# Multi-objective Meta-heuristics for Scheduling Applications with High Availability Requirements and Cost Constraints in Multi-Cloud Environments

Marc E. Frincu
*Research Institute e-Austria and West University of Timisoara*
*Timisoara, Romania*
*Email: mfrincu@info.uvt.ro*

Ciprian Craciun
*Research Institute e-Austria*
*Timisoara, Romania*
*Email: ccraciun@info.uvt.ro*

***Abstract*—As the popularity of cloud computing increases, more and more applications are migrated onto them. Web 2.0 applications are the most common example of such applications. These applications require to scale, be highly available, fault tolerant and able to run uninterrupted for long periods of time (or even indefinitely). Moreover as new cloud providers appear there is a natural tendency towards choosing the best provider or a combination of them for deploying the application. Thus multi-cloud scenarios emerge from this situation. However, as multi-cloud resource provisioning is both complex and costly, the choice of which resources to lend and how to allocate them to application components needs to rely on efficient strategies. These need to take into account many factors including deployment and run-time cost, resource load, and application availability in case of failures. For this aim multi-objective scheduling algorithms seem an appropriate choice. This paper presents an algorithm which tries to achieve application high-availability and fault-tolerance while reducing the application cost and keeping the resource load maximized. The proposed algorithm is compared with a classic Round Robin strategy – used by many commercial clouds – and the obtained results prove the efficiency of our solution.**

***Keywords*-cloud scheduling; multi-objective scheduling; meta-heuristics**

## I. Introduction

As cloud computing is becoming more and more popular with the constantly increasing need for on-demand and large-scale computing, public cloud providers such as Amazon, Google, Microsoft or Rackspace offer access to their reliable infrastructure either by means of specialized platforms (e.g., Google App Engine, Microsoft Azure, etc.) or directly to virtualized resources by means of infrastructure level APIs (e.g., Amazon EC2 / S3, RackSpace Cloud, etc.). But despite their efforts unforeseen incidents (e.g., natural disasters or software glitches) can cause entire clouds to fail. This impacts heavily businesses which rely entirely (or largely) on clouds to host their online applications. Public clouds can also induce operational costs on companies that want to externalize their application hosting. In order to cope with this problem some of them are required to invest in private cloud solutions (e.g., Eucalyptus, OpenStack). However businesses that require large computational and storage resources will eventually need to invest in large data-centres which might prove to be even more costly than externalizing the resource provisioning.

Nevertheless having a single (public or private) cloud still implies unforeseen incidents affecting reliability. One natural (and not necessarily the simplest) solution to the single cloud situation is to use several clouds as basis for running the application. By taking this approach the application can span various clouds – although cloud providers have different costs, storage options, Service Level Agreements (SLAs), etc. For instance an application requiring large horizontal scaling could fit in either the Amazon or Rackspace scheme and the choice between the two would involve the mix of various offers they advertise. Also having multiple clouds allows data to be safely stored and retrieved by using replication techniques.

In spite of the advantages of adopting multi-cloud solutions, current applications are not yet able to easily cope with them. Amazon for instance has its data-centres spread across several regions but no mechanism for automatic application scaling, load distribution or even replication across different regions. In addition clients need to choose up-front the region where to execute their application not knowing which one is the most suited for their purposes.

Therefore multi-cloud platforms – provided that there exists a functional solution – need to fulfil at least three requirements: *minimizing cost* while providing *high availability* through application component replication and *scalability* for the hosted applications. Ideally these requirements should be dealt with automatically by the platform which also needs to ensure that the node load is optimized in order to prevent over provisioning or SLA violations. Application components are co-located and mapped onto existing nodes and are independently scaled according to the node load by mapping new component instances onto already existing nodes – based on several optimization criteria – or on newly created nodes. Simultaneously this process needs to consider the overall application (fixed and recurrent) cost at each decision point. Hence *scalability*, *cost*, *load* and *availability* become essential in (multi-)cloud environments and finding a good balance between them is essential to any platform trying to bridge the gap between clouds and applications.

978-0-7695-4592-9/11 $26.00 © 2011 IEEE
DOI 10.1109/UCC.2011.43

However as it will be shown in Sect. IV there is currently no resource allocation solution that integrates all the previous requirements. Hence in this paper we provide a solution to the problem of efficiently mapping application component instances on cloud resources by proposing a multi-objective Scheduling Algorithm (SA) that aims at: (1) maximizing resource usage while at the same time (2) minimizing application run-time cost through optimizations in the component-to-node mappings by rescheduling when the node load is either too high or too low; (3) maximizing application availability and fault-tolerance by spreading its component instances evenly on allocated nodes. All these goals need to be taken into consideration in a multi-cloud environment where each provider offers different types of resources, with different costs and SLAs. We also consider the problem of application component scalability and resource provisioning to be a-priori solved or at least independent of the proposed algorithm.

The rest of the paper is structured as follows: Section II describes the proposed algorithm by starting with the general application model (cf. Sects. II-A and II-B) and continuing with the description of the scheduling heuristics (cf. Sect. II-C). Section III describes two experiments based on a Web 2.0 application. A brief overview of the state of the art is given in Sect. IV. Emphasis is also put on the main differences from our work. Finally the conclusions of the work are outlined in Sect. V.

## II. Algorithm Overview

### A. Application Architecture

Our targeted applications are mainly Web 2.0 ones, whose architecture is composed of *components* (e.g., web servers, databases and backend services, etc.) interconnected through *connectors* (e.g., message queues, sockets, etc.) running on (or consuming) cloud resources (e.g., virtual machines and block devices, cloud file-systems or databases, etc.) – such an architecture (abbreviated as *C/Cs*) is best described in [1] and [2]. Section III-A presents a concrete example of such an application. We have considered them as a good candidates as they are characterised by long running times (almost all run indefinitely) and high variation in their user access patterns. The latter depends on (1) visitor loyalty which forms a constant (non-negligible) background load and (2) occasional events that trigger usage spikes by capturing the attention of a high number of users for short (to medium) periods of time. These spikes can be either *predictable* – in case the triggering event is previously known, (e.g., sport events, etc.) – or most likely *unpredictable* – in case of randomly occurring events (e.g., media hype, disasters, etc.)

The previously mentioned characteristics make Web 2.0 applications ideal for studying the scalability efficiency offered by cloud SAs. The main reason is that the variation in user access patterns generates unpredictable dynamic load patterns, both on resources and on C/Cs, which leads to the need of easily and promptly scaling up and down the application C/Cs (i.e., elasticity) at all layers as the request rate varies over time.

A good example of component overloading is described by a backend service whose number of busy worker threads exceeds a maximum allowed threshold – which leads to a decrease in overall throughput. As a countermeasure another instance capable of dealing with the extra requests needs to be created. Similarly C/Cs that are underused should be removed; however removing them does come with risks if assuming an unpredictable environment – as a component removed at time *t* could be reactivated at time *t+1* and thus an overhead cost would unnecessarily be inflicted.

Moreover C/Cs scaling could eventually lead to node overloading. This usually happens when the memory, CPU or bandwidth of a certain node reach a predefined threshold which indicates that the performance of the hosted application will start to degrade. However an efficient SA needs to consider other existing nodes for possible relocations before deciding to request the creation of a new node. As it will be shown in Sect. III the simple Round Robin (RR) – presently in use in many commercial clouds – scheduling can lead to the creation of unnecessary nodes.

### B. Problem Model

Given an application (as described in the previous section) consisting of several components and connectors (C/Cs) $C = \{C_1, ..., C_m\}$ we need to schedule them on a set of nodes $N = \{N_1, ..., N_k\}$ – virtual machines inside one or several clouds or even physical machines in a plain-old data-centre – such that the total running cost (infrastructure + services) is minimized, node load is evenly balanced, and the application availability is increased. We assume that each node can contain several *Virtual Nodes* (VNs), each executing instances of exactly one type of C/Cs (e.g., web server, message queue, backend service, etc.). The VNs are homogeneous and atomic entities, i.e., we relocate entire VNs and all related C/Cs. When relocating a VN all contained C/Cs are individually relocated by requesting each C/C to make a snapshot and stop executing, then moving the snapshot to the new node and starting the C/C from the snapshot. After all C/Cs were relocated as previously described we consider the entire VN to be relocated. We also assume that the creation of nodes and C/Cs is the responsibility of a scaling mechanism which is beyond the scope of the current paper.

Considering that each cloud is made up of several zones (i.e., data-centres) we assume the following: cost between nodes in the same zone $<$ cost between nodes in different zones but in the same cloud $<$ cost between nodes in different clouds.

We obtain the total relocation cost for one C/C on $N_k$ given by the following single aggregate objective function:

$$cost_{C_i}^k = \omega_1 c_{C_i}^k + \omega_2 rc_i + \omega_3 \lambda_k \quad (1)$$

where: $\omega_i$ represent a set of the weights attached to each objective – the objectives are normalized as they have different limits and units of measure; $c_{C_i}^k$ represents the recurring *running cost* for C/C $i$ on $N_k$ – e.g. estimated *network traffic costs*, *execution cost* and other cloud services; $rc_i$ represents the one-time *relocation cost* of C/C $i$ – set to 0 if no relocation occurred after the moment the C/C was last relocated and its value was computed; and $\lambda_k$ represents the load of $N_k$ after relocation and is computed as follows:

$$\lambda_k = \sum_{VN_i \in N_k} \sum_{C_j \in VN_i} \left( \omega_1^{'} CL_j + \omega_2^{'} ML_j + \omega_3^{'} NL_j \right) \quad (2)$$

where: $\omega_i^{'}$ is another set of objective weights; the triplet $\langle CL_j, ML_j, NL_j \rangle$ represents the CPU, memory and network loads of $C_j$; and $\lambda_k \leq MAX\_LOAD$.

Depending on the values of $\omega_i$ Relation 1 can inflict high C/C relocation costs on highly loaded nodes or on nodes which communicate with C/Cs outside of their zone.

The total relocation cost of a VN on a new $N_k$ is computed as shown in Relation 3. It is represented as a sum between the load of the node holding the VN and the cost of each C/C running on that VN. As $cost_{C_i}^k$ already contains the load of $N_k$ we need to subtract it from every C/C cost and to adjust the node load weight in order to properly obtain $cost_{C_i}^k$ from $cost_{VN}$:

$$cost_{VN} = \mid VN \mid \omega_3 \lambda_k + \sum_{C_i \in VN} \left( cost_{C_i}^k - \omega_3 \lambda_k \right) \quad (3)$$

The total cost of an application over a period of time $t \in [t_1, t_2]$ can be defined as in Relation 4 and its computation is similar with the one used for calculating $cost_{VN}$:

$$\sum_{t=\overline{t_1,t_2}} \sum_{N_k \in N} \Bigg( |VN|^2 \omega_3 \lambda_k(t) + \sum_{VN \in N_k} \left( cost_{VN}(t) - |VN| \omega_3 \lambda_k(t) \right) \Bigg) \quad (4)$$

Our model is an adaptation of Amazon's Dynamo key-value store [3], which consists of a circular key space (a ring) split evenly into a number of VNs, each assigned to a node, and relies on key consistent hashing to distribute its load. Dynamo was selected as it already provides a method for building decentralized and reliable distributed systems. We have adapted it by: (1) replacing the values with C/C instances; (2) statically allocating certain VN ranges to a specific C/C type; and (3) carefully choosing the key of a C/C so that we efficiently fill its associated VN type range.

As previously mentioned the ring is statically partitioned into $N_{max} \times |C_{types}|$ partitions. The idea is to assign an equal number of partitions to each C/C type (part of $C_{type}$ set) so that given an estimated maximum number of nodes ($N_{max}$) required by the application, each node could have at least one partition for every $C_{type}$. These VNs should be allocated to nodes so that in case of failures (even network splits) there remains a high probability that $|C_{types}|$ types still execute. So the application will continue functioning uninterrupted (maybe with lower throughput) until it re-scales to a proper size which fulfils the imposed SLA.

In order to reduce costs, our system starts with a single node which hosts all the C/Cs needed for the application start-up (usually one C/C instance of each type), but as load builds-up new nodes are added. As in Dynamo, each VN is at all times mapped to a single node and the actual start of the VN is determined by the presence of at least one C/C inside it. As a new VN is started the allocation costs of its C/Cs are computed as defined by Relation 3. Also each time we start a new C/C instance its identifier – which determines its VN – is carefully chosen based on policy specific to the C/C type . Section II-C will describe a probabilistic method for selecting VNs. We underline again that the scaling mechanism is beyond the scope of this paper and it suffices to say that it should incorporate a monitoring and decision mechanism that oversees each C/C type.

Given the description of the model the goals of the proposed algorithm are:

- ensure cost reduction: $minimize \sum cost_{VN} \; \forall \; VN$;
- ensure C/C high-availability: every node contains $\mid C_{types} \mid$ C/C types;
- ensure nodes are have their loads maximized and balanced before scaling: $\forall \epsilon > 0 \exists_{N_k \in N} ! \forall_{N_i \in N \setminus \{N_k\}} ((MAX\_LOAD - \lambda_i \leq \epsilon) \wedge (MAX\_LOAD - \lambda_k > \epsilon))$.

## C. Algorithm Description

Because of the good results provided by Genetic Algorithms (GA) [4], [5] when scheduling Grid applications we opted for a similar approach for our scenario of scheduling Web 2.0 applications on multi-cloud systems. GAs have the advantage of allowing a scheduler to explore a wide range of solutions through mutations of an initial population. They are also used when solving multi-criteria optimisation. A mutation usually consists of one perturbation step in each population element. Applied iteratively the perturbations can lead to improved results as the solution converges to a local sub-optimal solution. The larger the initial population to be perturbed the likely the chances that a significantly improved schedule is obtained.

The general structure of a GA is depicted in Fig. 1. The *perturb* function gives each GA its uniqueness. Perturbations in the population are usually produced by mutations of the elements through random relocation or swapping. As each perturbation induces extra costs – because VNs get migrated between nodes – we restrict mutations from being obtained out of swapping.

```
1: Generate the set of identical initial schedules:
2: $S \leftarrow \{S_1, \ldots, S_n\}$
3: $maxIterations_k \leftarrow 0,\ \forall k = \overline{0, | N |}$
4: while ⟨the stopping condition is false⟩ do
5:     for $i = \overline{1, n}$ do
6:         $S'_i \leftarrow$perturb($S_i$)
7:     end for
8:     $S \leftarrow$ select($S, \{S'_1, \ldots, S'_n\}$)
9: end while
```

Figure 1. Pseudocode of the Genetic SA

```
1: for ⟨$N_k : \lambda_k > MAX\_LOAD$⟩ do
2:     $VN_{kj} \leftarrow$ random VN from $N_k$
3:     $N_m \leftarrow$ random node such that $\lambda_m < MAX\_LOAD$
4:     if ⟨not found $N_m$⟩ then
5:         increment $maxIterations_k$
6:     else
7:         relocate $VN_{kj}$ to $N_m$
8:     end if
9:     if ⟨ $maxIterations_k \geq L_1$⟩ then
10:        $N_{new} \leftarrow$ newly allocated node
11:        $maxIterations_{new} \leftarrow 0$
12:        while ⟨$\lambda_k > MAX\_LOAD$ and $\lambda_{new} \leq MAX\_LOAD$⟩ do
13:            relocate random $VN_{kj}$ to $N_{new}$
14:        end while
15:        $retries \leftarrow 0$
16:        while ⟨not all C/C types on $N_{new}$ and $\lambda_{new} \leq MAX\_LOAD$ and
           $retries \leq L_2$⟩ do
17:            randomly pick VN holding C/Cs of type not already existing on $N_{new}$
18:            increment $retries$
19:        end while
20:    end if
21:    if ⟨ $\lambda_k < MAX\_LOAD$ ⟩ then
22:        $maxIterations_k \leftarrow 0$
23:    end if
24: end for
```

Figure 2. Pseudocode of the perturb($S_i$) function

Our proposed solution is depicted in Fig. 2. Starting with a given population element we determine the set of overloaded nodes (i.e., $\lambda_k > MAX\_LOAD$). For every such node (i.e., $N_k$) we randomly migrate one VN to another randomly selected node which is under-loaded. If no such node is found after a certain number of attempts ($maxIterations_k$) a new node ($N_{new}$) is created and while $N_k$ remains overloaded and $N_{new}$ under-loaded we randomly move VNs from the former to the latter. In order to obtain high C/C availability we also attempt to move VNs holding C/C of different types than the ones already existing on $N_{new}$ as long as the new node is under-loaded.

Every mutated element replaces the former if the high C/C availability is achieved during the mutation step.

The *stopping condition* (cf. Fig. 1) for the mutation chain is set to true after a given number of iterations.

Once we obtained a set of perturbed elements the selection of the final schedule is performed by the *select* function. The selection is based on the three goals of the algorithm as depicted at the end of Sect. II-B. As the load balance issue is solved during the perturbation stage we focus on selecting the element that offers the lowest cost and achieves the desired C/C high availability. The cost of the platform is computed based on the cost of each VN (cf. Relation 3). The algorithm is executed every time new C/Cs are generated. As it can be noticed there is no method of moving C/Cs from underutilized nodes. The reason for this is that we leave the matter of deallocating underused nodes to the node manager component which is not addressed in this paper. This component decides based on its own metrics when to remove a node and to relocate – by calling our algorithm – the VNs residing on it.

The initial placement of every C/C is done as follows: first the node which provides the least cost (cf. Relation 1) for the C/C is selected; then a VN – from those fostering C/Cs of the same type – is selected randomly by favouring the ones with the largest number of C/Cs and the new C/C is assigned to it. The idea behind favouring VNs containing large number of C/Cs is to ensure high node load with a minimum number of running VNs.

## III. Experiments

In what follows we present the experimental setup and obtained test results.

### A. Experimental Setup

In order to test the proposed SA we opted for a real case application architecture in the form of a social-news Web 2.0 application (cf. Fig. 3). It allows users to add, vote and retrieve news for a certain topic. The application has seven types of components (e.g., web serve; add news, vote news, list news and detect spam components; database and messaging system) and two types of connectors (e.g., one for the database and one for the messaging system). From this example we can see that some components also act as connectors (i.e., the messaging system). Also we must note that these are only the types of C/Cs, and not their instances; as each of these types has one or more instances, depending on the load, which work together – even the database is actually a distributed database with multiple processes, each one embodied as one component.

The application flow is depicted in Fig. 3. Users start by accessing the web page through a browser, thus by sending requests to the web server. Depending on the request type (e.g., to add, vote or list the news) the request is placed in one of the corresponding message queues (the routing process). The attached components consume messages from their queues and perform their logic, by accessing the necessary database records. All C/Cs can scale-up as follows:

- the web server scales when the number of requests per second (or concurrent persistent connections) reaches a certain limit which slows down the server's throughput and increases the response latency;
- the components scale when the number of messages in the attached message queue increases as a result of either the increased request rate or the incapacity of the components to process them in a timely manner;
- the message queues scale when the number of requests per second or the number of stored messages goes

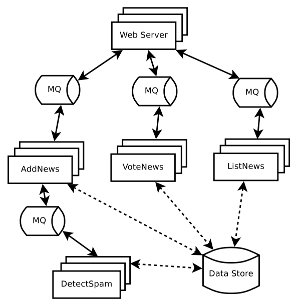


Figure 3. Overview of the social-news Web 2.0 test application

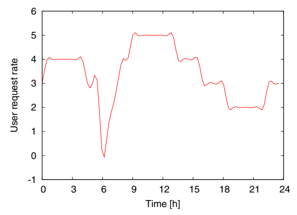


Figure 4. User request rate during a 24h period as a result of access rate variation

beyond a certain threshold which makes message processing inefficient;

- the database scales as the total amount of stored data or the number of operations reach a certain threshold, which as in the case of the messaging system impacts its performance.

Scaling down of C/Cs takes place in a similar manner.

For simulating C/C creation – as a result of application scaling – we opted for two approaches. The first one (called Scenario I) relies on the user activity within a 24h period for workstation tasks adapted to the current application requirements. This model is derived from the work of Feitelson [6] and the request graphic is shown in Fig. 4. The second approach (called Scenario II) relies on the study performed by Vicari [7] for HTTP session arrival rate and number of user connections per page. As stated in the cited work, these can be modelled by Weibull/Lognormal respectively Pareto/Lognormal distributions. A total of 922 C/Cs have been generated for Scenario I and 758 for Scenario II.

For simulating node creation we have taken an approach where the cloud and zone in which they reside is determined using a uniform random variable. Tests have assumed the existence of two clouds each with two zones. The maximum number of nodes required by the application was set to 20.

Each C/C type has been given a constant CPU, memory and network usage which are used at determining the C/C cost. The reason for opting for such an approach is that we consider that every C/C, except newly created ones, should be used at its maximum, otherwise there would not be any need for scaling them up. Similarly C/Cs used at minimum load require to be scale down and their request re-routed to remaining C/Cs of the same type. The values for $\omega_i$ were set to $1/3$ hence giving the objectives equal importance.

In order to compare the efficiency of our algorithm we tested it against a RR implementation tailored to the representational model of Amazon Dynamo. The algorithm is briefly explained in what follows.

VNs are grouped by their types – given by the C/C type running on them – and the first VN in each group is selected. Iterating on a group is accomplished by using the RR approach. The VNs of each node are iterated similarly.

Each new C/C is assigned to a VN of similar type in a RR fashion be selecting the next VN in the ring. Afterwards we compute the number of overloaded nodes and create a new node if at least such a node exists. Next, one VN – given by the current position in the corresponding node ring – from every node is migrated on the new node and the position in each ring is updated. The cycle continues until no over loaded nodes remain. The final cost is given by Relation 4.

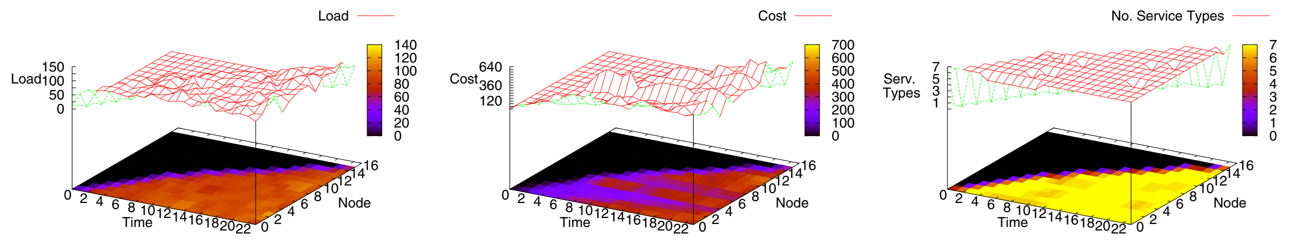
## B. Test Results

As the results for the two considered scenarios are quite similar we only analyse in what follows the results for Scenario I (cf. Fig. 5). The subfigures show the average load, cost and C/C availability for every node in the platform. The graphics represent instantaneous results for every allocated node over a 24h period of time as a result of C/C creation. The fully black cells in these subfigures indicate that a node has not been allocated yet.

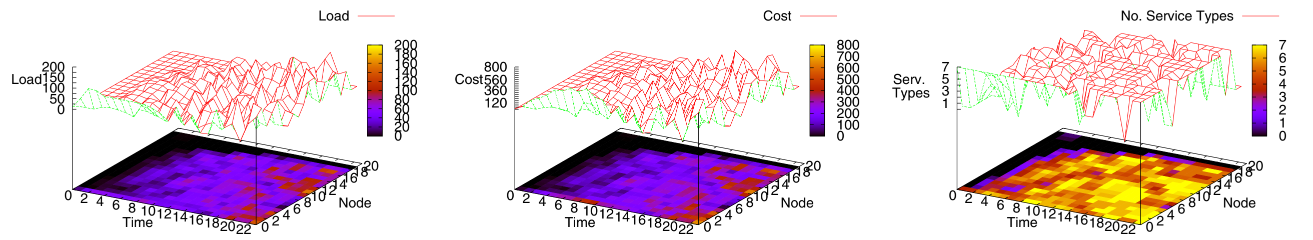
As each node becomes overloaded (i.e., $\lambda_k > 90\%$) – due to C/C creation resulted from high user usage – the tested algorithms allocate new ones. Figure 5(a) shows a linear evolution in the allocation of nodes made by our algorithm. In contrast RR allocates many new nodes simultaneously (as indicated in Fig. 5(d) by the large number of newly created nodes allocated for specific time slots). The reason for this behaviour is that RR does not consider $\lambda_k$ as essential in the allocation process.

As noticeable in Figs. 5(a) and 5(d) the load produced by our solution is not as heterogeneous as that given by the RR method as indicated by the many spikes in the latter's graphic. The RR algorithm also leaves many existing nodes unused as a result of its circular reallocation of C/Cs. This leads to highly unbalanced loads and so increases the costs caused by newly allocated nodes.

Our algorithm fully utilizes nodes immediately after allocation as indicated by the homogeneous orange colour of the graphic. For this reason a low number of new nodes is allocated. Since a high node workload policy is arguable as it can cause performance degradation – and hence SLA violations. To prove that even with a smaller load threshold the algorithm keeps its behaviour we show for comparison in Fig. 6 the results for a load threshold of 70% (as opposed to the 90% threshold used originally). As seen the nodes' loads remain almost homogeneous and fully used in the limits



(a) Node load as produced by our algorithm (b) Node cost as produced by our algorithm (c) C/C availability as produced by our algorithm



(d) Node load as produced by the RR algorithm (e) Node cost as produced by the RR algorithm (f) C/C availability as produced by the RR algorithm

Figure 5. Simulation results for Scenario I

given by the reduced threshold. The results for cost and C/C availability are not shown due to space limitations, however they are similar with the ones addressed in Scenario I.

Concerning platform cost we notice a rise in the cost per node when using our solution (cf. Fig. 5(b)). The motive for this behaviour is that loaded nodes usually have high number of C/Cs and thus the communication cost rises proportionally. As the algorithm does not consider grouping together C/Cs that are strongly linked the cost is undoubtedly expected to rise. However the cost per node is almost homogeneous and is an indication of the efficient node allocation as nodes equally share their costs.

The overall cost to running a platform when using our approach was of about 6,011 compared to that of the RR which was 5,742. The larger cost given by our algorithm is motivated by the higher costs per node observed in Fig. 5(b). However our algorithm counterbalances this cost overhead by the overall high-availability, homogeneous node load and lesser number of required nodes. The cost could be reduced by a smarter allocation of new nodes or by increasing the weights of $c^k_{C_i}$ and $rc_i$ from Relation 1).

When taking into consideration the C/Cs availability per node we observe that our solution produces relatively uniform node loads and highly available C/Cs which is not the case of RR. To obtain the desired high availability we need a perfectly levelled graphic at a hight of 7. As it can be seen in Fig. 5(f) RR exhibits many spikes which prove that C/C types are not evenly spread on nodes. In contrast, our algorithm shows (cf. Fig. 5(c)) a greater number of nodes having all C/C types running on them for the entire period of the execution. This is to be expected as our algorithm tries to move a partition from each C/C type to every new node while the RR algorithm does not take this into account.

In terms of schedule speed RR took around 300ms to execute while our SA required on average 4s. The large difference is due to the fact that the our scheduler used a population of 10 elements for taking a decision. As the tests used a sequential exploration the runtime leads to an average of 400ms for every element which is close to the RR case.

## IV. State of Art

Despite the early stage in the development of cloud computing, there is some research towards designing efficient SAs for load balancing inside (multi-)clouds. Most public clouds use simple scheduling heuristics for load balancing. Examples include Amazon which relies on RR and Rackspace which allows users to pick from several choices that comprise of configurable random, RR, weighted RR, least connections, and weighted least connections strategies. As shown in this paper RR strategies used for scenarios involving node load and high availability are not efficient when considering application made of heterogeneous C/Cs.

Moreover our approach is unique in the sense that we operate on top of virtual machines and schedule co-located application level C/C processes. In contrast other aproaches schedule virtual machines by asserting that one C/C instance runs isolated in its own virtual machine (e.g., Google App Engine, IBM Cloudburst).

In [8] Zhang et al. integrate resource consumption with resource allocation for obtaining efficient load balancing. The strategy is designed as two interconnected feedback loops. The first loop is responsible for optimizing the resource consumption by relying on a multi-objective approach that uses several metrics such as the number of errors or busy threads occurring inside the system. The proposed solution however lacks to integrate the notion of cost as an essential objective. This issue is dealt with in our approach.

Paper [9] proposes a SA that relies on a binary linear programming model to optimize cost for deadline constraint applications. The authors apply the strategy on both public and hybrid (public + private) clouds and show that although their solution works well for the former case, in the latter scenario the solver's performance drastically decreases. Furthermore results show that using the linear programming solver is time consuming and that there is an almost linear dependency between the solver runtime and the number of applications to be scheduled. Our approach focuses on long running application instead of deadline constrained and as shown in Sect. III-B provides results similar with that given by RR, a simple but fast algorithm.

Another solution based on linear programming is given by Mao et al. [10]. They propose a solution to automatic scaling by operating inside a deadline and budget constrained environment. The mechanism is evaluated on both real and simulated applications using the Microsoft Azure platform. The approach is proven to be efficient but is different from ours as we target long running application and not deadline constraint ones.

Another multi-cloud solution is proposed in Chaisiri et al. [11]. The algorithm, called OVMP (Optimal Virtual Machine Placement), relies as well on a linear programming model for determining the optimal placement of virtual machines. The optimal allocation cost is computed given probability distributions of future demand and prices. Starting from the assumption that the reservation price is cheaper than the on-demand price the algorithm first makes a reservation for some nodes and then if necessary allocates additional ones. Foreseeing demand is an interesting approach and could be applied to some extent to Web 2.0 applications. Taking for instance the case of a news web site, special events (e.g., sport or entertainment events) can be predicted and thus advance reservation is possible; however some events that can generate large number of page hits are impossible to predict (e.g., natural disasters). In this case advance reservation is hard to put in practice as it could unnecessary increase costs if user access patterns would behave unpredictably. Presumably in these situation on demand approaches such as our own would be preferred.

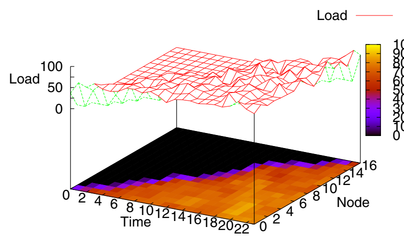


Figure 6. Node load as produced by our algorithm for a node load threshold set to 70%

Yazir et al. [12] propose a distributed strategy for dynamically allocating resources on clouds. Their approach uses a multi-criteria selection for deciding which virtual machines to relocate and on what nodes. The selection process ranks alternatives based on their pairwise comparison and the decision on where to relocate a virtual machine relies on CPU, memory and network usage as well on their recent variability. Our approach uses a similar approach but disregards variability as criterion. The method proves the efficiency of a distributed approach against a centralized one by minimizing the number of VN relocations. However it does not consider the global deployment and execution cost when making the selection decisions. This can be an issue since the presented tests show that the number of created nodes is high. The authors infer that the low utilization (set at 70%) and high number of nodes reduces the risk of SLAs violations. On one hand the approach leaves a large portion of the node unused, while on the other hand SLAs could be maintained by offering highly available components that are able to divide among themselves the load though an intermediary proxy. Our solution followed this approach by offering several components of the same type distributed on allocated nodes while keeping the node load maximized.

In [13] a flexible resource provisioning on clouds is presented. The authors present an approach in which users simply submit an application and are presented with a price estimate and schedule which the cloud needs to obey in order to meet the user requirements. Hence the burden of specifying the number of virtual machines and optimizing the application execution on those machines is shifted from the user to the cloud. Tests performed on the proposed scenario used four SAs in order to determine the schedule that offers the best cost. The results were promising but the testbed was restricted to Map-Reduce applications – i.e., applications comprised of parallel and data intensive tasks chained in a pipe-line. While our approach tackled the cost minimization in long running Web 2.0 applications, the algorithm is generic enough and could provide good solutions even for such a use case.

A recent paper on this subject is the work of Oprescu et al. [14]. The authors of the work study the problem of budget-constrained schedules of bag-of-tasks problems on clouds. The work focuses on a statistical method for estimating costs and application makespan for a given bag on different clouds. Their work however does not addresses the problem of reliability discussed in this paper.

The presented work on cloud scheduling has shown the efficiency of various techniques – out of which the linear programming model seems to be the most vastly used – but failed to cope with an important aspect: high-availability. This issue is extremely important as it allows an application to continue functioning even when several nodes have failed. The work presented in this paper tackled this aspect as well from a multi-objective point of view by considering besides availability the cost of the application and the node load.

## V. Conclusions

This paper has presented a multi-objective SA designed for offering increased high-availability and load balance inside multi-cloud systems. The targeted applications are represented by long running applications (e.g., Web 2.0). As shown in Sect. IV most cloud SAs ignore this aspect. Our algorithm was tested on a real life social news application with synthetically generated costs and loads. Results were compared with a RR algorithm in use in many commercial cloud systems and proved the efficiency of our solution.

Future work will involve the problem of node provisioning and C/C scaling as well as the algorithm's integration inside the mOSAIC platform[1].

## Acknowledgment

The work of the authors has been partially funded by the European FP7-ICT project mOSAIC grant no. 256910.

## References

[1] D. Petcu, C. Craciun, N. Neagul, M. Rak, and I. Lazcanotegui, "Building an interoperability api for sky computing," in *Proceedings of the 2011 International Conference on High Performance Computing and Simulation Workshop on Cloud Computing Interoperability and Services*, 2011, pp. 405–412.

[2] S. Venticinque, R. Aversa, B. Di Martino, and D. Petcu, "Agent based cloud provisioning and management. design and prototypal implementation," in *Proceedings of 1st International Conference on Cloud Computing and Services Science*, 2011, pp. 184–191.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, October 2007. [Online]. Available: http://doi.acm.org/10.1145/1323293.1294281

[4] T. D. Braun, H. J. Siegel, and N. Beck, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 801–837, 2001.

[1]http://mosaic-cloud.eu/

[5] F. Zamfirache, M. Frîncu, and D. Zaharie, "Population-based metaheuristics for tasks scheduling in heterogeneous distributed systems," in *NMA '10: Proceedings of the 7th International Conference on Numerical Methods and Applications*, ser. Lecture Notes in Computer Science, vol. 6046. Springer-Verlag, 2011, pp. 321–328.

[6] D. G. Feitelson, "Workload modeling for computer systems performance evaluation," February 2011. [Online]. Available: http://www.cs.huji.ac.il/ feit/wlmod/

[7] N. Vicari, "Modeling of internet traffic: Internet access influence, user interference, and tcp behavior," Ph.D. dissertation, University of Würzburg, 4 2003.

[8] Y. Zhang, G. Huang, X. Liu, and H. Mei, "Integrating resource consumption and allocation for infrastructure resources on-demand," in *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, ser. CLOUD '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 75–82. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2010.11

[9] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove, "Cost-optimal scheduling in hybrid iaas clouds for deadline constrained workloads," in *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, ser. CLOUD '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 228–235. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2010.58

[10] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *Proceedings of the 2010 11th IEEE/ACM International Conference on Grid Computing, Brussels, Belgium, October 25-29, 2010*, 2010, pp. 41–48.

[11] S. Chaisiri, B.-S. Lee, and D. Niyato, "Optimal virtual machine placement across multiple cloud providers," in *Proceedings of the 2009 4th IEEE Asia-Pacific Services Computing Conference*, 2009, pp. 103–110.

[12] Y. O. Yazir, C. Matthews, R. Farahbod, S. Neville, A. Guitouni, S. Ganti, and Y. Coady, "Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis," in *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, ser. CLOUD '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 91–98. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2010.66

[13] T. A. Henzinger, A. V. Singh, V. Singh, T. Wies, and D. Zufferey, "Flexprice: Flexible provisioning of resources in a cloud environment," in *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, ser. CLOUD '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 83–90. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2010.71

[14] A.-M. Oprescu, T. Kielmann, and H. Leahu, "Budget estimation and control for bag-of-tasks scheduling in clouds." *Parallel Processing Letters*, vol. 21, no. 2, pp. 219–243, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/ppl/ppl21.html#OprescuKL11