# An economic approach for scalable and highly-available distributed 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*—Service-oriented architecture (SOA) paradigm for orchestrating large-scale distributed applications offers significant cost savings by reusing existing services. However, the high irregularity of client requests and the distributed nature of the approach may deteriorate service response time and availability. Static replication of components in datacenters for accommodating load spikes requires proper resource planning and underutilizes the cloud infrastructure. Moreover, no service availability guarantees are offered in case of datacenter failures. In this paper, we propose a cost-efficient approach for dynamic and geographically-diverse replication of components in a cloud computing infrastructure that effectively adapts to load variations and offers service availability guarantees. In our virtual economy, components rent server resources and replicate, migrate or delete themselves according to self-optimizing strategies. We experimentally prove that such an approach outperforms in response time even full replication of the components in all servers, while offering service availability guarantees under failures.**

***Keywords*-component; net benefit; replication; agent; web service**

## I. Introduction

Cloud computing is deemed to replace high capital expenses for infrastructure with lower operational ones for renting cloud resources on demand by the application providers. However, with static resource allocation, 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 clouds scale up, hardware failures of any type are unavoidable.

A successful online application should be able to handle traffic spikes and flash crowds efficiently. Moreover, the service provided by the application needs to be resilient to all kinds of failures (e.g. software stales, hardware, rack or even datacenter failures, etc.). A naive solution against load variations would be static over-provisioning of resources, which would result into resource underutilization for most of the time. Resource redundancy should be employed to increase service reliability and availability, yet in a cost-effective way. Most importantly, as the size of the cloud increases its administrative overhead becomes unmanageable. The cloud resources for an application should be self-managed and adaptive to load variations or failures.

In this paper, we propose a middleware ("Scattered Autonomic Resources", referred to as *Scarce*) for supple sharing to avoid stranded and underutilized computational resources that dynamically adapts to changing conditions, such as failures or load variations. 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.

Our approach combines the following unique characteristics:

- Adaptive component replication for accommodating load variations.
- Geographically-diverse placement of clone component instances.
- Cost-effective placement of service components for supple load balancing.
- Decentralized self-management of the cloud resources for the application.

Having implemented a full prototype of our approach, we experimentally prove that it effectively accommodates load spikes, it provides a dynamic geographical replica placement without thrashing and cost-effectively utilizes the cloud resources. Specifically, we found that our approach offers lower response time even than full replication of the service components to all servers.

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. In Section IV, we present our experimental results. In Section V, we overview the related work and, finally in Section VI, we conclude our work.

## II. Motivation - running example

Building an application that both provides robust guarantees against failures (hardware, network, etc.) and handles dynamically load spikes is a non-trivial task. As a running example, we have developed a simple web application for selling e-tickets (print@home) composed by 4 independent components :

- A web front-end, which is the entry point of the application and serves the HTML pages to the end user.
- A user manager for managing the profiles of the customers. The profiles are stored in an highly scalable, eventually consistent, distributed, structured key-value store [1].
- A ticket manager for managing the amount of available tickets of an event. This component uses a relational database management system (MySQL).
- An e-ticket generator that produces e-tickets in PDF format (print@home).

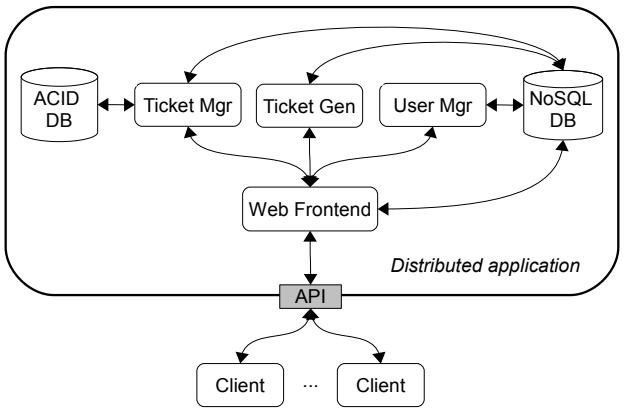


Figure 1. A distributed application using different components.

Each component can be regarded as a stateless, standalone and self-contained web service. Figure 1 depicts the application architecture. A token (or a session ID) is assigned to each customer's browser by the web front-end and is passed to each component along with the requests. This token is used as a key in the key-value database to store the details of the client's shopping cart, such as the number of tickets ordered. Note that even if the application uses the concept of sessions, the components themselves are stateless (i.e. they do not need to keep an internal state between two requests).

This application is highly sensitive to traffic spikes, when, for example, tickets for a concert of a famous band are sold. If the spike is foreseeable, one wants to be able to add spare servers that will be used transparently by the application for a short period of time, without having to reconfigure the application. After this period, the servers have to be removed transparently to the end users. As this application is business-critical, it needs to be deployed on different geographical regions, hence on different datacenters.

## III. Scarce: the quest of autonomic applications

### A. The approach

We consider applications formed by many independent and stateless components that interact together to provide a service to the end user, as in Service Oriented Architecture (SOA). A component is self-managing, self-healing and is hosted by a server, which is in turn allowed to host many different components. A component can stop, migrate or replicate to a new server according to its load or availability as 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, 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 retrieve the current routing table from the bootstrap agent. A routing table consists 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 geographical granularity could also be considered. 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, each server keeps locally a mapping between components and servers. It is maintained by a gossiping algorithm

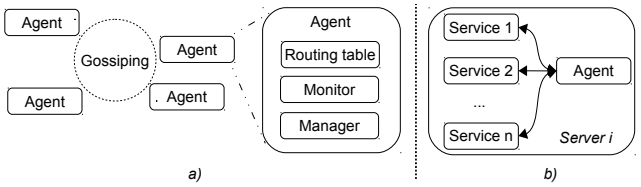


Figure 2. *a)* Agents communicate using a gossiping protocol *b)* A server hosts many services and an agent.

(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 on their respective server. Contrary to usual web services architectures, there is no central repository [such as a UDDI registry (uddi.xml.org)] for locating a service, but each agent maintains its own local registry (i.e. the routing table), as shown in Table I.

Table I
THE LOCAL ROUTING TABLE.

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

A component may be hosted by several servers, therefore we consider 4 different policies that a server $s$ may use for choosing the replica of a component:

- a *proximity-based* policy: thanks to the labels attached to each server, the geographically nearest replica is chosen;
- a *rent-based* policy: the least loaded server is chosen; this decision is based on the rent price of the servers.
- a *random-based* policy: a random replica is chosen.
- a *net benefit-based* policy: the geographically closest and least loaded replica. For every replica of the component residing at server $j$, we compute a weight:

$$w_j = \frac{\epsilon + proximity(s,j) - rent_j}{\sum_{i \in replicas} \epsilon + proximity(s,i) - rent_i}\ , \quad (1)$$

where $0 < \epsilon << 1$ is a very small positive. This weight represents the probability that the replica $j$ will be chosen.

### 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 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* $r$ 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 at an epoch. 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 [2] 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 on a server $s$ are directly related to the economic viability or $balance$ of the component, which is given by:

$$balance_c = utility_c - rent_s \quad (2)$$

The utility of a component corresponds to the value that it creates for the applications and we assume it as an increasing function of the server resources that it utilizes:

$$utility_c = priority_c * \frac{{usage_c}^2}{usage_{\ threshold}}\ , \quad (3)$$

where $usage_c$ is a factor computed using the utilization of the server resources by the component $c$. $usage_{\ threshold}$ is a certain threshold that determines when a component should be considered "fit enough" in order to replicate (currently, this is set to $25\%$ of server usage). Also, as some components may be more business critical than others (e.g. a billing component), a priority $priority_c$ can be assigned to them. The virtual rent paid by the component $c$ to the server $s$ is given by:

$$rent_s = conf_s * usage_s\ , \quad (4)$$

where $conf_s$ is a subjective estimation of the server quality and reliability based on technical factors (hardware quality, datacenter connectivity, redundancy, ...) as well as non-technicals ones (e.g. political and economical stability of the country hosting the server, ...). Also, $usage_s$ is a factor that expresses the resource utilization of the server. Note that other utility and rent function could be used as long as

they were both increasing to the resource usage and result in comparable values.

Based on the $balance$, 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 (6)). 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 $usage_s$ of the current server $s$ is above a soft limit.
- *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:

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

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 (6)), 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. The availability of a component should be always kept above a required minimum level $th$. As estimating the probability of each server to fail necessitates access to an large set of historical data and private information of the server, we express the potential availability of a service by means of the geographical diversity of the servers that host its replicas, similarly to [3]. Therefore, the availability of a service $i$ is defined as the sum of *diversity* of each distinct pair of servers, i.e.:

$$avail_i = \sum_{i=0}^{|S_i|} \sum_{j=i+1}^{|S_i|} conf_i \cdot conf_j \cdot diversity(s_i, s_j) \quad (5)$$

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 pairs. This distance can be represented as a $n$- bit number, having each bit corresponding the 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 have the same location, their corresponding *proximity* bit is set to 1, otherwise to 0. Once a bit has been set to 0, all less significant bits are also set to 0. For example, two servers belonging to the same data center but located in different rooms cannot be in the same rack, thereby all bits after the third bit (data center) have to be 0. The proximity number would then look like this:

| cont | coun | city | datac | room | rack | serv |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

A binary "NOT" operation is then applied to the proximity to get the diversity value:

$$\overline{1110000} = 0001111 = 15(decimal)$$

The diversity values of server pairs are summed up, because having more replicas in distinct servers located even in the same location always results in increased availability. A component knows the locations of its replicas by the local routing table at the server where it is hosted.

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.

$$\sum_{k=1}^{|S_i|} g_j \cdot conf_j \cdot diversity(s_k, s_j) - rent_{\ j}\ , \quad (6)$$

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.* [3]). 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 (6) and randomly choose the target for replication among the top-k ones. This is done in order to avoid overloading the same destination server at an epoch, which would result to thrashing. Thus, the availability tends to be increased as much as possible at the minimum cost, while the network latency for the query reply also decreases. The

availability level $th$ allows a fine-grained control over the replication process.

Note that the same approach according to (6) is used for choosing the candidate server for component migration.

## IV. Evaluation

### A. Experimental Setup

We employ two different testbed settings: a *single-application* setup consisting of 7 servers and a *multi-application* setup consisting of 15 servers. In the former setup, the cloud resources serve 1 application and in the latter one 3 applications. The hardware specification of each server is Intel Core i7 920 @ 2.67 GHz, 8GB Ram, Linux 2.6.32-trunk-amd64. We run two databases (MySQL 5.1 and Cassandra 0.5.0) as well as one generator of client requests for each application (FunkLoad 1.10, http://funkload.nuxeo.org/) on their own dedicated servers. Thus, the cloud consists of 4 and 10 servers in the *single-application* and the *multi-application* setup respectively. We assume that the components of the application may require 1 up to all servers in the cloud.

We simulate the behavior of a typical user of the e-ticket application of Section II by performing the following actions: *1)* request the main page that contains the list of entertainment events; *2)* request the details of an event $A$; *3)* request the details of an event $B$; *4)* request again the details of the event $A$; *5)* login into the application and view user account; *6* update some personal information; *7)* buy a ticket for the event $A$; *8)* download the corresponding ticket in PDF. A client continuously performs this list of actions over a period of 1 minute. An epoch is set to 15 seconds and an agent sends gossip messages every 5 seconds. Moreover, the default routing policy is the random-based policy.

We consider two different placements of the components:

- A *static* approach where each component is assigned to a server by the system administrator.
- A *dynamic* approach where all components are started on a single server and dynamically migrate / replicate / stop according to the load or the hardware failures.

### B. Results

*Dynamic vs static replica placement:* First, we employ the *single-application* experimental setup to compare our approach with *static* placements of the components, where we consider two cases: *i)* each different component is hosted at a different dedicated server; *ii)* full replication, where every component is hosted at every server. The response time of the 95% percentile of the requests is depicted in Figure 3. In the static placement (i), where a component runs on its own server, the response time is lower bounded by that of the slowest component (in our case, the service for generating PDF tickets). Thus, the response time increases exponentially when the server hosting this component is overloaded. In the case of full replication [static placement (ii)], the requests are balanced among all servers, keeping the latency relatively low, even when the amount of concurrent users is significant. In the dynamic placement approach, all components are hosted at a single server at startup: then, when the load increases, a busy component is allowed to replicate, and unpopular components may replicate to a less busy server. Our economic approach achieves better performance than full replication, because the total amount of CPU available in the cloud is used in an adaptive manner by the components: processing intensive (or "heavy") components migrate to the least loaded servers and heavily-used components are assigned more resources than others. Therefore, the cloud resources are shared according to the processing needs of components and no cloud resources are wasted by over-provisioning.

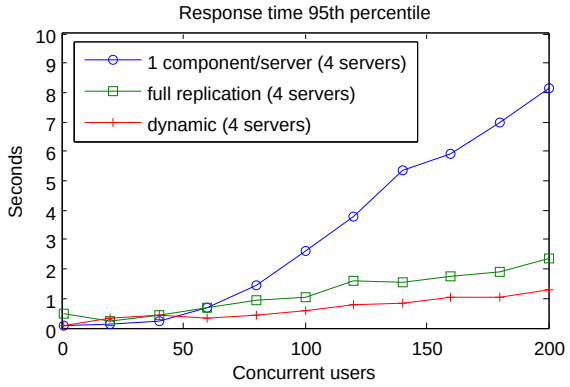


Figure 3. Response time of different placement approaches.

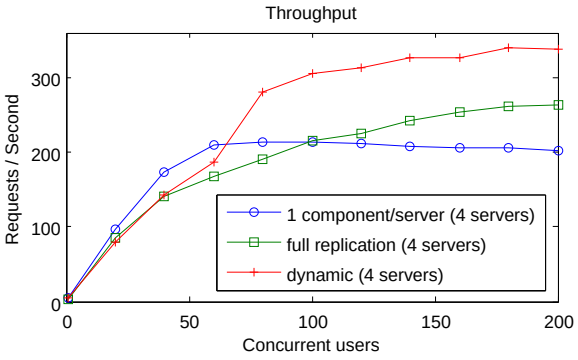


Figure 4. Throughput compared with different placement approaches.

Also, as the cloud resources are properly utilized by the economic approach, the application throughput (i.e. the number of request served per second) that it achieves outperforms static placements, as depicted in Figure 4.

*Scalability:* Having established the effectiveness of our dynamic component placement approach over static ones, we next investigate the resulting scalability in the cloud in

the *multi-application* experimental setup. We assume that all 10 servers reside at 1 datacenter. We gradually increase the number of concurrent users from 150 to 1500. The service requests are equally shared among applications and randomly routed among the replicas of a component. As depicted in Figure 5, the response time per application increases linearly to the load and the resources of the cloud are shared among the components of different applications in a fair way. The experiment is repeated 5 times and mean values and confidence intervals are presented.

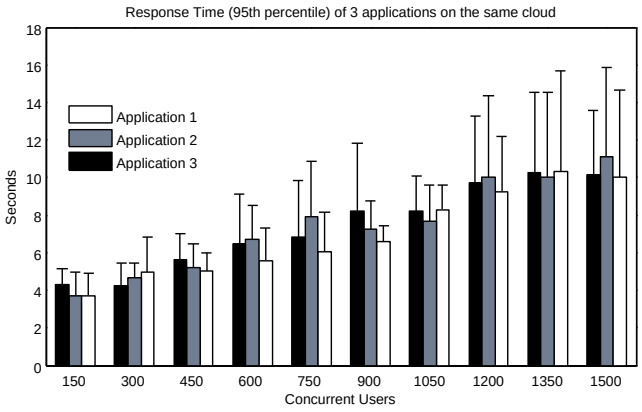


Figure 5. 95% percentile response times for 3 different applications as load increases.

*High-availability:* Next, we show that our approach is highly resilient to hardware failures. To this end, we employ the *single-application* experimental setup. We assume that each component has 2 replicas that reside at separate servers. 10 concurrent clients continuously send requests for 1 minute. After 30 seconds, one random server between those hosting the replicas of a component fails. As illustrated in Figure 6, the percentage of requests that were not satisfied is 0.34% in this case. The failures correspond to requests already sent to the failed replica. If both servers hosting the replicas of a component fail at the same time, 3.58% of the requests are lost. Due to the gossiping protocol, the remaining servers quickly detect the failure, start the failed components locally and broadcast the updated routing entries for them.

*Adaptation to new cloud resources:* In this experiment, we investigate the adaptability of our dynamic placement approach when new resources are added to the cloud. We employ the *single-application* experimental setup, but the number of available servers in the cloud ranges from 1 to 10. The application is concurrently accessed by 1500 users, while service requests are equally shared among the replicated instances of a particular component. As observed by Figure 7, our dynamic placement approach fully exploits the new server resources that are added to the cloud, as the application response time decreases, while the service

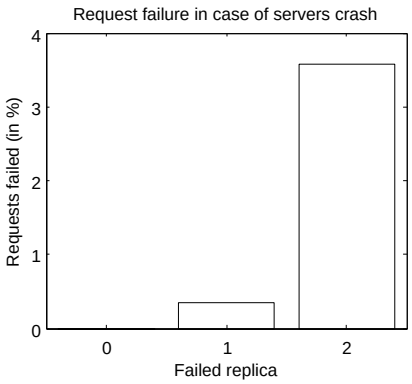


Figure 6. Request failure percentages when 0, 1 and 2 replicas (out of 2) crash.

throughput increases. The experiment is repeated 5 times and mean values and confidence intervals are presented.

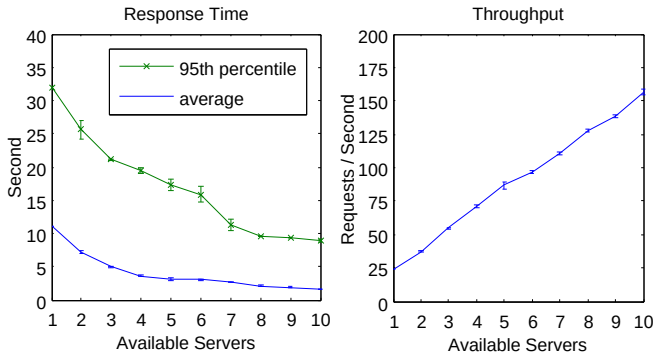


Figure 7. Response time (left) and throughput (right), when new cloud resources are added.

*Evaluation of routing policies:* When multiple instances of a particular component are available, the requests have to be split among the several instances of the requested component to efficiently balance the load. One approach could be that the requests are equally shared (i.e. at random) among the instances of the requested component. However, this approach does not take into account neither the network delay among the service hosting the requesting and the requested components, nor the load at the servers hosting the instances of the requested components. To this end, we experimentally investigate the three other approaches that were described in Section III-C.

In this case, we employ the *single-application* setup, but the 4 servers of the cloud are located in 2 datacenters (2 servers per datacenter). The round-trip time between the datacenters is 50 ms and the minimum availability (i.e. number of replica per component) is set to 2.

First, the application requests are evenly split between the two datacenters (50% to datacenter 1, 50% to datacenter 2). As depicted in Figure 8(left), the proximity-based routing

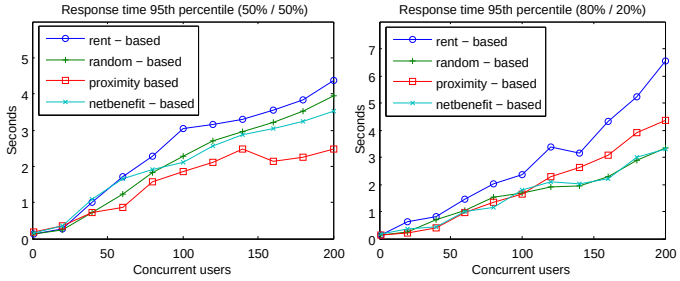


Figure 8. Response time when requests between datacenters are: i) 50%-50% (left), ii) 80%-20% (right).

policy achieves the lowest response time, as it saves the delay for transmitting requests between the datacenters. However, it may result in an unbalanced server usage if the traffic is not balanced between the datacenters, as illustrated in Figure 8(right). The rent-based policy (as well as the net benefit one) may suffer from the fact that the rents of servers may not always be up-to-date due to the gossiping algorithm. The net benefit routing policy performs at least as good as the random one both for balanced and for unbalanced load between datacenters, as depicted in Figure 8, yet at a higher computational cost at runtime.

## V. Related Work

There is significant related work in the area of economic approaches for distributed computing. In [4], an approach is proposed for the utilization of idle computational resources in a 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 [5] employs a first-price sealed-bid auction model.

Cougaar distributed multi-agent system [6] 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 among the objectives of Cougaar, and moreover our approach is more lightweight in terms of communication overhead.

In [7], 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. Providers estimate their opportunity cost for accepting a job and regularly announce a unit price table to a central entity for a specific period. The central Egg entity informs all candidate providers about the new job and acquires responses (cost estimations). However, the approach in [7] is centralized and it does not provide availability guarantees.

In [8], 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 saturates the cluster capacity. However, [8] 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 [9] 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. The autonomic component also has self-healing capabilities. However, proper thread placement in the cluster and communication overhead among threads are not considered in [9].

Also, in [10], SLA agreements for a specific QoS level for web services are established. However, monitoring of SLA compliance may require the involvement of third-parties or centralized services. A bio-networking approach was proposed in [11], where services are provided by autonomous agents that implement 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.

Moreover, several implementation frameworks exist to build reliable SOA-based applications: [12] is a mechanism for specifying fault tolerant web Service compositions, [13] is a virtual communication layer for transparent service replication, and [14] 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 about geographical diversity of replicas.

## VI. Conclusions

We proposed an economic, lightweight approach for dynamic accommodation of load spikes for composite web services deployed in clouds. Application components act as individual optimizers and autonomously replicate, migrate or stop based on their economic fitness. Inter-dependencies (traffic and workflow) among components, their processing overhead and server capabilities are implicitly taken into account by means of server rent prices. Our approach also offers high availability guarantees by maintaining a certain number of the various components in geographically diverse locations. As a future work, we intend to explore our

economic paradigm for the self-tuning in the cloud of service components with heavy data dependencies.

## ACKNOWLEDGMENT

This work was partially funded by the EU projects HYDROSYS (224416, DG-INFSO) and OKKAM (215032, ICT).

## REFERENCES

[1] "The apache cassandra project," http://cassandra.apache.org/.

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

[3] N. Bonvin, T. G. Papaioannou, and K. Aberer, "Cost-efficient and differentiated data availability guarantees in data clouds," in *Proc. of the ICDE*, Long Beach, CA, USA, 2010.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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, pp. 357–366.