component 3	server A

D. Economic model

Service replication should be highly adaptive to the processing load and to failures of any kind in order to maintain high service availability. To this end, each component is treated by the server agent as an individual optimizer that acts autonomously so as to ascertain the pre-specified by the SLA availability guarantees and to *balance* its economic fitness. Time is assumed to be split into epochs. At every epoch, the server agent verifies from the local routing table that the minimum number of replicas for every component is satisfied; thus, no global or remote knowledge is required. If the required availability level is not satisfied and if the service is not already running locally, the agent starts the service. When the service has started, the server agent informs all others by using a hierarchical broadcast to update their respective routing tables.

At each epoch, a service pays a *virtual rent* to the servers where it is running. The virtual rent corresponds to the usage of the server resources, such as CPU, memory, network, disk (I/O, space). A service may be replicated or migrated to another server, or stopped by the server agent. These decisions are made based on the service demand, the renting cost and the maintenance of high availability upon failures. There is no global coordination and each server agent behaves independently for each hosted service. Only one replica of a service is allowed to be stopped at the same epoch by employing Paxos [4] distributed consensus algorithm. The virtual rent of a server is updated at the beginning of a new epoch by the server agent. The price of the other servers participating in the application cluster are updated by the same gossiping algorithm that is used to maintain the routing table.

The actions (i.e. replication, migration, stop) performed by the server agent on behalf of a component c hosted at a server s are directly related to the economic fitness or $balance$ of the component c, which is given by:

$$balance_c = utility_c - rent_s \,, \tag{1}$$

The utility of a component corresponds to the value that it creates for the various applications that employ it and it can be safely assumed to be an increasing function of the server resources utilized by the component. We denote as x_c the usage percentage of the server resources by the component c and as x_s the resource utilization percentage of the server s. x_c can be calculated as follows:

$$x_c = \frac{w_c \cdot cpu_c + w_m \cdot mem_c + w_n \cdot net_c + w_d \cdot disk_c}{w_c + w_m + w_n + w_d} \,, \tag{2}$$

where $cpu_c, mem_c, net_c, disk_c$ are the component usage percentages for CPU, memory, network and disk respectively. w_c, w_m, w_n, w_d are weights to adapt to different kinds of application components (CPU-intensive, I/O-intensive, etc.) and they can be determined based on a process performance profiler. x_s reflects the *contention level* of the server for a given application component, and is calculated similarly to x_c by employing the resource utilization percentages for the total server instead of the single component ones and the *corresponding* weights w_c, w_m, w_n, w_d of the component. For example, x_s of a server s may be high for a certain component and low for another one to enable better multiplexing of server resources. To this end, the utility of a component c residing at server s is assumed to be given by the following convex formula:

$$utility_c = k \frac{x_c - x_s^*}{(C - x_c)^2}, \tag{3}$$

where $C > 100$ is a constant determining the starting point of the fast-increasing part of the curve and k is a normalization factor for maintaining the balance of the component close to 0 for moderate resource utilization. The selection of the parameters k, C determines the reactivity of the balance with respect to the resource usage. x_s^* is a certain component utilization threshold that determines when the component residing at s is economically *fit-enough* to replicate, i.e.:

$$x_s^* = \frac{srvMinUsage}{|components_s|}\,, \tag{4}$$

where $|components_s|$ is the number of components running at the server and $srvMinUsage$ is a percentage threshold denoting a *soft limit* for server utilization, e.g. 25%. The utility function is chosen so that it grows exponentially to the usage and it is 0 for $x_c = x_s^*$. The virtual rent paid by the component c to the server s is given by:

$$rent_s = conf_s \cdot x_s\,, \tag{5}$$

where $conf_s$ is a subjective estimation of the server quality and reliability based on technical factors (hardware quality, datacenter connectivity, redundancy, etc.) as well as non-technicals ones (e.g. political and economical stability of the country hosting the server, etc.).

Based on the $balance_c$, at the beginning of a new epoch, a component may:

- *migrate or stop:* if it has negative balance for the last f epochs. First, the component calculates its availability without itself. If the availability is satisfactory, the component stops. Otherwise, it tries to find a less expensive (i.e. busy) server that is closer to the client locations (according to maximization formula (7)). To avoid oscillations of a replica among servers, the migration is only allowed if the following *migration conditions* apply:
  - The minimum availability is still satisfied using the new server,
  - the absolute price difference between the current and the new server is greater than a threshold,
  - the x_s of the current server s is above the soft limit $srvMinUsage$.
- *replicate:* if it has positive balance for the last f epochs, it may replicate. For replication, a component has also to verify that it can afford the replication by having a positive balance b' for consecutive f epochs:

  $$b' = balance_c - (1+\phi)\cdot rent_{s'}$$

  where $rent_{s'}$ is the current virtual rent of the candidate server s' for replication (randomly selected among the top-k ones ranked according to the formula (7)), while the factor $1+\phi$ accounts for a $\phi{\cdot}100\%$ increase at this rent price in the next epoch due to the potentially increased usage of the candidate server (an upper bound of $\phi = 0.2$ can typically be assumed). This action aims to distribute to load of the current server towards another one located closer to the clients. Thus, it tends to decrease the processing and network latency of the requests for the component.

E. Maintaining high-availability

Server or component failures or network partitioning may unexpectedly occur at any time and they are often spatially-correlated. As estimating the probability of each server to fail necessitates access to a large set of historical data and private information of the server, we adopt the approach of [5] for maintaining high availability by geographically-diverse placement of component replicas. The availability of a service i is defined as the sum of *diversities* between each distinct pair of servers, i.e.:

$$avail_i = \sum_{i=0}^{|S_i|}\sum_{j=i+1}^{|S_i|} conf_i \cdot conf_j \cdot diversity(s_i, s_j) \tag{6}$$

where $S_i = (s_1, s_2, \ldots, s_n)$ is the set of servers hosting replicas of the service i and $conf_i$, $conf_j \in [0,\ 1]$ are the confidence levels of servers i, j. The diversity function returns a number calculated based on the geographical distance among each server pair. This distance can be represented as a n-bit number, having each bit corresponding to the n location parts of a server, e.g. continent, country, city, data center, room rack, server etc. The most significant bit (leftmost) represents the wider enclosing geographical location (e.g. the continent), while the least significant bit (rightmost) represents the server. When two servers are not in the same location part, their corresponding *diversity* bit is set to 1, otherwise to 0. The diversity values of the server pairs are summed up, because having more replicas in distinct servers always results in increased availability regardless of their location. A component knows the locations of its replicas by the local routing table at the server where it is hosted.

The availability of a component should always be kept above a minimum level th, which is derived by the SLA. When the availability of a component falls below th, a new service instance should be started (i.e. replicated) at a new server. The best candidate server is selected so as to maximize the *net benefit* between the diversity of the resulting set of replica locations for the service and the virtual rent of the new server, i.e.

$$\sum_{k=1}^{|S_i|} g_j \cdot conf_j \cdot diversity(s_k, s_j) - rent_{\ j}\,, \tag{7}$$

where $rent_{\ j}$ is the virtual rent price of candidate server j. g_j is a weight related to the proximity (i.e. inverse average diversity) of the server location to the geographical distribution of the client requests for the service (*cf.* [5]). Note that client requests may come from other components. As a result, the components will tend to replicate closer to the components that heavily rely on the services of the former. The components rank servers according to their net benefit (7) and randomly choose the target for replication among the top-k ones for avoiding server congestion. Note that the same approach according to (7) is used for choosing the candidate server for component migration.

IV. Meeting SLA performance guarantees

Autonomic migration or replication of the application components in order to utilize in a fair manner the available resources may not always be good enough to guarantee acceptable end-to-end service quality. If the latency of client requests is not satisfactory and the allocated resources to the application are not underutilized, one solution is to give more resources to the application, e.g. by increasing the number of cores of a virtual machine (VM), or by starting a new VM.

To this end, our framework is able to manage the physical resources dedicated to the application based on a SLA defined by the application owner. If the SLA is not met, then the framework asks for more resources via the cloud API. Or, if the application easily honors the SLA, it can remove some extra resources.

A. Cascading performance constraints

The application owner requires the compliance of the performance to certain constraints pre-specified in a SLA, e.g. an upper bound on the response time for a service request. In case of complex applications that consist of many components that have dependencies on each other (as the one depicted in Figure 3), it is not always possible to comply to the SLA-driven performance constraints, unless the latter are individually set to each component constituting the application. However, the performance constraints can be directly derived by the SLA only for the entry component (i.e. the one that receives the user requests) of the application; derivation of the performance constraints for the other components by the SLA would necessitate *a priori* knowledge of the application internals, e.g. the exact execution workflow, the hardware resources allocated to each component, the component computational needs, etc. Moreover, the performance constraints for each component should change over time, in order for the SLA to be met, due to i) the dynamic demand for the application, ii) the fact that its components are multiplexed with the components of other applications and iii) the dynamic behaviour (e.g. software stales, hardware failures, etc.) of the cloud infrastructure.

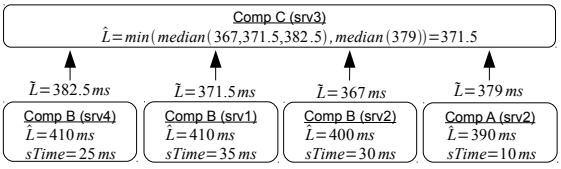

Fig. 3. Propagation of SLA from parents to children. The child *Comp C* receives 4 SLA updates from its parents : 3 from replicas of *Comp B* and 1 from *Comp A*. The new SLA of the child component is computed following equation (10).

More formally, assuming that the SLA requirement is an upper bound in the response time, the response time L_j of a component j can be calculated as follows:

$$L_j = sTime_j + max_{i \in D(j)}(L_i)\,, \tag{8}$$

where $sTime_j$ is the service time of the j component, which is the time required by a component j to process the request locally, not accounting for the response time of its dependencies $D(j)$. Therefore, the response time of a component is the sum of its service time and the response time of the slowest of the components that it depends on.

In order to meet the user requirements, each component j periodically propagates the individual suggested performance constraints (e.g. an upper bound $\widetilde{L_i}$ in the response time) to its dependencies $i \in D(j)$ according to the following formula:

$$\widetilde{L_i} = \widetilde{L_j} - \kappa \cdot sTime_j \; - \lambda \cdot prop_{ji}, \forall i \in D(j)\,, \tag{9}$$

where $prop_{ji}$ is the network delay between components j and i, while κ, λ are factors (typically 1.1) to take into account the volatility of the service time and the network delay respectively.

When a component receives individual performance constraints from dependent components, it groups together the constraints that come from replicas of a certain component. Then, in order to compute its individual SLA constraint $\hat{L}_i$, the component i chooses the minimum value among the median SLA values of each group, i.e. :

$$\hat{L}_i = min\{median(\Lambda_0), \ldots, median(\Lambda_n)\}\,, \tag{10}$$

where Λ_g is the group of the suggested performance constraints sent by the replicas of the dependent component g, while n is the number of unique dependent components (not counting the replicas). Choosing the $median()$ performance constraint instead of the $mininum()$ or the $average()$ one allows the system to be more robust against unstable hosts. Each component i should satisfy that its response time meets its individual constraint within a certain *confidence bound* d, i.e.

$$L_i \leq \hat{L}_i - d\,.$$

The selection of the global confidence bound implies a trade-off between the worst-case SLA-compliance and cost-efficiency. The proper value of d per application can be dynamically learnt by employing a tattonment process for meeting the performance constraint of the application.

V. Automatic provisioning of cloud resources

As explained in Section III, the framework takes care of balancing the load among the available cloud resources in a fair way, by the use of autonomic migration, replication and suicide of components. However, these mechanisms might not be sufficient to ensure that the end-to-end latency is acceptable for an application owner. Essentially, each component of the distributed application needs to satisfy an individual SLA. When the application load is globally balanced, if a component is not able to process the requests fast enough, it usually means that the dedicated cloud resources are too scarce to host the application and to provide acceptable performance. A component that does not comply to its SLA is allowed to dynamically ask for more resources. If the virtual machine (VM) hosting the component is able to scale up (*vertical scaling*), the framework will assign more resources to it (e.g. increasing the number of CPU cores, adding memory,

etc.) by interacting directly with the cloud infrastructure API. If the VM is already at the maximum of its capacity, the framework will start a new VM (*horizontal scaling*), with the minimum amount of dedicated resources. After a short period of time, some components will migrate or replicate to the new available VM. Essentially, after this load-balancing process, the component is probably able to meet its SLA. On the other hand, when a server agent realizes that the components, which it is responsible for, have enough resources to serve requests x times faster than required by their respective SLAs, the framework will decrease the dedicated resources of the VM. Thus, the adaptive provisioning of cloud resources is mainly driven by the capacity of the components to satisfy their SLA.

A. Adaptivity to slow servers

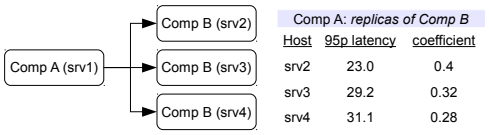

Fig. 4. *Comp A* keeps statistics about the response time of requests sent to its children locally. Based on the 95th percentile of the response time of the children, a parent computes the probability of choosing a replica of *Comp B*.

Each component keeps locally statistics about the latencies of its children. Every time a component sends a request to one of its dependencies, it stores the mean and the 95th percentile of its response time. With these statistics, the component computes a routing coefficient for every replica of a child component (i.e. a component that it depends on) in order to dynamically choose an appropriate replica. This coefficient is the probability that the child replica will be chosen for processing of subsequent requests.

Figure 4 illustrates an example where there are three replicas of the child component *Comp B*. At the beginning, each child component has a coefficient of $p = 1/R = 1/3$, where R is the number of replicas, i.e. 3 in this case. Periodically, the parent component (i.e. the dependent one) updates the coefficients based on the latency of the children, as shown in pseudo-code in the Algorithm 1. According to the algorithm, a small value δ is added to the coefficient of the fastest component, which is subtracted from the coefficient of a randomly chosen one that is slower by more than θ. δ expresses the robustness/adaptivity trade-off of the reaction to the current component performance and it is referred to as *reactivity factor*. If the replica of *Comp B* on server *srv2* is faster than the two other replicas, then, after some time, it will have a greater coefficient, e.g. 0.4, while the coefficients of components *Comp B* on servers *srv3*, *srv4* will become 0.32, 0.28 respectively. So, on the average, *Comp B* on *srv2* will receive 40% of the requests from the parent, *Comp B* on *srv3* 32% and *Comp B* on *srv4* will only get 28%, so that the latency of the overall requests is always minimized.

If one of the VMs (or the underlying physical server) is much slower than the others, then the components hosted at this slow VM will gradually receive less and less requests, and thus the framework will scale the VM down. At some point, the VM may also be completely stopped.

When a new replica of a component shows up at a server (after a *migration* or a replication), only a small coefficient δ_0 (e.g. $\delta_0 = 0.1$) is assigned to the replica, in order not to overload it until it is properly initialized. The coefficient of the other replicas of the same component is then decreased by $\delta_0/(R-1)$. When a replica of a component disappears (after a suicide or a migration), its coefficient is equally shared among the rest of the replicas of the component.

Algorithm 1 Routine for updating the forwarding coefficients of child components by a small value δ (e.g. $\delta = 0.05$)

Require: set of child components S,
array P of component coefficients,
array L of component 95th perc. of response times
Ensure: $\sum_{j=1}^{|S|} P[j] = 1$

$fastest \leftarrow \arg\min_j L[j]$
$slowComps \leftarrow \{\}$
for all $j \in S \setminus \{fastest\}$ **do**
  if $L[j] - L[fastest] > \theta$ **then**
    $slowComps \leftarrow slowComps + \{j\}$
  end if
end for
$slower \leftarrow Random(slowComps)$
$P[slower] \leftarrow P[slower] - \delta$
$P[fastest] \leftarrow P[fastest] + \delta$

VI. Evaluation

The results regarding the load-balancing, the scalability and the fault-tolerance of the approach have been thoroughly discussed in our previous work [3]. Here, we investigate the effectiveness of our approach for meeting statistical SLA performance guarantees under varying request load and software/hardware failures.

A. Experimental Setup

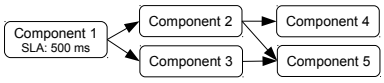

Fig. 5. Architecture of a test application composed by 5 components.

In our evaluation, we consider an application composed by 5 different components, as depicted on Figure 5. The results presented in this section refer to an application that is mostly CPU-intensive, hence $w_c >> w_m, w_n, w_d$ for every component. However, we have also conducted experiments with different types of applications (I/O intensive, CPU intensive, a mix of both, etc.) and with different component workflows (fully parallel, fully sequential, a mix of parallel/sequential with several numbers of tiers) with similar conclusions on the

effectiveness of our approach. At startup, all components are started on a single 1-core server. The minimum number of replicas per component for ensuring fault-tolerance is set to 2.

The underlying cloud infrastructure is composed by 8 8-cores servers (Intel Core i7 920 @ 2.67 GHz, 8GB Ram, Linux 2.6.32-trunk-amd64). The components interact with the cloud infrastructure through an API that allows asking for more resources (adding cores to a server, starting a new server) or less resources (remove cores from a server, stopping a server). Although each server CPU has 8 cores, we only allow a server to employ 2, 3 or 6 cores at maximum in our evaluation, in order to force the earlier provisioning of a new server. The servers reside on a 1Gbps switched Ethernet LAN.

We have set the SLA (by means of an upper bound in the response time) of the first component of the application (i.e. "Component 1" on Figure 5) to be 500 ms, while no confidence bound (i.e. $d = 0$) was considered. The parameters employed in the utility formula are $C = 110$, $k = 10000$, while $x_s^* = 25\%$.

B. Results

a) Adaptation to varying load: In this experiment, we investigate the reactivity of our framework to quickly increasing or shrinking load of application requests. The initial load is assumed to be 5 requests per second. At the 8-th minute of the experiment, we start increasing the load every minute by 5 requests per second until the total traffic reaches 60 requests per second. Then, we keep the request load constant for 15 minutes. Afterwards, we start decreasing the load every minute by 5 requests per second until the initial load is reached. We allow a maximum number of 3 cores to be employed per server.

We compare the performance (in terms of response time) of our dynamic approach with that of a static one under the same load conditions. In the static setup, the total amount of cloud resources allocated to the application remains constant and equal to 2 servers with 2 cores each. During the total time of the experiment (60 minutes), our dynamic approach has employed for the application components 4 cores on the average from the cloud. For a fair comparison regarding the performance, we also employed a total of 4 cores for the application in the static setup.

As depicted in Figure 6, our framework reacts appropriately to the increasing amount of requests by asking for more cloud resources in order to satisfy the SLA. Once the additional resources are no longer required for SLA compliance (i.e. the 95th percentile response time of every component in the server is x times faster than required), then the framework releases them for reducing the costs. Clearly, in the long run, the overall cost for the static setup would be much higher, as the user would constantly pay for 4 cores even in low load periods (where 2 cores can satisfy the SLA). Our framework uses the minimum required resources to serve the application within the SLA requirements by fully leveraging the elasticity of today's cloud infrastructures.

Also, thanks to the adaptivity of our framework, the maximum request load for the application (60 requests per second) can be sustained, while keeping the average response time under 500 ms, as shown in Figure 7. The static approach is also able to serve the maximum request load, but the average response time is greater and significantly varies with time. Also, our adaptive framework achieves much lower 95th percentile of the response time than that of the static approach, as depicted in Figure 8. The framework only reacts when the 95th percentile of the response time reaches 500 ms. Until minute 13, the static setup has more resources (i.e. 2 servers with 2 cores) compared to Scarce, and therefore performs better. After minute 20, Scarce has allocated the needed resources to the application to meet the SLA and clearly outperforms the static setup.

However, the confidence bound d should be properly selected, according to the application tolerance to the QoS violations. There is a clear *trade-off* between worst-case SLA-compliance and cost-efficiency in the selection of the confidence bound. As shown in Figure 11, if the application is assumed to be inelastic and the confidence bound is selected as $d = 60\% \cdot response\_time$, then our adaptive approach would allow almost no SLA violations, as opposed to the static setup.

As soon as the number of requests per second sent to the application increases, the service time as well as the response time of each component are impacted. Recall that each component periodically sends a suggested SLA constraint update to its child components. As the suggested SLA update (given by equation (9)) depends on the component service time, the application load has a direct effect on the derived SLA constraint of each component. This effect is depicted in Figure 10, where the SLA constraints for the components are getting stricter with the growing application load.

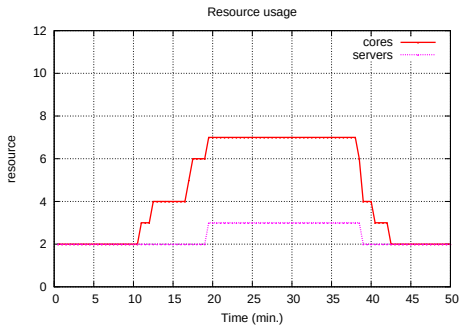

Fig. 6. Scarce : Resources used by the application over time for varying request load.

b) Adaptation to slow servers: A recurring issue with cloud infrastructures is that the user has no control over the performance of the rented resources. As a physical server is shared by several virtual machines (VM), a VM might intensively use the I/O subsystem, and may therefore degrade the performance for the other VMs collocated on the same physical server. In addition, a physical server, which has an unreliable hardware component or a non-optimized operating

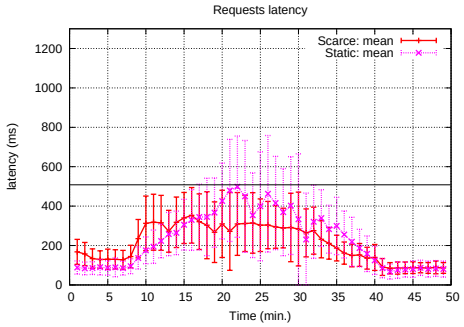

Fig. 7. Mean response times of the application (SLA : 500 ms) as perceived by remote clients under the adaptive approach ("Scarce") and the static setup.

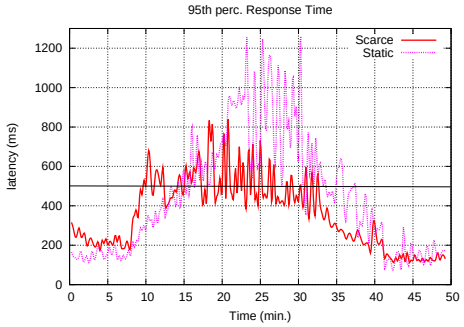

Fig. 8. 95th percentile response times of the application (SLA : 500 ms) as perceived by remote clients under Scarce and the static setup.

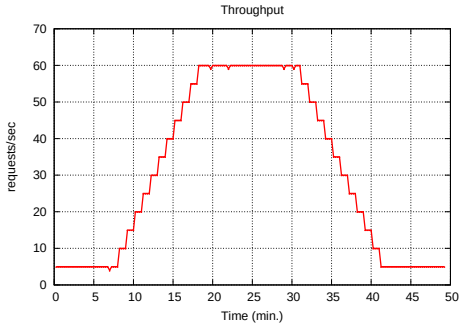

Fig. 9. Throughput of the application during the varying load experiments.

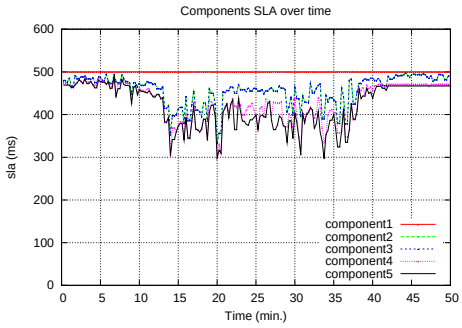

Fig. 10. Computed SLA constraints of the components hosted at a server.

system setup, will also have poorer performance, and will therefore negatively impact the application end-to-end latency. Our framework is able to detect slower servers and to discard them transparently to the application. In this case, the maximum number of cores allowed to be employed per server is 2.

Here, the request load is assumed to be 25 requests per second to the entry component of the application. After 4 minutes, one of the server starts to be slower and every component hosted at it serves requests with a delay of 200ms. All component that are not hosted at the "wonky" server detects this slowdown, adapts the coefficients of their dependencies accordingly and the traffic is slowly redirected to faster components. Two minutes later (at minute 6), a new server is started, because some components are no more able to honor their SLA due to the redirected traffic. At minute 11, the wonky server is removed from the cloud by the framework as it receives only a negligible amount of requests. As shown in Figure 12, our approach quickly adapts to the situation and renders the response time of the application again compliant to the SLA. The dynamic resource allocation for the application in this scenario is depicted in Figure 13.

c) Scalability and stability: In this experiment, the rate of requests for the application increases every minute by 5 requests/second until reaching the load of 150 requests/second. Each server is allowed to employ up to 6 cores. In Figure 14, the 95th percentile of the response time quickly stabilizes close to the SLA constraint after the request rate stops increasing and becomes constant. Finally, as shown in Figure 16, the global amount of physical resources employed follows the trend of the request load (depicted in Figure 15).

C. Discussion

Simple approaches proposed by cloud providers such as Amazon allow a customer to set rules to automatically add or remove resources when a metric (e.g. CPU usage) goes above or below a threshold H and L respectively. However, even if the CPU usage higher than H, the performance constraint per component may be met, or vice versa. The employment of fine-grained metrics, such as 95th percentile of response time per component (server metrics are not enough), is required to use the minimum amount of resources for a given SLA. Moreover, Scarce efficiently multiplexes components at servers based on component migration and replication.

VII. Related Work

There is significant related work in the area of economic approaches for resource management in distributed computing. In [6], an approach is proposed for the utilization of idle computational resources in an heterogeneous cluster. Agents assign computational tasks to servers, given the budget constrain for each task, and compete for CPU time in sealed-bid second-price auction held by the latter. In a similar setting, Popcorn approach [7] employs a first-price sealed-bid auction model.

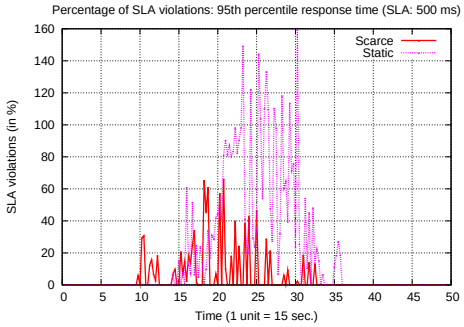

Fig. 11. Percentage of SLA violations from Scarce and the static approach when the 95th percentile response time should stay under 500 ms.

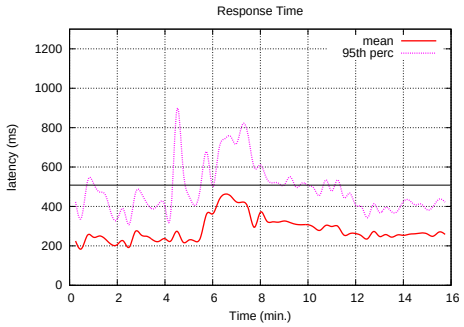

Fig. 12. Mean and 95th percentile response times of the application (SLA : 500 ms) as perceived by remote clients in case of a "wonky" server.

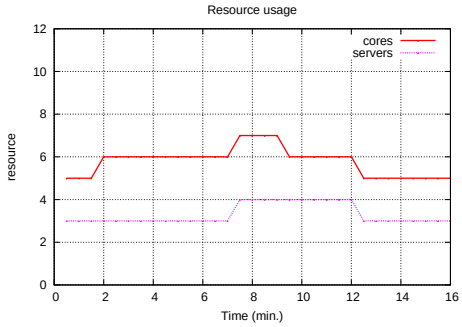

Fig. 13. Resources used by the application over time in case of a "wonky" server.

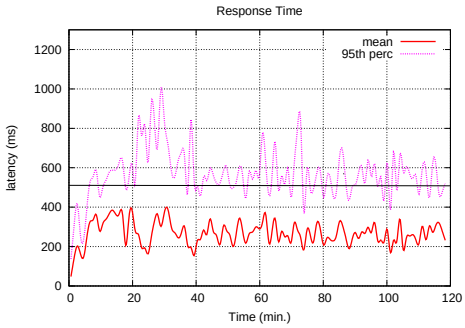

Fig. 14. Mean and 95th percentile response times of the application (SLA: 500ms) as perceived by remote clients in the scalability experiment.

Cougaar distributed multi-agent system [8] has an adaptivity engine which monitors load by employing periodic "health-check" messages. An elected agent operates as load balancer and determines the appropriate node for each agent that must be relocated based on runtime performance metrics, e.g. message traffic and memory consumption. Also, a coordinator component determines potential failure of agents and restarts them. However, cost-effectiveness is not among the objectives of Cougaar, and moreover our approach is more lightweight in terms of communication overhead.

In [9], a virtual currency (called Egg) is used for expressing a user's willingness to pay as well as a provider's bid for a accepting the job, and finally is given to the winning provider as compensation for job execution. The central Egg entity informs all candidate providers about the new job and acquires responses (opportunity cost estimations for accepting the job). However, the approach in [9] is centralized and it does not provide availability guarantees.

In [10], applications trade computing capacity in a free market, which is centrally hosted, and are then automatically activated in virtual machines on the traded nodes on-call of traffic spikes. The applications are responsible for declaring their required number of nodes at each round based on usage statistics and allocate their statically guaranteed resources or more based on their willingness to pay and the equilibrium price; this is the highest price at which the demand satu-rates the cluster capacity. However, [10] does not deal with availability guarantees, as opposed to our approach. Also, our approach accommodates traffic spikes in a prioritized way per application without requiring the determination of the equilibrium price.

Pautasso *et al.* propose in [11] an autonomic controller for the JOpera distributed service composition engine over a cluster. The autonomic controller starts and stops navigation (i.e. scheduler) and dispatcher (i.e. execution and composition) threads based on several load-balancing policies that depend on the size of their respective processing queues. However, proper thread placement in the cluster and communication overhead among threads are not considered in [11].

Also, SLA provisioning for web services [12] or multi-tier web applications [18], [19] has been studied. [18], [19] adapt the behavior of the underlying resources based on the capacity of an application to honor an SLA. However, monitoring of SLA compliance in [12], [18], [19] may require the involvement of third-parties or centralized services. A decentralized approach for SLA provisioning in grids based on service migration between containers is proposed in [13]. However, this approach does not dynamically vary the total amount of allocated resources to the application according to the load and the potential software/hardware failures in the cloud, as opposed to our work.

A bio-networking approach was proposed in [14], where services are provided by autonomous agents that implement

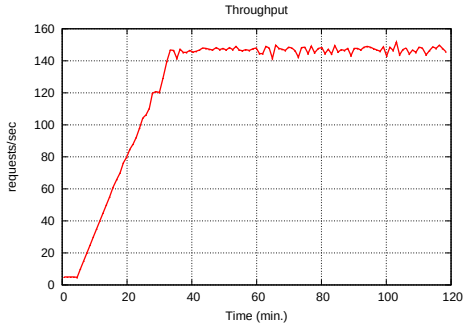

Fig. 15. Scarce : Throughput of the application during the scalability experiment.

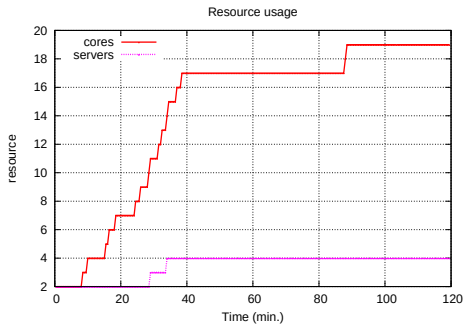

Fig. 16. Resources used by the application over time during the scalability experiment.

basic biological behaviors of swarms of bees and ant colonies such as replication, migration, or death. To survive in the network environment, an agent obtains "energy" by providing a service to the users. However, [14] does not consider the problem of satisfying performance constraints.

Moreover, several implementation frameworks exist towards reliable SOA-based applications: [15] is a mechanism for specifying fault tolerant web service compositions, [16] is a virtual communication layer for transparent service replication, and [17] is a framework for the active replication of services across sites. These frameworks do not consider dynamic adaptation to changing conditions, such as load spikes, or do not provide guarantees for geographical diversity of replicas.

VIII. Conclusions

We proposed an economic, lightweight approach for dynamic accommodation of load spikes and failures for composite web services deployed in clouds, so as to satisfy performance and availability guarantees. We derive performance constraints per component and we scale up existing VMs or create new whenever they are not met. Application components act as individual optimizers and autonomously replicate, migrate across VMs or terminate based on their economic fitness. Their resource inter-dependencies are implicitly taken into account by means of server rent prices. The requests are routed across components based on their respective prior performance. Our approach also detects unstable cloud resources and reacts accordingly, so as to minimize the end-to-end application response time. As a future work, we intend to explore our economic paradigm for autonomic resource management in the context of multiple competitive or cooperative cloud providers.

Acknowledgment

This work was partially funded by the EU project HYDROSYS (224416, DG-INFSO).

References

[1] J. Dejun, G. Pierre and C. H. Chi, "EC2 Performance Analysis for Resource Provisioning of Service-Oriented Applications," in *Proc. of NFPSLAM-SOC*, Stockholm, Sweden, 2009.

[2] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc. of 5th USENIX Conference on File and Storage Technologies (FAST '07)*, San Jose, CA, USA, February 2007.

[3] N. Bonvin, T. G. Papaioannou, and K. Aberer, "An economic approach for scalable and highly-available distributed applications," in *Proc. of the CLOUD*, Miami, FL, USA, 2010.

[4] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, pp. 133–169, 1998.

[5] N. Bonvin, T. G. Papaioannou, and K. Aberer, "A self-organized, fault-tolerant and scalable replication scheme for cloud storage," in *Proc. of the SOCC*, Indianapolis, USA, 2010.

[6] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta, "Spawn: A distributed computational economy," *IEEE Transactions on Software Engineering*, vol. 18, pp. 103–117, 1992.

[7] O. Regev and N. Nisan, "The popcorn market. online markets for computational resources," *Decision Support Systems*, vol. 28, no. 1-2, pp. 177 – 189, 2000.

[8] A. Helsinger and T. Wright, "Cougaar: A robust configurable multi agent platform," in *Proc. of the IEEE Aerospace Conference*, 2005.

[9] J. Brunelle, P. Hurst, J. Huth, L. Kang, C. Ng, D. C. Parkes, M. Seltzer, J. Shank, and S. Youssef, "Egg: an extensible and economics-inspired open grid computing platform," in *Proc. of the GECON*, Singapore, May 2006.

[10] J. Norris, K. Coleman, A. Fox, and G. Candea, "Oncall: Defeating spikes with a free-market application cluster," in *Proc. of the International Conference on Autonomic Computing*, New York, NY, USA, May 2004.

[11] C. Pautasso, T. Heinis, and G. Alonso, "Autonomic resource provisioning for software business processes," *Information and Software Technology*, vol. 49, pp. 65–80, 2007.

[12] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef, "Web services on demand: Wsla-driven automated management," *IBM Syst. J.*, vol. 43, no. 1, pp. 136–158, 2004.

[13] C. Reich, K. Bubendorfer, M. Banholzer, and R. Buyya. "A SLA-Oriented Management of Containers for Hosting Stateful Web Services". In *Proc. of the IEEE Conference on e-Science and Grid Computing*, Washington, DC, USA, 2007.

[14] M. Wang and T. Suda, "The bio-networking architecture: a biologically inspired approach to the design of scalable, adaptive, and survivable/available network applications," in *Proc. of the IEEE Symposium on Applications and the Internet*, 2001.

[15] N. Laranjeiro and M. Vieira, "Towards fault tolerance in web services compositions," in *Proc. of the workshop on engineering fault tolerant systems*, New York, NY, USA, 2007.

[16] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He, "Transparent symmetric active/active replication for service-level high availability," in *Proc. of the CCGrid*, 2007.

[17] J. Salas, F. Perez-Sorrosal, n.-M. M. Pati and R. Jiménez-Peris, "Ws-replication: a framework for highly available web services," in *Proc. of the WWW*, New York, NY, USA, 2006.

[18] W. Iqbal, M. N. Dailey and D. Carrera, "SLA-Driven Dynamic Resource Management for Multi-tier Web Applications in a Cloud," in *Proc. of the CCGrid*, Melbourne, Australia, 2010.

[19] G. Lodi, F. Panzieri, D. Rossi and E. Turrini, "SLA-Driven Clustering of QoS-Aware Application Servers," in *IEEE Trans. Softw. Eng*, March 2007.