Scheduling Highly Available Applications on Cloud Environments

Marc Eduard Frîncu

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

Abstract

Cloud computing is becoming a popular solution for storing data and executing applications due to its on-demand pay-per-use policy that allows access to virtually unlimited resources. In this frame applications such as those oriented towards Web 2.0 begin to be migrated on cloud systems. Web 2.0 applications are usually composed of several components that run indefinitely and need to be available to end users throughout their execution life cycle. Their availability strongly depends on the number of resource failures and on the variation in user hit rate. These problems are usually solved through scaling. A scaled application can span its components on several nodes. Hence if one or more nodes fail it could become unavailable. Therefore we require a method of ensuring the application's functionality despite the number of node failures. In this paper we propose to build highly available applications, i.e., systems with low downtimes, by taking advantage of the component based architecture and of the application scaling property. We present a solution to finding the optimal number of component types needed on nodes so that every type is present on every allocated node. Furthermore nodes cannot exceed a maximum threshold and the total running cost of the applications needs to be minimized. A sub-optimal solution is also given. Both solutions rely on genetic algorithms to achieve their goals. The efficiency of the sub-optimal algorithm is studied with respect to its success rate, i.e., probability of the schedule to provide highly available applications in case all but one node fail. Tests performed on the sub-optimal algorithm in terms of node load, closeness to the optimal solution and success rate prove the algorithm's efficiency.

Preprint submitted to Future Generation Computing Systems *April 3, 2012*

Keywords: cloud scheduling, multi-objective scheduling, meta-heuristics, nonlinear-programming, high available systems

1. Introduction

Cloud computing has become a popular solution for storing data and executing applications for many companies due to its on-demand pay-per-use policy which allows access to virtually unlimited resources. Public cloud providers such as Amazon, Google, Microsoft or Rackspace offer access to their data centers by means of infrastructure (e.g., Amazon EC2 [1], Rackspace cloud [2]) or platform (e.g., Google App Engine [3], Microsoft Windows Azure [4]) level APIs. There even exist solutions for companies that wish to create their own cloud systems (e.g., Eucalyptus [5], OpenStack [6]). For companies as well as for private users choosing to switch to a cloud based solution does not come without risks such as network crashes and splits, data center failures or even security breaches. A key aspect when migrating to cloud computing is to ensure the application *availability* in case of failures.

For Web 2.0 applications it is essential to minimize the impact of failures throughout the application's life cycle and to constantly monitor and adapt resources to user request rates and failures through *automatic scaling* (cf. Sect. 4).

Web 2.0 applications usually consist of several interlinked components (e.g., web server, database, application logic modules or cloudlets [7], communication server) which need to scale according to user request rates. Due to this modular approach a further requirement, especially in case of failures, is to ensure that all required components execute on the remaining resource nodes. This guarantees that despite a possible considerable drop in the applications's performance, caused by under provisioning or node failures, the application is still able to handle a certain percent of the user requests. In this way the application's ability to handle large incoming requests is diminished – but not halted – until the components are scaled back to fit the size of the request rate.

In case of performance drops the average response time (milliseconds to seconds) increases. This causes some users to experience long response times and even timeouts. For profit oriented websites this translates into less profit being made, due to low page clicks or commercial activities.

We mentioned earlier the notion of **availability**. In web based applications this is defined as the ratio between the number of serviced requests and

the total number of requests [8]. It is also generally expressed in terms of "number of nines": the more "nines" the less denial of service messages in minutes per year [9]. According to Gray et al. [9] **High Availability** (HA) means an availability of 99.999%, i.e., 5 minutes of unavailability per year. High-availability systems are characterized by fewer failures and faster repair times.

Two major issues arise when dealing with scalable and component based applications that need to be HA. First, as nodes cost it is undesirable for long running applications to rent extra resources as in the case of many HA cluster systems that require $N+M$ nodes to operate (cf. Sect. 2). Second, as applications are modular we could end up isolating on several nodes particular components. In case the hosting nodes would fail our application would experience a drop in its availability while it restarts the failed resources.

In this paper we propose a solution for these two problems by introducing a method of placing each application component type on every needed node. Hence we (1) avoid allocating unnecessary extra nodes and (2) ensure with a certain degree of probability that in case only one node remains the application would still execute – although at a slower rate – as though all the required components types were available on it.

The proposed **scheduling algorithms** consider the method of scaling to be known *a priori* and focus on searching for an optimal allocation of components on nodes in order to ensure a homogeneous spread of component types on every node. Numerous current scheduling strategies (cf. Sect. 2) assume short-to-medium lived tasks (e.g., bag of tasks, MapReduce or parameter sweep applications) and ignore the Web 2.0 applications which are inherently long (or infinitely) running. Despite the apparent simplicity of such schedules it is often the case that we need to periodically migrate components in order to optimize node load, minimize communication costs between nodes and ensure HA.

The rest of the paper is structured as follows: Section 3 describes the application model by introducing the mathematical formalism used throughout the rest of this work. Section 4 presents two scheduling algorithms for achieving HA in the context of this paper. We first address the problem of component and node scaling (cf. Sects. 4.1 and 4.2) as they are essential aspects in the proposed algorithms. Section 4.1.1 deals with the special case in which the components' loads are known and gives an optimal solution to the number of components each node can accommodate. Section 4.2 presents a pro-reactive algorithm for node scaling based on the user hit rate. Section

4.3 then presents the two algorithms used to schedule components on nodes and to reactively allocate nodes if no suitable node exists – i.e., the proactive algorithm failed to allot the necessary nodes. The optimal solution can be used in case the loads of every component are known and is depicted in Sect. 4.3.1, while Sect. 4.3.2 presents the algorithm for the general case. The model used to measure the success rate of the proposed algorithms is addressed in Sect. 4.3.3. Section 5 describes the testing scenarios and includes discussions on the provided success rate, node load and closeness to the optimal solution. The paper concludes by reviewing some of the main achievements of this study (cf. Sect. 6).

2. Related Work on High Available Systems and Cloud Scheduling

The issue of achieving HA systems has gathered a lot of attention with work ranging from cluster and grid to utility computing. The overall problem can be reduced to the problem of placing virtual machines on a limited physical node so that the number of physical resources is minimized. This is also known as the *bin packing problem* [10] which is known to be *NP-hard*.

A popular technique towards HA systems is to use virtual resources that are migrated in case of failures. Several commercial solutions including VMware [11] and Xen [12] use it but take opposite approaches: VMware is a reactive solution which restarts virtual machines on other resources in case of errors. This leads to temporal delays in the application uptime. Xen on the other hand uses a proactive method based on monitoring data and migrates virtual machines before the predicted failure occurs. However this approach has also a major drawback as failures are difficult to predict in advance.

For Internet based services solutions include deploying load balancers which in case one or more servers fail redirect the traffic to the remaining ones [13] [14] [15] [16].

Loveland et al. [17] study how virtualization techniques can augment HA and propose a simple redundant method for placing virtual machines on multiple nodes.

Besides migration there are also solutions based on clustering. Databases servers such as MySQL [18] and key value stores like Amazon's Dynamo [19] rely on it. In this approach processes are distributed to replication servers by using DNS Round Robin [18] or key consistent hashing [19].

One interesting aspect concerning the migration based methods depicted earlier is that they separate the HA from the allocation problem. This raises an important question regarding their overall efficiency when virtual machines contain dependent applications.

Some work which considers HA as part of the allocation algorithm has been done. Recently Machida et al. [20] proposed an optimal solution for achieving HA systems by using redundant nodes. The authors also show that for some cases their algorithm uses less than the $N + M$ nodes needed when classic and simple approaches such as First-Fit Decrease [21] are used.

Gottumukkala et al. [22] [23] propose a reliability-aware allocation algorithm for achieving HA in large scale computing systems. In their work they study the possibility of using only reliable nodes when allocating virtual machines and show their solution to offer a greater improvement in terms of waste time and completion time than the classic Round Robin approach.

All of the previous examples have relied on software to achieve HA systems, yet hardware solutions exist as well. For instance, in their work, Aggarwal et al. [24] propose a low-level isolation for fault containment and reconfiguration for chip multiprocessors. Their method partitions the chip into multiple failure zones which can be exploited by redirecting power from failed chip multiprocessor components to remaining ones.

Our proposed scheduling approach is different from these strategies in the sense that we address a specific problem – that of providing HA long running applications – and also because we operate on top of virtual machines and schedule *co-located* application level component processes. In contrast other approaches – from industry, e.g., Google App Engine [3], IBM Cloudburst [25]; or from academia, e.g., [20], [22] – schedule virtual machines by asserting that one component instance runs isolated in its own virtual machine.

Another difference in our approach is that we take advantage of the inherit scalability property of Web 2.0 based applications and of the component based (workflow-like) structure of these applications. Cloud systems rent resources on an hourly basis and for long running applications this induces regular costs. Therefore applying traditional solutions which allocate redundant resources only increases the overall cost. Due to the fact that scalable applications usually require several components of the same type to coexist in order to cover the requests a solution would be to spread the component types homogeneously on every needed node. In this way we ensure that in case all nodes except one fail we still have all the application components running. So we both achieve HA without needing extra nodes and minimize

the actual number of used nodes.

Virtualization is a key aspect in cloud computing as it allows to harvest the full potential of the "unlimited" resource pool of these systems. Since we have already mentioned attempts to integrate HA with allocation (scheduling) algorithms have been made. However most scheduling solutions used in the industry still use simple scheduling heuristics and either offer the HA as an additional feature [19] or rely on existing virtualization techniques to deal with it [11] [12]. Examples include the Round Robin approach taken by Amazon [1] or Rackspace [2] which offers several policies including configurable random, Round Robin, weighted Round Robin, least connections, and weighted least connections.

Cloud scheduling has been widely addressed in recent years in academia as well. For instance we have a variety of methods for determining the (sub-)optimal schedules: linear programming [26] [27] [28], multi-objective functions [29] [30], genetic algorithms [31] [32] [33] [34], statistical estimates [35], dynamic selection of the best schedule [36] or feedback control loops [29].

Genetic Algorithms (GA) proved to give good results [37, 38] when scheduling Grid applications. It comes with no surprise that they have been widely proposed for Cloud systems also. They are split into algorithms for virtual machine scheduling and for application scheduling. Due to their nature GAs allow the exploration of a wider ranger of solutions through mutations and crossovers. They can be used in solving multi-criteria optimization too.

Gu et al. [33] propose a GA for scheduling virtual machines and show it to be efficient with regard to other approaches such as Least-Load or Rotating-Scheduling. In [31] Zhong et al. present a GA that relies on the Dividend Policy in Economics in order to select a(n) (sub)optimal allocation for the virtual machine requests. Tests show the algorithm to perform better than Eucalyptus, OpenNebula [39] or Nimbus [40] schedulers and faster than the traditional GA for Grid scheduling.

Tayal [32] depicts a GA for scheduling independent tasks on clouds but focuses on tasks with execution times rather than long running applications as we do. Zhao et el. [34] also propose a GA for independent and divisible tasks for distributed systems but do not consider costs such as those for renting resources (i.e., nodes and network bandwidth). A short overview of a GA relying on the Lexi – search is given in [41]. The paper however lacks any validation of the algorithm's efficiency.

Linear programming is another efficient scheduling technique as it allows

schedulers to determine the optimal placement of tasks on resources.

Paper [28] proposes a SA that relies on a binary linear programming model to optimize cost for deadline constraint applications. Tests are performed on both public and hybrid (public + private) clouds and show the efficiency in the former case. Results prove however that the method is time consuming and that the dependency between the time needed to solve the problem and the number of applications to be scheduled is almost linear.

Another solution based on linear programming is given by Mao et al. [26]. They propose a solution to automatic scaling by operating inside a deadline and budget constrained environment. The approach is different from our own as we target long running application and not deadline constraint ones.

Chaisiri et al. [27] propose an algorithm called OVMP (Optimal Virtual Machine Placement). It relies 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. Foreseeing demand is an interesting approach and could be applied to some extent to Web 2.0 applications when predicting future hit rates.

3. Application Model

In what follows we focus on Web 2.0 applications and present the application model used to represent them.

Web 2.0 applications are ideal candidates for studying HA and scalability in cloud systems as they are characterized by long running times (almost all run indefinitely) and high variations in user access rates. The latter is known to depend 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 *unpredictable* – in case of randomly occurring events (e.g., media hype, disasters, etc.). Figure 1 depicts an example Web 2.0 application in the form of a social-web site that allows users to add, vote and retrieve news for a certain topic.

We model the architecture of Web 2.0 applications by using *components* and *connectors* (C/Cs).

A *component* is seen as a module that handles part of the overall application logic. This includes web servers – responsible for handling HTTP

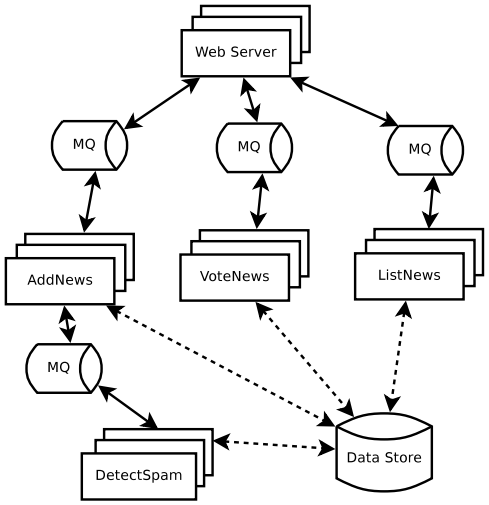


Figure 1: Overview of the social-news Web 2.0 test application

requests and responses –, databases – for managing required data – and back-end services – that perform application specific tasks. Components run on (or consume) cloud resources such as virtual machines and block devices, cloud file-systems or databases.

A *connector* is viewed as a communication tunnel between several components. It can be represented by a socket or message queue described by the AMQP standard [42].

The social-web application depicted in Fig. 1 can be easily mapped on a C/C architecture as shown in what follows. Each individual module (e.g., web server; add news, vote news, list news and detect spam subroutines; database and messaging system) can be seen as a distinct component which interacts with other components through connectors represented by message queues. Message queues have several advantages over sockets as they not only allow asynchronous calls but also make communication transparent as components only need to know the queue address and not the IP and port of the partner components. The obtained application has seven component types and one connector type.

3.1. Mathematical Model

Given the described application design we present the mathematical notations and model used for representing it.

The set of **components** that make up an application is defined as a set $C = \{c_1, ..., c_m\}$. Based on this set we define the **connectors** by using a directed graph $\mathcal{D}$ in which a component c_i is said to be connected with c_j if there exists an edge $(i, j) \in \mathcal{D}$.

Every component has an input and output **throughput**. Throughput is a measure of the rate in which messages from/to the components are being consumed/produced per time unit (e.g., seconds). We assume the throughput values to be, for every component type, within predetermined limits and consider that large fluctuations outside these boundaries are caused by component malfunctions due to internal logic errors. The limits inside which the throughput varies are considered to be determined *a priori* by using profiling techniques. An input/output throughput of a component c_i from/to component c_j is represented by r_{ij}^{in} respectively r_{ij}^{out}. This means that c_i can consume r_{ij}^{in} messages per second from c_j and can produce r_{ij}^{out} messages per second to be consumed by c_j. While it can be argued that the time needed to consume/produce a message also varies depending on network load we assume in what follows that these fluctuations are small enough to be neglected. This can be backed by the fact that most HTTP communication with a website contains with few exceptions (e.g., file tranfers) small message bodies (e.g., the average size of a web page is around several kB).

In general $r_{ij}^{out} \neq r_{ji}^{in}$. Figure 2 exemplifies the throughput of the social-web application (cf. Fig. 1) in case only one component from each type exists.

An efficient application which does not need to scale is one in which a component's c_j input throughput for consuming messages can handle the output throughput of c_i, with $(i, j) \in \mathcal{D}$:

$$\sum_{j:(j,i)\in\mathcal{D}} r_{ji}^{in} = \sum_{i:(i,j)\in\mathcal{D}} r_{ij}^{out} \tag{1}$$

Relation 1 is usually not true for a platform that handles only one instance of each component type. The motive for this is that usually every component has different input/output throughputs, i.e., $\forall_{c_i,c_j} \exists_{r_{ij}^{out}, r_{ji}^{in}} (r_{ji}^{in} \neq r_{ij}^{out})$. This paper tries to overcome this aspect by providing a scaling mechanism in which Relation 1 is met for every application, even newly deployed ones. This is usually achieved through scaling and Sect. 4 will further detail this.

As mentioned in Sect. 3 components communicate through connectors that can be modeled as queues. So every component has a **queue** from which

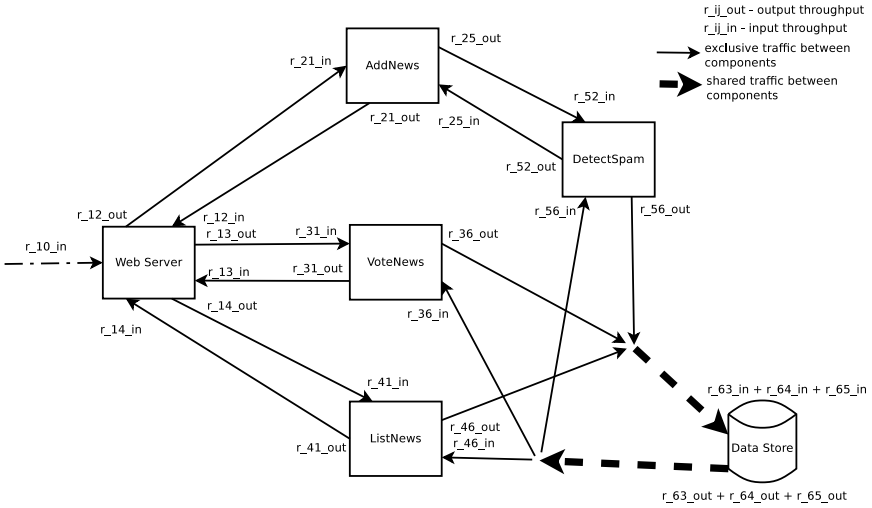


Figure 2: Example of input/output throughputs for every component of the application depicted in Fig. 1

it consumes messages at the rate given by its input throughput. Similarly it produces messages at a certain output throughput and places them in the queue linking it to the next component. As such any request req needs to loop through the entire application graph $\mathcal{D}$ in order to return to the sender. The time needed for this operation is called **response time** and is computed based on Relation 2:

$$response\_time_{req} = \sum_{c_i \in C} \left(\left\lceil \frac{req\_pos_{queue_{ji}^{in}}}{r_{ji}} \right\rceil + \max \left(\epsilon_i(t), \sum_{(i,l) \in \mathcal{D}} \left\lceil \frac{req\_pos_{queue_{il}^{out}}}{r_{il}} \right\rceil \right) \right) \tag{2}$$

where $response\_time_{req} \in \mathbb{N}$; $queue_{ji}^{in}$ and $queue_{il}^{out}$ represent the input respectively output queues of component c_i; $req\_pos_{queue_{ji}^{in}}$ and $req\_pos_{queue_{il}^{out}}$ represent the positions in the input/output queues of message req; and $\epsilon_i(t)$ is a function of time representing the actual time needed by c_i to process req. The relation indicates that a message can be read from only one queue but can be published in any number of queues (the second term of the sum).

The response time should always be minimized. This happens when: (1)

$req\_pos_{queue_{ji}^{in}} \leq r_{ji}$ (i.e., req can be consumed immediately); (2) $req\_pos_{queue_{il}^{out}} \leq r_{ij}$ (i.e., req can be published immediately); and (3) $\epsilon_i(t) \to 0$. The last condition is hard to achieve as it depends on resource processing power at time t and on the size and dependencies of req. The first two conditions are however achievable (by ensuring Relation 1 with a certain ϵ limit) through component scaling as discussed in Sect. 4.1.

The set of allocated **nodes** is represented by a set $N = \{n_1, ..., n_k\}$. Nodes can be *virtual machines* inside one or several clouds or *physical machines* located in a data-center.

Each node n_k can hold one or more **virtual nodes** (vn): $n_k = \{vn_1, ..., vn_l\}$. A *vn* is a homogeneous entity which executes instances of exactly one component type. Hence it can be modeled by a multiset: $vn = \{c_i^{instances}\}$. The *vn* is also *atomic* meaning that it can only be relocated as a whole and not by moving individual components running inside of it. *Vn*s can be seen as containers for executing one or more components. In this way they are similar with Java Virtual Machines which execute Java code. Their goal is to isolate the execution of various types of components.

Next we define the notion of **node load**. Load is seen as the number of processes that execute on a node. Closely linked to it is the concept of resource usage which is sometimes considered to be a direct indicator of the system load (e.g., the greater the load the higher the CPU usage should be). Efficient components will always try to maximize the node usage (CPU, memory, network, etc.). In our model we consider the two terms to be interchangeable.

Therefore we define the node load λ_k as follows:

$$\lambda_k = \sum_{vn_i \in n_k} \sum_{c_j \in vn_i} (\omega_1 CL_j + \omega_2 ML_j + \omega_3 NL_j) \tag{3}$$

where: ω_i is a set of objective weights and the triplet $\langle CL_j, ML_j, NL_j \rangle$ represents the normalized values for CPU, memory and network usage of c_j.

Cost is another important aspect that must be taken into account. Three objectives need to be minimized in this aspect: *communication cost*, *relocation cost* and *node load*. Each of these plays an important role in computing the final cost and while node load is not a monetary cost by itself it is indirectly linked to the efficiency of the node: a load over a maximum threshold τ can inflict delays in processing the requests which causes long wait times for end users. Based on the objectives we define the cost of a component as:

$$cost^k_{C_i} = \omega^{'}_1 reccost^k_{c_i} + \omega^{'}_2 relcost_i + \omega^{'}_3 \lambda_k \quad (4)$$

where: $\omega^{'}_i$ represent another set of weights attached to each objective – the objectives are normalized as they have different limits and units of measure; $reccost^k_{c_i}$ represents the recurring *running cost* for component i on N_k – e.g. estimated *network traffic costs*, *execution cost* and other cloud services; and $relcost_i$ represents the one-time *relocation cost* of component i – set to 0 if no relocation occurred after the moment the component was last relocated and its value was computed;

Considering that migration usually occurs by *vn* relocation we compute its cost as:

$$cost_{vn} =\mid vn \mid \omega^{'}_3 \lambda_k + \sum_{c_i \in vn} \left(cost^k_{c_i} - \omega^{'}_3 \lambda_k \right) \quad (5)$$

As noticed the cost is represented by a sum between the load of the node holding the *vn* and the cost of each component running it. Since $cost^k_{C_i}$ already contains the load of n_k we need to subtract it from every component cost and adjust the node load weight in order to properly obtain $cost^k_{C_i}$ from $cost_{vn}$.

4. Component and Node Scaling

In Sect. 3.1 we mentioned that the efficiency of a Web 2.0 application, seen as request response time, can be affected by fluctuations in user hit rate and failures. This is true as in general $r^{out}_{ij} \neq r^{in}_{ji}$ which leads to increasingly larger response times (i.e., $response\_time_{req} \nearrow$) as messages accumulate in the queues. For Web based applications there is a limit to which a user request is considered to have timed-out. This limit can be customized for individual web servers but the default usually ranges between 120s for Microsoft's IIS [43] and 300s for Apache Tomcat [44]. Any requests exceeding this limit are simply dropped out when read from the queue by a component.

In this paper we consider a pessimistic random failure model in which all but one node can fail at any given moment. We assume this scenario as it allows us to test whether our proposed algorithms achieve or not HA.

From Fig. 2 it can be easily seen that there exists an incoming flow of user requests to the component representing the web server (i.e., r^{in}_{10}). This flow must be always satisfied by the application as any requests that are not dealt

with in due time force the users that issued them to experience increasing waiting times (i.e., high web server response times). To solve this problem the web server must always adjust to the user request rate. Since the I/O throughput of a component cannot be changed dynamically as it depends on the implementation the solution is to scale the components so that their combined throughput matches the request rate. This operation has a cascading effect as all the components need to scale accordingly to match their throughputs. This increases the input/output throughput to $c_i^{instances} r_{ji}$ and $c_i^{instances} rij$, where $c_i^{instances}$ represents the number of instances of components having the type of c_i. Relation 2 becomes in this case:

$$response\_time_{req} = \sum_{c_i \in C} \left(\left\lceil \frac{req\_pos_{queue_{ji}^{in}}}{c_i^{instances} r_{ji}} \right\rceil + \max \left(\epsilon_i(t), \sum_{(i,l) \in \mathcal{D}} \left\lceil \frac{req\_pos_{queue_{il}^{out}}}{c_i^{instances} r_{il}} \right\rceil \right) \right) \tag{6}$$

Thus we obtain in case of the input queue $c_i^{instances} r_{ji} = req\_pos_{queue_{ji}^{in}} + \epsilon$, $\forall req \in queue_{ji}^{in}$ $\forall \epsilon > 0$ small enough. The same relation is true for output queue. This ensures that any message will be consumed/produced immediately. The proof is straightforward:

Proof. Let $req \in queue_{ji}^{in}$ arbitrarily chosen. Since Relation 1 is met as a result of component scaling within ϵ limits: $\sum_{(j,i) \in \mathcal{D}} c_i^{instances} r_{ji}^{in} = \sum_{(i,l) \in \mathcal{D}} c_i^{instances} r_{il}^{out} + \epsilon$, $\forall \epsilon > 0$ small enough, we have: $\mid queue_{ji}^{in} \mid \leq c_i^{instances} r_{ji}$, i.e., the input throughput of c_i can handle all the components of its input queue during one time interval including req: $c_i^{instances} r_{ji} = req\_pos_{queue_{ji}^{in}} + \epsilon$.

The proof for the output queue $queue_{il}^{out}$ is identical. □

Component scaling is performed by creating new component instances which bind to the same message queues as the originals. This ensures that no additional operations are needed to reroute the excess messages to the new components. When scaling occurs it is only inevitable for the existing nodes to scale once the existing nodes get overloaded. Based on this fact we can view **node scaling** as a side effect of component scaling. Section 4.2 details how node scaling is performed in our approach while component scaling is discussed in Sect. 4.1.

There are two approaches that can be used for predicting user hit rate: *reactive* and *pro-active* ones.

Reactive methods (e.g., Rightscale algorithm [45]) usually require knowledge of the current user hit rate. Based on it the scalability algorithm decides to adapt the number of components accordingly. Despite its simplicity this approach experiences a major problem in the fact that it takes time to start up a node (e.g., up to 10 minutes for an Amazon EC2 spot instance). In case of short bursts in the hit rate by the time the nodes are allocated the spike could be already over. In addition the users responsible for the burst would experience large response times caused by under provisioning.

Pro-active approaches are based on historical records to predict the future behavior of the request rate. These methods include linear regression, auto-regression of order 1 [46] [8], pattern matching [47] [48] or neural networks [49]. Similar to their reactive counterparts these methods have disadvantages too. Most pro-active methods have difficulties in catching drastic changes in the request rate as they rely on data which might not have contained values close to the unpredicted spike. This is the case for regression and neural network based techniques as well as for pattern based methods in case the historical data did not comprise similar patterns. In addition public cloud providers usually rent nodes on an hourly basis and as such over provisioning caused by prediction errors rises the costs unnecessarily.

In what follows we consider the method of predicting the user request rate known. We focus on determining from the user hit rate the actual number (1) of components and their placements such that HA is achieved, and (2) of nodes needed to handle this rate. These results will be used in Sect. 4.3 by the scheduling algorithms that we propose for ensuring HA.

4.1. Component Scaling

Component scaling in an application based on the model described in Sect. 3 follows a cascade effect triggered by the number of web server components needed to handle the flow of user hits. The number of web server components needs to be determined so that the rate of producing messages is always balanced by the rate at which they are consumed: $c_{Web\_Server}^{instances} = user\_hit\_rate/r_{Web\_Server}^{in}$, where *user_hit_rate* represents the number of user hits per time unit.

Starting from the number of instances the web server needs, the number of each component type can be determined by using the dependency graph $\mathcal{D}$: $\mid c_i^{instances} r_{ij}^{out} - c_j^{instances} r_{ji}^{in} \mid \leq \epsilon, \forall \epsilon > 0 \ \forall (i,j) \in \mathcal{D}$. The problem that

remains is that of finding exactly how many components of each type can be placed on every allocated node.

As shown in what follows this problem can be solved by using nonlinear optimization when information on the component load is known and through GAs when the load is unknown.

4.1.1. Optimal Number of Components per Node

The optimal number of components of every type each node can hold can be used by the allocation algorithm to decide whether or not to accept a new component for n_k. In case the component load is known the exact value can be determined through nonlinear optimization as described next:

For simplicity we rewrite Relation 3 by considering the load of every existing component load $\lambda_{c_j^k}$ on a node n_k:

$$\lambda_k = \sum_{vn_i^k, i=1}^{|n_k|} \sum_{c_j^k, j=1}^{|vn_i^k|} \lambda_{c_j^k} \leq \tau \tag{7}$$

where τ represents the maximum admissible load for every n_k.

In what follows we study the optimal number of components that can be placed on a node n_k for both homogeneous (identical components and nodes) and heterogeneous (identical/different component types and different node types) systems.

Determining the maximum number of components that n_k can accommodate is trivial for *homogeneous environments* as all component and node types are homogeneous. So we have $\lambda_{c_j^k} = \lambda_{c_l^{k'}} = \lambda_c, \forall n_k, n_{k'} \in N \forall c_j^k, c_l^{k'}$. Based on this observation Relation 7 becomes:

$$\mid n_k \mid\mid vn_i^k \mid \lambda_{c_j^k} \leq \tau \tag{8}$$

As $\mid n_k \mid = \mid n_{k'} \mid, \quad \forall n_k, n_{k'} \in N$ and $\mid vn_i^k \mid = \mid vn_j^{k'} \mid, \quad \forall vn_i^k \in N_k, vn_j^{k'} \in N_{k'}$ for homogeneous systems the previous relation can be generalized for all $n_k \in N$:

$$\mid N \mid\mid n_k \mid\mid vn_i^k \mid \lambda_{c_j^k} \leq \mid N \mid \tau \tag{9}$$

From Relation 8 the maximum number of components that can be placed on n_k can be determined as being $\mid C \mid_k = \mid n_k \mid\mid vn_i^k \mid$. We immediately

obtain the relation between the total number of components ($\mid C \mid_{max}$), the total number of nodes ($\mid N \mid$) and τ:

$$\mid C \mid_{max} = \frac{\mid N \mid \tau}{\lambda_c} \tag{10}$$

When considering *heterogeneous environments* the problem of determining $\mid C \mid_{max}$ is more difficult as each node and component type has its own characteristics.

In what follows we model the load of a heterogeneous system with $\mid N \mid$ nodes by extending Relation 7:

$$\begin{cases} \mid vn_1^1 \mid \lambda^1_{c_{type_1}} + \mid vn_2^1 \mid \lambda^1_{c_{type_2}} + ... + \mid vn^1_{|n_1|} \mid \lambda^1_{c_{type_{|n_1|}}} & = \tau \\ \mid vn_1^2 \mid \lambda^2_{c_{type_1}} + \mid vn_2^2 \mid \lambda^2_{c_{type_2}} + ... + \mid vn^2_{|n_2|} \mid \lambda^2_{c_{type_{|n_2|}}} & = \tau \\ ... \\ \mid vn_1^{|N|} \mid \lambda^{|N|}_{c_{type_1}} + \mid vn_2^{|N|} \mid \lambda^{|N|}_{c_{type_2}} + ... + \mid vn^{|N|}_{|n_{|N|}|} \mid \lambda^{|N|}_{c_{type_{|n_{|N|}|}}} & = \tau \end{cases} \tag{11}$$

It can be noticed in the previous relation that λ_c is dependent on the component type residing on the *vn* and on the node itself. As the number of component types is known we reduce the system of equations to one in which we have exactly $\mid C_{types} \mid$ unknowns instead of a variable number that depends on $\mid n_k \mid$:

$$\begin{cases} \alpha_1^1 \lambda^1_{c_{type_1}} + \alpha_2^1 \lambda^1_{c_{type_2}} + ... + \alpha^1_{|C_{types}|} \lambda^1_{c_{type_{|C_{types}|}}} & = \tau \\ \alpha_1^2 \lambda^2_{c_{type_1}} + \alpha_2^2 \lambda^2_{c_{type_2}} + ... + \alpha^2_{|C_{types}|} \lambda^2_{c_{type_{|C_{types}|}}} & = \tau \\ ... \\ \alpha_1^{|N|} \lambda^{|N|}_{c_{type_1}} + \alpha_2^{|N|} \lambda^{|N|}_{c_{type_2}} + ... + \alpha^{|N|}_{|n_{|C_{types}|}|} \lambda^{|N|}_{c_{type_{|C_{types}|}}} & = \tau \end{cases} \tag{12}$$

where $\alpha_i^k = \sum \mid vn_j^k \mid$ is the sum of identical components running on node n_k and represents the unknowns in the system of equations. The system is particularly difficult to solve when nothing else is known about nodes. More precisely we need to know the type of nodes to be considered: homogeneous, uniform or unrelated nodes. The following paragraphs describe how the optimal number of components can be determined for each case.

Clouds usually expose their resources as virtual machines – called nodes in this paper – for which there is no restriction on the degree of heterogeneity.

We first consider a set of homogeneous nodes. Thus we have a number of heterogeneous components deployed on *homogeneous nodes*. Homogeneous nodes are usually used when users are interested in or afford a single type of resource.

In this scenario $\lambda^{k}_{c_{type_i}} = \lambda^{k'}_{c_{type_i}} = \lambda_{c_{type_i}}, \forall n_k, n'_k \in N$. By summing the equations from Relation 12 we get:

$$\mid N \mid \alpha_1 \lambda_{c_{type_1}} + \mid N \mid \alpha_2 \lambda_{c_{type_2}} + ... + \mid N \mid \alpha_{C_{types}} \lambda_{c_{\mid C_{types} \mid}} = \mid N \mid \tau \quad (13)$$

Hence we reduced the problem of finding the optimal number of components per *vn* to one of finding the optimal number of components per C_{type} for every node. The optimal value for this number can be found through nonlinear optimization. For this we need to find the function to be minimized. The function is easily determined as our goal is to balance the I/O throughputs of every component. This is done by minimizing the following equation based on the least squares method:

$$f = \sum_{i=1}^{|C_{types}|} \sum_{(i,j) \in \mathcal{D}} (r^{out}_{ij} \alpha_i - r^{in}_{ji} \alpha_j)^2 \quad (14)$$

with the following constraints:

$$\begin{cases} \sum_{i=1}^{|C_{types}|} \alpha_i \lambda_{c_{type_i}} = \tau \\ \alpha_i \geq 1, \forall i = \overline{1, \mid C_{types} \mid} \end{cases} \quad (15)$$

where the first constraint ensures that the node is fully loaded and the second one guarantees the HA of the application.

The function $f : \mathbb{R}^{|C_{types}|} \rightarrow \mathbb{R}$. As the solutions need to be integer values we truncate the α_i to their closest integer value.

The goal of Relation 14 is to find the optimal number of components such that the outgoing throughput of every component is completely consumed by the incoming throughput of the parter components. The ideal case is to have $f = 0$;

When a scenario with *uniform nodes* is considered the system of equations from Relation 12 can be reduced to:

$$\begin{cases} \alpha_1^1\lambda^1_{c_{type_1}} + \alpha_2^1\lambda^1_{c_{type_2}} + ... + \alpha^1_{|C_{types}|}\lambda^1_{c_{type_{|C_{types}|}}} & = \tau \\ \alpha_1^2\beta_1^2\lambda^1_{c_{type_1}} + \alpha_2^2\beta_2^2\lambda^1_{c_{type_2}} + ... + \alpha^1_{|C_{types}|}\beta^2_{|C_{types}|}\lambda^1_{c_{type_{|C_{types}|}}} & = \tau \\ ... & \\ \alpha_1^{|N|}\beta_1^{|N|}\lambda^1_{c_{type_1}} + \alpha_2^{|N|}\beta_2^{|N|}\lambda^1_{c_{type_2}} + ... + \alpha^{|N|}_{|n_{|C_{types}|}|}\beta^{|N|}_{|C_{types}|}\lambda^1_{c_{type_{|C_{types}|}}} & = \tau \end{cases} \tag{16}$$

where β_i^k represents the scaling factor that correlates the loads of each component on a reference node selected in this case to be n_1. From the system of constraints needed to find the optimal number of components (cf. Relation 15) it is only the first one that changes. By summing up the equations from Relation 16 the first constraint becomes:

$$\sum_{i=1}^{C_{types}} \left(\lambda_{c_{type_i}} \sum_{j=1}^{|N|} \alpha_i^j \beta_i^j \right) = \tau \tag{17}$$

The case of *unrelated nodes* is the most difficult to solve because it is impossible to determine a dependency between the equations in Relation 11. To overcome this problem we must treat each one independently by applying the nonlinear optimization for each equation independently. This gives the optimal number of components per type for every node. The sum of these values reflect the optimal number of components per type that can be placed on the platform.

4.2. Node Scaling

Node scaling usually occurs when the existing nodes cannot handle newly arrived components due to overloading. Ideally node allocation would occur only after the optimal number of components – as determined after minimizing Relation 14 – on every node has been reached.

In reality this approach inflicts useless delays, usually caused by node booting time in the web server's response time. One solution to this problem is to combine pro-active methods with reactive ones (as mentioned at the beginning of Sect. 4). This allows us to reserve nodes in advance through predictions on the hit rate and adjust these predictions on the spot by using the actual hit rate.

The efficiency of this approach is high when adjustments are minimal and the prediction is fairly accurate. As seen in Figs. 3(a) and 3(c) medium and

large traffic websites have cyclic patterns which can be easily predicted. In case of Fig. 3 predicted values are based on a feed forward neural network with five input and ten hidden elements. The X axis represents hours while the Y axis shows the number of page hits. Low traffic websites are more chaotic (cf. Fig. 3(b)) which makes predictions highly unreliable. A classic prediction method – such as the one based on a neural network – would only manage to catch the general trend of the traffic at best. The study of these prediction methods does not make the topic of this research. Our intention is only to point out the difficulty in predicting web hit rates as these affect the reaction time of the scaling algorithms.

Once the number of components needed for the next time frame $(t{+}1, t{+}2]$ has been determined based on the user hit rate, the algorithm decides how to scale the number of nodes.

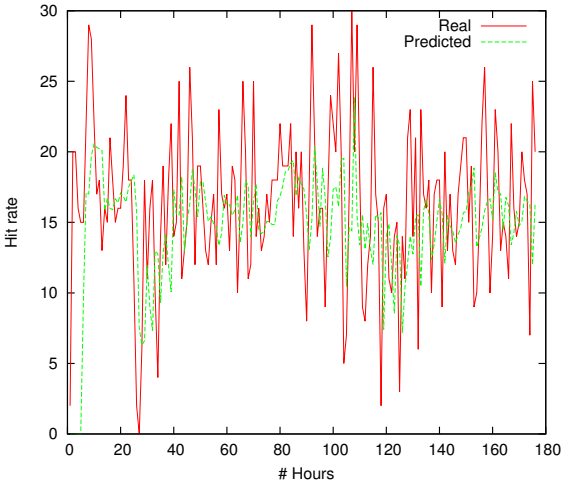
Figure 4 shows a simple algorithm that we propose for scaling nodes. The algorithm is a pro-active one and relies on the required number of components at a specific moment. The fine tuning of the needed number of nodes will be accomplished by the scheduling algorithms described in Sect. 4.3.

This algorithm relies on the difference between the current number of components and the number of required components at $t + 1$. Based on this value it decides if it should consider adding (Lines 1–10) or removing nodes (Lines 11–14). If the difference is positive then it checks if it can place the extra components on the existing nodes (Line 4). Providing that new nodes are needed it calculates the requirement and allocates the nodes (Line 7). If the difference is negative the algorithm determines the nodes to be removed based on the number of components on each node (Lines 11–15): the nodes are ordered by the number of components and the first *nodes_to_deallocate* are selected such that the sum of their components is approximately equal to the number of components not needed anymore.

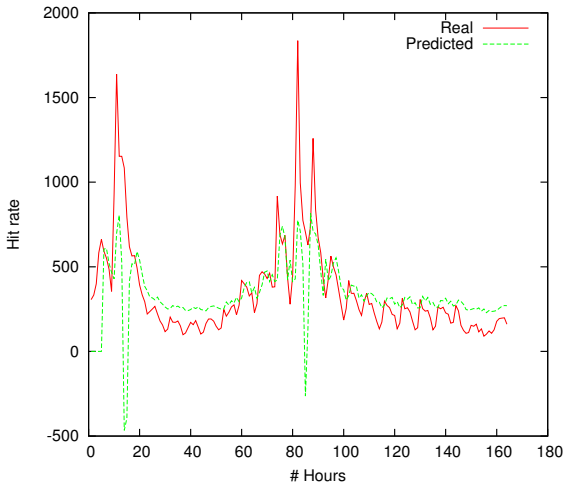
It can be noticed that this algorithm only allocates nodes and does not place components on them. Component allocation is handled by the algorithm presented in Sect. 4.3. The aim is to allocate nodes in advance in order to minimize the effects of the node startup in case of sudden changes.

4.3. Algorithm for Achieving Application High Availability

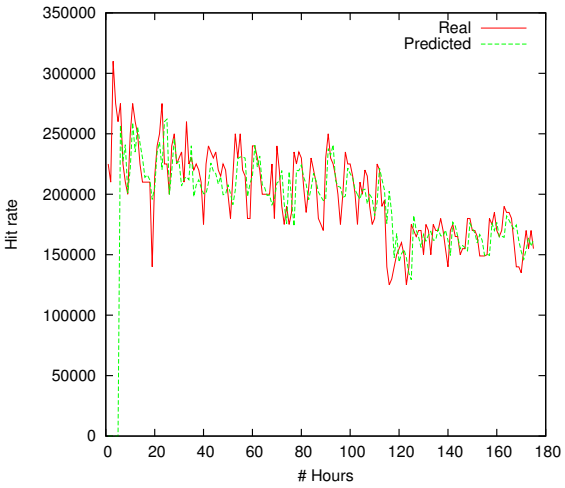
In this section we present two algorithms for achieving HA by carefully choosing where to (re)schedule components of certain types. The first leads to an optimal solution and can be used in case the load of every component



(a) Small page hit rate



(b) Medium page hit rate



(c) Large page hit rate

Figure 3: Predicted vs. real traffic using neural networks

type is known. The second produces a sub-optimal solution and relies on a GA to allocate components in case the component load is unknown.

4.3.1. Known Component Load

This scenario considers that the component load on any node type is known *a priori*. The HA algorithm then proceeds as follows:

Require: $C_{existing}(t)$ - the set of components existing in interval $(t, t+1]$
Require: $C_{needed}(t+1)$ - the set of components needed for interval $(t+1, t+2]$
Require: N - the set of nodes
Require: $nc\_new(t+1) =\mid C_{needed}(t+1) \mid - \mid C_{existing}(t) \mid$ - the number of new components needed for interval $(t+1, t+2]$

1: **if** $\langle nc\_new(t+1) > 0 \rangle$ **then**
2: $C_{new}(t+1) = C_{needed}(t+1) \setminus C_{existing}(t)$
3: **if** $\langle C_{new}(t+1) \neq \varnothing \rangle$ **then**
4: find subsets $C^j_{new}(t+1)$ such that $\mid \sum_{c_k \in C^j_{new}} \lambda_{c^j_{type_k}} - \left(\tau - \sum_{i=1}^{|C_{types}|} \alpha_i \lambda_{c^j_{type_i}}\right) \mid \rightarrow 0$
5: $C^{overload}_{new} = C_{new}(t+1) \setminus \bigcup C^j_{new}$
6: **if** $\langle C^{overload}_{new} \neq \varnothing \rangle$ **then**
7: create m new nodes such that $\mid \sum_{c_k \in C^{overload}_{new}} \lambda_{c^j_{type_k}} - m\tau \mid \rightarrow 0$
8: **end if**
9: **end if**
10: **end if**
11: **if** $\langle nc\_new(t+1) < 0 \rangle$ **then**
12: order nodes by number of components
13: sum up number of components of the first *nodes_to_deallocate*
14: stop when the sum given by adding the next node $>\mid nc\_new(t+1) \mid$
15: deallocate *nodes_to_deallocate* nodes
16: **end if**

Figure 4: Node scaling algorithm

The optimal number of components on each node is determined based on Relations 14 and 15. Afterwards the initial placement of each component will take place on a node that can accommodate a new instance of that particular component type. The node must at the same time provide the least cost as given by Relation 4. Once all nodes have been occupied or have become fully loaded and new components still require to be created the algorithm creates on the spot node instances. Figure 5 outlines the algorithm for the initial component placement. The key point in the algorithm is depicted on Line 3 which indicates the selection of the node that will host the new component. The selection is made by comparing the optimal number of components that can be placed on the node with the already existing components of the same type. If the optimal number has been reached on all existing nodes a new node will be created to accommodate the component.

The new node will be initially populated with randomly picked *vn*s (Lines 9–10) that can hold the new components (i.e., the optimal number of components plus the new one is not exceeded on the new node).

Once a new node is created it gets populated with new or existing components so that every type of component will find itself there. This process usually requires to relocate *vn*s from existing nodes. Deciding on which *vn* to relocate is however a difficult problem as it depends on a global cost function

```
1: N_new = ∅
2: while ⟨C_new ≠ ∅⟩ do
3:     select n_k ∈ N which can host c_i
4:     if ⟨ found n_k ⟩ then
5:         assign c_i ∈ C_new to vn_j ∈ n_k
6:         C_new = C_new \ {c_i}
7:     else
8:         create n_new
9:         randomly pick vn_j that can hold c_i on n_new
10:        move vn_j to c_new
11:        assign c_i ∈ C_new to vn_j
12:        N_new = N_new ⋃{n_new}
13:    end if
14: end while
```

Figure 5: Pseudocode of component initial component placement for the known component load scenario

```
1: Generate the set of identical initial schedules:
2: S ← {S_1, ..., S_n}
3: while ⟨the stopping condition is false⟩ do
4:     for i = 1,n do
5:         S'_i ←perturb(S_i)
6:     end for
7:     S ← select(S, {S'_1, ..., S'_n})
8: end while
```

Figure 6: Pseudocode of the genetic algorithm

which needs to be minimized. We opted for a GA as it allows us to explore a wide range of solutions. Applied iteratively mutations and crossovers can lead to improved results as the solution converges to a local sub-optimal solution. The larger the initial population the more likely that a significantly improved solution is obtained.

Figure 6 depicts the general structure of a GA. The uniqueness of each GA is given by the *perturb* subroutine (Line 5). Perturbations in the population are usually produced by element mutations through random relocations or swapping. As each perturbation induces extra communication costs – due to *vn* migrations – we restrict mutations from being obtained out of swapping.

The *perturb* version that we propose is depicted in Fig. 7. The idea is to migrate *vn*s of every existing component type from existing nodes to the newly created ones as long as the latter remain underloaded.

The GA stops when all the component types are present on the new node, when the node's load has been exceeded or when the maximum number of iterations has been reached.

```
1: for ⟨n_new ∈ N_new⟩ do
2:     while ⟨not all component types on n_new and λ_new ≤ τ and retries ≤ L_1⟩ do
3:         randomly pick vn holding components of type not already existing on n_new
4:         increment retries
5:     end while
6: end for
```

Figure 7: Pseudocode of the perturb(S_i) function in case the component load is known

4.3.2. Unknown Component Load

In case we cannot determine *a priori* the optimal number of components the node load λ_k can be used instead by comparing it with the maximum admissible load τ. The initial placement of newly generated components is done as follows: first the node which provides the least cost (cf. Relation 4) is selected; then the *vn* that holds the least number of components having the same type as the new one is picked. This is done to minimize the chance of leaving empty *vn*s in case of relocations.

The scheduling algorithm for this case is depicted in Fig. 8.

By starting from an element in the population we first determine the set of overloaded nodes (i.e., $\lambda_k > \tau$). For every such node n_k we try to find an existing node that can hold randomly picked *vn*s from n_k. If the node still remains overloaded after a number of attempts ($maxIterations_k < L_1$) we create a new node (n_new) spot instance. This will be populated with randomly picked *vn*s of every component type until $\lambda_k \leq \tau$. In case not every component type is present on n_{new} an attempt to migrate the necessary *vn*s from other nodes will be made while keeping $\lambda_{new} \leq \tau$.

If a mutated element holds all component types on every node it will replace the former one.

The GA stops once the *stopping condition* (cf. Fig. 6) is met: one or more elements have all component types on every node. If more than one such element exist the one that has the least cost attached to it is selected (cf. Relation 5). This selection is done by the *select* subroutine (Figure 6 Line 7).

The GA is executed every time new components are generated and placed on nodes.

It is clear from the model that *perturb* subroutines for the two algorithms (cf. Figs. 7 and 8) leave nodes underloaded. This is not an issue if we consider that having a node load under the maximum possible value can help when deallocating nodes as presented in Fig. 4.

```
1: for ⟨n_k : λ_k > τ⟩ do
2:     vn_j^k ← random vn from n_k
3:     n_m ← random node such that λ_m < τ considering also the load of vn_j^k
4:     if ⟨not found n_m⟩ then
5:         increment maxIterations_k
6:     else
7:         relocate vn_j^k to n_m
8:     end if
9:     if ⟨ maxIterations_k ≥ L_1⟩ then
10:        n_new ← newly allocated node
11:        maxIterations_new ← 0
12:        while ⟨λ_k > τ and λ_new ≤ τ⟩ do
13:            relocate random vn_j^k to n_new
14:        end while
15:        retries ← 0
16:        while ⟨not all component types on n_new and λ_new ≤ τ and retries ≤ L_2⟩ do
17:            randomly pick vn holding components of type not already existing on n_new
18:            increment retries
19:        end while
20:    end if
21:    if ⟨ λ_k < τ ⟩ then
22:        maxIterations_k ← 0
23:    end if
24: end for
```

Figure 8: Pseudocode of the perturb(S_i) function function in case the component load is unknown

4.3.3. Success Rate of the Algorithm

The *success rate of a system*, i.e., **system reliability**, represents the probability of obtaining "k" successes out of "n" trials [50]. In case of scheduling algorithms we define the schedule's π reliability $rel(\pi)$ as its probability to be successful. In our particular case this metric measures the capacity of algorithm to produce schedules in which each component type is present on every node so that each node remains underloaded.

In what follows we show how we can maximize $rel(\pi)$ in the context of this paper.

There are two major parts of the algorithm that have a direct impact on its reliability, mainly the allocation of new nodes and the mapping of components on existing nodes.

Given the creation of new nodes the first question that needs to be asked is what is the probability of having all component types on every newly instantiated n_{new}? Formally this can be expressed as:

$$Pr\Big[n_{new}^{C_{types}} =\mid C_{types} \mid\Big] = \prod_{c_{type} \in C_{types}} Pr\Big[\exists \mid vn_{c_{type}} \in n_{new} \mid > 0\Big] \qquad (18)$$

where the product indicates that the probability events need to take place simultaneously.

To ensure HA Relation 18 needs to equal 1. This indicates that we always have all component types on every allocated node.

After a new node n_{new} is created the algorithm requires to relocate *vn*s on n_{new}. These *vn*s come from overloaded nodes which need to increase their performance by reducing their load. A restriction on the relocation procedure is to maintain, after relocation, exactly $\mid C_{types} \mid$ component types on every preselected node n_{sel}. In other words we need to maximize the chance of having every *vn* populated with components. Empty *vn*s create the risk of leaving a node, after *vn* relocations, without the maximum number of component types. In order to achieve $rel(\pi) = 1$ we need to have $Pr\Big[n_{sel}^{C_{types}} =\mid C_{types} \mid\Big] = 1$.

Nodes are usually created when the existing ones are overloaded. This normally takes place when the probability of having overloaded nodes is large enough, i.e., $Pr\Big[\exists n_k \in N : \lambda_k > \tau\Big] \rightarrow 1$. Hence the scheduling algorithm must maximize its chance of having only underloaded nodes each consisting of $\mid C_{types} \mid$ component types. Based on this requirement we can compute $rel(\pi)$ as:

$$rel(\pi) = Pr\Big[\forall_{n_k} : n_k^{C_{types}} =\mid C_{types} \mid\Big]\Big(1 - Pr\Big[\exists n_k : \lambda_k > \tau\Big]\Big) \qquad (19)$$

In what follows we give weak and strong conditions for maximizing Relation 19 (i.e., $rel(\pi) \rightarrow 1$):

Condition 1. *(weak) Newly created components should always be placed on the least occupied vn.*

Condition 2. *(strong) The node load λ_k must not exceed the threshold before all $vn_i \in n_k$ are populated.*

It can be easily seen that *Condition 1* reduces the number of empty *vn*s due to allocations of new components. However this condition alone does not

ensure HA. The reason is that having both empty and non empty *vn*s, all of them accommodating the same c_{type}, could lead to the probable event of relocating only the populated *vn*s. This would cause n_{sel} to remain without components of c_{type} type which would impact on the application's HA.

To ensure the previous case does not happen *Condition 2* needs to be met as well. So node n_k will not affect the application's HA when *vn*s are relocated from it. For this reason we argue that *Condition 2* is strong while *Condition 1* is weak and ensures HA only for particular cases that imply the strong condition.

If none of the conditions is met the application's HA cannot be ensured, resulting a scenario similar with the one depicted in case *Condition 1* is not met.

5. Algorithm Evaluation

In this section we present the test scenarios as well as the results obtained on the proposed algorithms.

5.1. Test Scenario

To test the behavior of our algorithms we validated them against two scenarios one based on a real application graph and the other based on synthetically generated ones.

Scenario 1 used the Web 2.0 application structure as depicted in Fig. 1. The application consists of 7 component types: web server, message queue server, database and 4 modules for adding news, voting news, listing news and detecting spam. Except for the message queue server, every component can scale. The application only requires 12 message queues (two queues are needed for a bidirectional component-to-component communication) as indicated by Fig. 2. As each queue can store an arbitrary large number of messages – the number of queues being constant – there is no need for scaling the message server. The fail-safe for this component is achieved through a backup copy. Consequently we do not consider failures of the communication server. The characteristics of each component were simulated as described in Sect. 5.1.1.

A typical application use case proceeds as follows: Users start by accessing the web page through a browser. Depending on the HTTP request goal (e.g., to add, vote or list news) the request is placed in one of the corresponding message queues by the web server component. The receiver component

consumes messages and processes them. Every initial user request ends up as a query on the database component. From this point forward the response message follows the same route backwards until the user receives it in the form of a HTTP response message. It can be noticed that the two most heavily used components are represented by the database and the web server.

Scenario 2 used a wider range of randomly generated application graphs in order to test the efficiency and limits of our solution. Each graph was generated by using the *samepred* method in which each component can be linked to any existing component [51]. The number of components in a graph ranged from 10 to 60 in increments of 10. As shown later this interval suffices to determine the boundaries within which our algorithm can achieve HA.

A single type of connector exists for both scenarios and is represented by the input/output message queues.

We assume that *vn*s can only be migrated and not created at runtime. Hence when the application is first deployed all the required *vn*s are created – even though some are empty.

Each application is initially instantiated on a single node having the required number of components adapted based on the maximum user hit rate a single web server can cope with. To achieve HA we need to have every component type on each node. As a result we created a number of $\mid C_{types} \mid \times N_{max}$ *vn*s on the first node so that we have enough *vn*s to spread – in a worse case scenario – our entire application component set on all the nodes we are willing to rent.

5.1.1. Simulating Platform and Component Characteristics

An important aspect for performing the tests was to **simulate platform heterogeneity**. For this we varied the **component load** depending on the node type (i.e., homogeneous, uniform or unrelated):

For *homogeneous components and nodes* we used a load of $\lambda = 0.5$ for every component type. Thus every component had the same load on every allocated node: $\lambda_{c_i}^k = 0.5$.

For *heterogeneous components and homogeneous nodes* we used a normal variate $N(0.5, 0.25)$ to depict the load of each component type.

In the case of *heterogeneous components with uniform nodes* we assumed the node load changes as follows: for every n_k node uniformity is modeled by a uniform variate $U(0, 1)_k$ so that the load on each node is computed as:

$\lambda_{c_{type_j}} \times U(0,1)_k$, where $\lambda_{c_{type_j}}$ represents the load as obtained from a normal variate $N(0.5, 0.25)$.

The last case is represented by *heterogeneous components and unrelated nodes*. In this case we used a a uniform variate whose value was different for every component and node. Hence we obtained the load of a component of type c_{type_j} on node n_k to be $\lambda^k_{c_{type_j}} = \lambda_{c_{type_j}} \times U(0,1)$.

The maximum admitted load for every node was set to $\tau = 100$.

To compute the **cost of a component** we needed besides its load, the recurring and relocation costs. The former was computed based on the load it inflicts on the network (i.e., $\lambda_{c_{type_j}}/3$) and on its throughput, while the latter was computed based on its size which we assumed to be equal to its memory load (i.e, $\lambda_{c_{type_j}}/3$).

Component throughput was modeled individually for each scenario.

For Scenario 1 we used values as depicted in Table 1. The unit of measure is messages per second. Because web servers and databases usually handle large number connections we let them exhibit a similar behavior by allowing them to process twice as much as an application specific component (add, view, vote and detect spam). This permitted us to simulate the scaling of the web server and database as well. Because the application specific components used in the example are similar and do not perform complex operations we specified the same throughput rates for each of them. The only difference was that we set the produce rate to be lower than the consume one.

Scenario 2 used for the input rate a normal variate $N(175, 25)$ and for the output rate $N(160, 20)$. The output rate was intentionally set lower than the input one as we needed to simulate a non zero processing time for each message.

Throughput	c_1	c_2	c_3	c_4	c_5	c_6
consume rate	100	50	50	50	50	100
produce rate	100	40	40	40	40	100

Table 1: The consume/produce rate per component type in messages per second

The **user hit rate** also required to be modeled. Together with the components' throughputs we used it for determining the number of required components at any given moment. Two methods were employed to model the arrival rate:

First we used an extrapolation polynomial based on the user activity

within a 24h period [52]. Second we modeled user hit rate based on probabilistic methods. The work of Vicari [53] provides a starting point for describing HTTP session arrival rate and number of user connections per page. Both of these can modeled by Weibull/Lognormal respectively Pareto/Lognormal distributions. For our tests we used a Weibull distribution having a shape parameter of 0.17 and a scale parameter of 0.60. A Pareto distribution with a shape of 1.75 and a minimum value of 1.68 was also considered.

5.1.2. Test Goals

To test the algorithm we aimed at studying its behavior relative to two key aspects that had been discussed in this paper. The first one, detailed in Sect. 5.2.1, is linked to the algorithm's ability to achieve HA. For this we wanted to see if the algorithm is capable of distributing every component type on every allocated node. The algorithm's reliability (cf. Sect. 4.3.3) is a strong indicator of this aspect.

The second aspect, studied in Sect. 5.2.3, is related to the heterogeneity of the produced load on every node. A homogeneous load indicates nodes to be used uniformly without over or under utilized resources. If the load is close to the resource maximum capacity then we can assume that the resources are fully utilized and also that none is wasted.

The tests focused on the behavior of the algorithm version for unknown component loads. The motive behind this choice was that in the case of known component loads we have an algorithm for obtaining the optimal solution. Intuitively the algorithm for unknown component loads is more chaotic due to its random relocation of *vn*s and is consequently of greater interest. Nonetheless to get a glimpse of how results vary in this case from the optimal allocation we show the allocation difference for *Scenario 1* (as seen in Sect. 5.2.2).

For each test a total of 10 experiments were executed and the average result was taken. Results were compared with a Round Robin algorithm in use in many commercial cloud systems.

5.2. Test Results

5.2.1. Reliability Test

For *Scenario 1* the results for the reliability tests are depicted in Table 2. The first column presents the model for generating the user hit rate as described in Sect. 5.1. The reliability is computed based on Relation 19. To determine it we generated 10 experiments and randomly picked all but

one node to fail. The existence of every component type on the remaining node was checked and the experiments in which this happens were counted. The value for $rel(\pi)$ was then computed as the ratio between the number of events in which the remaining node contained all component types over the total number of experiments.

Table 2 shows the results for the case of heterogeneous component types and unrelated nodes. The first aspect we were interested in was to check the validity of *Condition 1* – for maximizing reliability. It can be easily noticed that the case in which the least occupied *vn* is selected for component allocation (*polynomial* $\times$ *10 least loaded vn*) gives a better reliability (**0.98** $\pm$ 0.05) than the case in which the most loaded virtual node (*polynomial* $\times$ *10 most loaded vn*) is picked as destination **0.84** $\pm$ 0.12.

Another aspect that we discovered was the fluctuation of the reliability in case small and medium number of components are allocated (*polynomial* and *Pareto*). This behavior can be explained by the fact that in this case the number of database and web server instances is smaller than the number of allocated nodes. Hence the probability to remain without any of those is higher in case of failures.

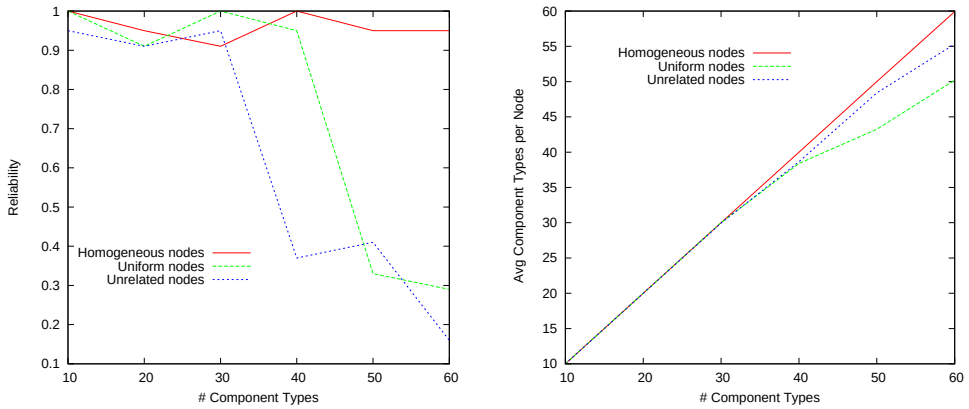
For large number of components (*Weibull*) the algorithm's reliability is more than twice as good as the one offered by Round Robin. The same is true for a medium sized number of components (both *polynomial* $\times$ *10* cases). It must be noted however that the reliability in case of the Round Robin is similar with that of our approach when a small number of components is generated. The reason behind this behavior could be the circular distribution of components implied by Round Robin. In case of small number of components (and subsequently nodes) this allocation method could lead to a uniform spread of component types per node. The intuition dictates that the same is true for large number of components and nodes. Yet this behavior is not observed in this case. The reason for this is still under debate but it could have to do with the way in which we adapted Round Robin to our model.

Overall our proposed solution provides a higher and more stable reliability than that of Round Robin, a fact attributed to the design of our algorithm that is targeted for obtaining HA.

In order to check whether or not our algorithm has a limit on the component types beyond which its reliability degrades we performed tests by using *Scenario 2*. Results for a hit rate modeled by the extrapolation polynomial are depicted in Fig. 9. It can be noticed that the only case in which the

Hit Rate Model	# components	$rel(\pi_{Our})$	$rel(\pi_{RoundRobin})$
polynomial × 10 most loaded vn	922	**0.84** ± 0.12	0.38 ± 0.11
polynomial × 10 least loaded vn	922	**0.98** ± 0.05	0.41 ± 0.05
polynomial	81	0.89 ± 0.04	**0.96**
Pareto	65	**0.96** ± 0.07	0.95 ± 0.11
Weibull	18,988	**0.96** ± 0.06	0.41 ± 0.29

Table 2: Reliability results for the case of heterogeneous component types and unrelated nodes



(a) Reliability

(b) Average number of component types per node

Figure 9: Reliability results for Scenario 2

reliability is kept stable and close to 1 is that of *heterogeneous components and homogeneous nodes*. In this case the average reliability of 0.96 indicates that in more than 90% of experiments we achieve HA. For the rest of the cases the reliability drops abruptly when more than 30 (*heterogeneous components with uniform nodes*), respectively 40 (*heterogeneous components with unrelated nodes*) component types are used in the application. The reason for this is that when uniform and unrelated nodes are used the chances of not being able to find space to fit the load of a missing component type are higher especially if we allocate less powerful machines than the one we relocate from. The large number of heterogeneous components that need to fit inside every node also contributes to this aspect. However the fact that we have achieved good results for the case of homogeneous nodes is promising

taking into account the fact that a customer usually rents a single type of resources from a cloud provider.

Conclusively we can argue for this scenario that when uniform or unrelated nodes are used applications should not exceed more than 30 or 40 components as this represents the limit beyond which the algorithm reliability starts to decrease rapidly. For homogeneous nodes however this limit is pushed beyond 60 components.

Figure 9(b) indicates the average number of component types per node in the three mentioned cases. Ideally the line should have a slope of 1 as in the case of homogeneous nodes. Still the number of missing component types ranges from below 8% for unrelated nodes to 17% for uniform nodes. The reason for the higher values obtained for uniform nodes is that given the same amount of components we have less nodes assigned in the case of uniform nodes.

5.2.2. Optimality Test

To test the efficiency and outline the differences of the GA algorithm (cf. Sect. 4.3.2) from the optimal solution we compared it based on *Scenario 1* against the optimal values obtained by solving Relation 14 under the constraints from Relation 15.

For our testing case, through expansion and replacement, Relation 14 becomes:

$$\begin{aligned} f =& 600\alpha_1^2 + 82\alpha_2^2 + 83\alpha_3^2 + 82\alpha_4^2 + 82\alpha_5^2 + 600\alpha_6^2 - \\ & 180\alpha_1\alpha_2 - 180\alpha_1\alpha_3 - 180\alpha_1\alpha_4 - 80\alpha_2\alpha_5 - \\ & 180 * \alpha_5\alpha_6 - 180\alpha_3\alpha_6 - 180\alpha_4\alpha_6 \end{aligned} \tag{20}$$

where the upper coefficients represent the powers of the α_i variables.

As the results were similar we only depict in what follows the values obtained by generating components based on the extrapolation polynomial (cf. Sect. 5.1).

In case of homogeneous components and nodes the set of constraints becomes:

$$\begin{cases} \sum_{i=1}^{6} 0.5\alpha_i = 100 \\ \alpha_i > 1 \end{cases} \tag{21}$$

Solving this nonlinear system gives (by truncating the double values to integers) a solution vector of : $\langle\alpha\rangle = \langle 18, 40, 40, 41, 40, 18\rangle$. The solutions represent the optimal number of component instances for every existing type. A total of 10 nodes have been allocated during tests for this case. Figure 10 depicts the difference between the number of optimal components and the ones that were actually allocated (i.e., *optimal allocation difference*). A positive difference indicates that the actual number of components is smaller than the optimal number by exactly the value indicated on the Y axis for every component type. In contrast negative values show that the number of instances for the respective component types have exceeded the optimal values.

The large positive values in Fig. 10 indicate that the average node can accommodate more components without exceeding their load threshold and more importantly that the number of nodes could be reduced by migrating components such that the nodes' loads would reach the maximum limit τ. However this is not usually possible as components migrate in groups (i.e., *vn*s) and finding the best relocation configuration is difficult to obtain.

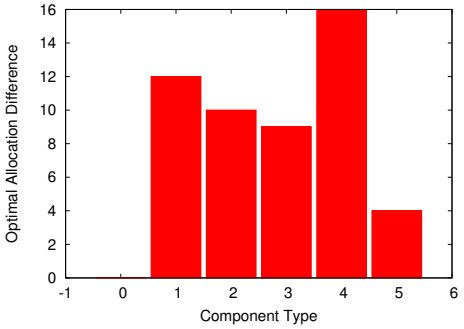


Figure 10: Optimal allocation difference in case of homogeneous components and nodes

When heterogeneous components with homogeneous nodes are considered the results are similar with those of homogeneous components and nodes. In this case we obtain a total of 12 allocated nodes and the following optimal solution vector
$\langle\alpha\rangle = \langle 5, 12, 12, 12, 12, 5\rangle$. The optimal allocation difference for this case can be seen in Fig. 11. As it may be noticed, in this case the algorithm provides a solution that is closer to the optimal values. Only the first component type is slightly over allocated but this exception is counterbalanced by the under allocation found in the case of the other component types. The reason why

the algorithm produces solutions that are closer to the optimal one can be linked to the heterogeneity of the components which allow a greater number of combinations to take up more of each node's load.

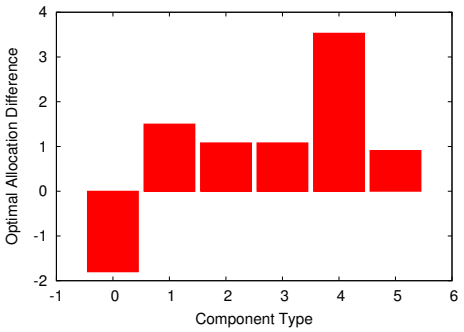


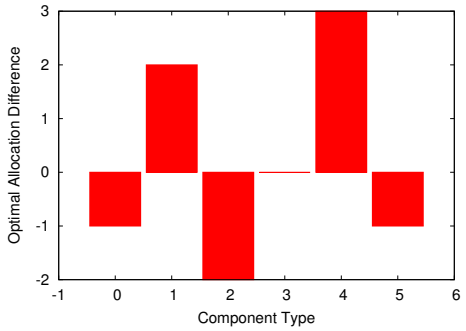
Figure 11: Optimal allocation difference in case of heterogeneous components with homogeneous nodes

For determining the optimal number of components in case of heterogeneous components with uniform nodes we solve the nonlinear optimization problem for each equation in Relation 16. The obtained result gives the optimal solution matrix for each of the three allocated nodes:

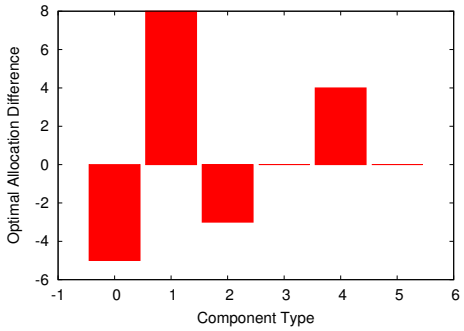
$$\begin{pmatrix} \alpha_i^1 \\ \alpha_i^2 \\ \alpha_i^3 \end{pmatrix} = \begin{pmatrix} 8 & 17 & 17 & 17 & 17 & 8 \\ 14 & 31 & 30 & 31 & 31 & 14 \\ 69 & 154 & 152 & 155 & 154 & 69 \end{pmatrix} \tag{22}$$

Figure 12 presents the optimal allocation difference in case of uniform nodes. Because every node has a different number of optimal components (cf. Relation 16) each subfigure represents the allocation difference on a distinct node. It can be noticed the highly under-allocated node 3 in Fig. 12(c). Although this is apparently a problem in the allocation algorithm a quick look at the third row in the optimal solution matrix from Relation 22 provides the answer. Node 3 is a powerful machine that can hold a large number of components. As the total number of components allocated in this experiment is of 81 we quickly notice that there are simply insufficient components to load three nodes close to the threshold τ (cf. Fig. 14(c)).

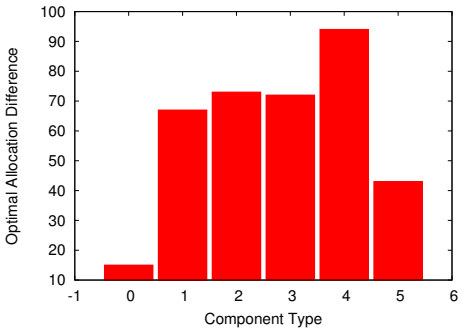
The case of heterogeneous components with unrelated nodes provides the following optimal solution matrix for the four allocated nodes:



(a) Node 1



(b) Node 2



(c) Node 3

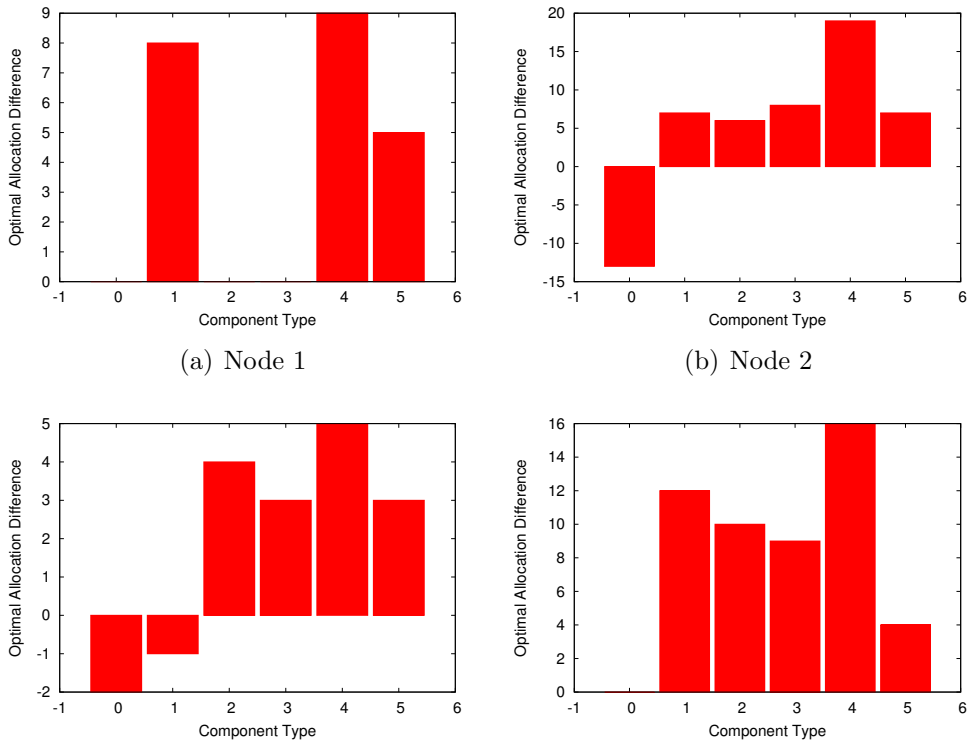
Figure 12: Optimal allocation difference in case of heterogeneous components and uniform nodes

$$\begin{pmatrix} \alpha_i^1 \\ \alpha_i^2 \\ \alpha_i^3 \\ \alpha_i^4 \end{pmatrix} = \begin{pmatrix} 23 & 52 & 50 & 51 & 51 & 23 \\ 24 & 53 & 55 & 54 & 54 & 25 \\ 10 & 23 & 23 & 23 & 22 & 10 \\ 10 & 23 & 23 & 23 & 23 & 10 \end{pmatrix} \tag{23}$$

Figure 13 shows the optimal allocation difference for the case of unrelated nodes. Results are similar with the uniform nodes case (cf. Fig. 12(c)). It can be noticed that node 4 is underloaded due to the small number of generated components.

5.2.3. Load Node Test

In this section we test and describe the evolution of the load during the experiments for our two scenarios. Results for *Scenario 1* are depicted



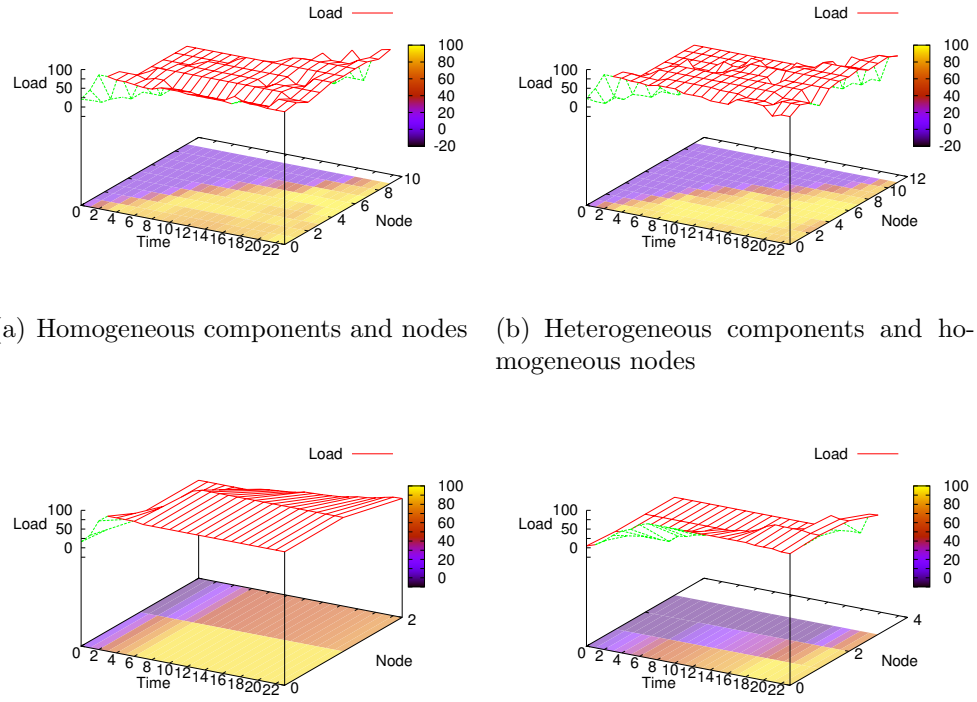
(a) Node 1 (b) Node 2

(c) Node 3 (d) Node 4

Figure 13: Optimal allocation difference in case of heterogeneous components and unrelated nodes

in Figure 14. We show the load variation due to node allocations and *vn* relocations. Dark cells indicate that the node has not been allocated yet. We were mainly interested in the evolution of the load due to the constant arrival of new tasks and consequently we did not study the deallocation problem. The reason was that deallocation implies a simple node removal including its components which does not influence the load of other nodes (cf. Fig. 4).

The strongest variation seems to occur when using heterogeneous components and homogeneous loads (cf. Fig. 14(b)). The average load has had a value of 86.25 ± 10.54 which indicates that although some nodes are fully loaded others remain with approximately a quarter of their resources unused. The main reason for this behavior has been identified in Sect 5.2.2 as being caused by relocations of *vn*s. Having heterogeneous components deployed on



(a) Homogeneous components and nodes

(b) Heterogeneous components and homogeneous nodes

(c) Heterogeneous components and uniform nodes

(d) Heterogeneous components and unrelated nodes

Figure 14: Average node load for Scenario 1

homogeneous nodes reduces the diversity in the *vn* load and thus diminishes the chances of finding suitable places for relocation. Therefore the algorithm has difficulties in selecting *vn*s that match the empty unused load slots during the relocation process.

The same behavior can be observed for the case of homogeneous components and nodes. In this case the average load is of 93.2 ± 5.86. The variation is smaller in this case but the reason behind it is the same as the one indicated in the previous paragraph.

The last two cases involving uniform (cf. Fig. 14(c)) and unrelated nodes (cf. Fig. 14(d)) clearly depict the allocation process which takes place only after the existing nodes have reached their load threshold. After that the allocation of new components takes place only on the new node leaving the

existing ones working at full capacity. Despite the small number of allocated nodes we can safely assume that the overall load evolution will follow patterns similar with those found in the two cases involving homogeneous nodes. Figures 14(c) and 14(d) also provide a reason for the high optimal allocation difference found in case of nodes three – uniform nodes (cf. Fig. 12(c)) – and four – unrelated nodes (cf. Fig. 13(d)): mainly that the difference is caused by the low loads on these nodes (55 respectively 56).

Figure 15 presents the average load obtained during tests for *Scenario 2*. As it can be seen the load varies between 82 and 96 out of 100. As it is undesired to fully load a node because it diminishes its efficiency a load in the given range is sufficient not to over/under utilize the node. Like in the reliability tests (cf. Sect. 5.2.1) we notice that the case when *heterogeneous components and homogeneous nodes* are used gives the most homogeneous and higher load: 92.98 ± 1.68. The larger variations observed in case of uniform and unrelated nodes is expected due to the difficulty of finding good candidates for fitting *vn*s.

Results have shown that the algorithm produces high loads and for some cases even homogeneous ones. This is a good indicator of the fact that for some cases nodes are used at their full potential.

6. Conclusions

This paper has addressed the problem of deploying HA long running applications on cloud systems. For component based applications, such as Web 2.0 ones, this goal is difficult to achieve as several criteria need to be considered. These include: scaling the application on the allocated nodes in order to manage the incoming user requests; keeping nodes within a given load threshold; and minimizing the application cost. In this paper we provided a solution for achieving HA distinct from the classic $N+M$ approach, by deploying every application component type on each node. Hence we aimed not only at minimizing node use by maximizing node load but also at keeping the application running even if all but one node have failed.

We have proposed two algorithms for achieving application HA. The optimal algorithm is presented as a reference model and can be used in case the load requirement for every component type is known. As an alternative in case the loads cannot be determined a sub-optimal algorithm is also introduced. To prove the efficiency of the proposed methods we have described and tested their reliability. Tests on the distance of the sub-optimal schedules

from the optimal one as well as on the node load distribution have also been presented. Results have shown the efficiency of the algorithms especially when considering heterogeneous components running on top of homogeneous nodes. For heterogeneous nodes the algorithm still behaves well but experiences a sudden drop in reliability after more than 30 components are used per application.

Future work involves integrating the algorithms inside the mOSAIC platform and their use as scheduling policies for the intended multi-cloud middleware.

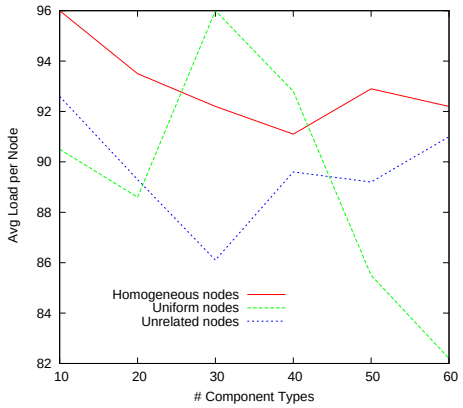


Figure 15: Average node load for Scenario 2

Acknowledgment

The work of the author has been partially funded by the European FP7-ICT project mOSAIC grant no. 256910 and by the Romanian AMICAS project PN-II-ID-PCE-2011-3-0260.

References

[1] Amazon, Amazon elastic compute cloud (ec2), (accessed Feb 14 2012) (2012).
URL `http://aws.amazon.com/ec2/`

[2] Rackspace, Rackspace, (accessed Feb 14 2012) (2012).
URL `http://www.rackspace.com/`

[3] Google, Google appengine, (accessed Feb 13 2012) (2012).
URL http://code.google.com/appengine/

[4] Microsoft, Windows azure, (accessed Feb 14 2012) (2012).
URL http://www.windowsazure.com/en-us/

[5] I. Eucalyptus Systems, Eucalyptus cloud computing software, (accessed Feb 14 2012) (2012).
URL http://www.eucalyptus.com/

[6] R. Hosting, NASA, Openstack, (accessed Feb 14 2012) (2012).
URL http://openstack.org/

[7] D. Petcu, C. Craciun, N. Neagul, M. Rak, 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.

[8] J. Kupferman, J. Silverman, P. Jara, J. Browne, Scaling into the cloud, (available on-line accessed Jan 29 2012) (2009).
URL http://cs.ucsb.edu/ jkupferman/docs/ScalingIntoTheClouds.pdf

[9] J. Gray, D. P. Siewiorek, High-availability computer systems, Computer 24 (2002) 39–48.

[10] N. Bobroff, A. Kochut, K. Beaty, Dynamic placement of virtual machines for managing sla violations., in: Integrated Network Management, IEEE, 2007, pp. 119–128.

[11] VMware, Vmware high availability, (accessed Feb 15 2012).
URL http://www.vmware.com/files/pdf/ha_datasheet.pdf

[12] A. B. Nagarajan, F. Mueller, C. Engelmann, S. L. Scott, Proactive fault tolerance for hpc with xen virtualization, in: Proceedings of the 21st annual international conference on Supercomputing, ICS '07, ACM, New York, NY, USA, 2007, pp. 23–32.

[13] A. S. Foundation, Apache module mod_proxy_balancer, (accessed Feb 15 2012).

URL http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html

[14] Novell, Suse linux enterprise high availability extension, (accessed Feb 14 2012) (2002).
URL http://www.ultramonkey.org/

[15] RedHat, Piranha, ip load balancing, (accessed Feb 14 2012) (2005).
URL http://www.redhat.com/software/rha/cluster/piranha/

[16] Novell, Suse linux enterprise high availability extension, (accessed Feb 14 2012) (2011).
URL http://www.suse.com/documentation/sle_ha/index.html

[17] S. Loveland, E. M. Dow, F. LeFevre, D. Beyer, P. F. Chan, Leveraging virtualization to optimize high-availability system configurations, IBM Syst. J. 47 (2008) 591–604.

[18] M. A. Oracle, Mysql cluster ndb 6.x/7.x reference guide, (accessed Feb 15 2012).
URL http://dev.mysql.com/doc/#cluster

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: amazon's highly available key-value store, SIGOPS Oper. Syst. Rev. 41 (2007) 205–220.

[20] F. Machida, M. Kawato, Y. Maeno, Redundant virtual machine placement for fault-tolerant consolidated server clusters., in: NOMS, IEEE, 2010, pp. 32–39.

[21] B. Korte, J. Vygen, Combinatorial Optimization: Theory and Algorithms, 4th Edition, Springer Publishing Company, Incorporated, 2007.

[22] N. R. Gottumukkala, C. B. Leangsuksun, N. Taerat, R. Nassar, S. L. Scott, Reliability-aware resource allocation in hpc systems, in: Proceedings of the 2007 IEEE International Conference on Cluster Computing, CLUSTER '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 312–321.

[23] N. Gottumukkala, C. Leangsuksun, R. Nassar, M. Paun, D. Sule, S. L. Scott, Reliability-aware optimal k-node allocation of parallel applications in large scale hpc systems, in: Proceedings of the 2008 High Availability and Performance Computing Workshop, HAPCW '08, 2008.
URL http://www.hpcsw.org/presentations/workshops/high_availability/105.pdf

[24] N. Aggarwal, P. Ranganathan, N. P. Jouppi, J. E. Smith, Configurable isolation: building high availability systems with commodity multi-core processors, SIGARCH Comput. Archit. News 35 (2007) 470–481.

[25] IBM, Cloudburst, (accessed Feb 14 2012) (2012).
URL http://www-01.ibm.com/software/tivoli/products/cloudburst/

[26] M. Mao, J. Li, 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.

[27] S. Chaisiri, B.-S. Lee, 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.

[28] R. Van den Bossche, K. Vanmechelen, 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, CLOUD '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 228–235.

[29] Y. Zhang, G. Huang, X. Liu, H. Mei, Integrating resource consumption and allocation for infrastructure resources on-demand, in: Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing, CLOUD '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 75–82.

[30] Y. O. Yazir, C. Matthews, R. Farahbod, S. Neville, A. Guitouni, S. Ganti, 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, CLOUD '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 91–98.

[31] H. Zhong, K. Tao, X. Zhang, An approach to optimized resource scheduling algorithm for open-source cloud systems, in: Proceedings of the The Fifth Annual ChinaGrid Conference, CHINAGRID '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 124–129.

[32] S. Tayal, Tasks scheduling optimization for the cloud computing systems, IJAEST 5 (2011) 111–115.

[33] J. Gu, J. Hu, T. Zhao, G. Sun, A new resource scheduling strategy based on genetic algorithm in cloud computing environment, Journal of Computers 7 (2012) 42–52.

[34] C. Zhao, S. Zhang, Q. Liu, J. Xie, J. Hu, Independent tasks scheduling based on genetic algorithm in cloud computing, 2009 5th International Conference on Wireless Communications Networking and Mobile Computing (2009) 1–4.

[35] A.-M. Oprescu, T. Kielmann, H. Leahu, Budget estimation and control for bag-of-tasks scheduling in clouds., Parallel Processing Letters 21 (2) (2011) 219–243.

[36] T. A. Henzinger, A. V. Singh, V. Singh, T. Wies, D. Zufferey, Flexprice: Flexible provisioning of resources in a cloud environment, in: Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing, CLOUD '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 83–90.

[37] T. D. Braun, H. J. Siegel, 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 61 (6) (2001) 801–837.

[38] F. Zamfirache, M. Frîncu, 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, Vol. 6046 of Lecture Notes in Computer Science, Springer-Verlag, 2011, pp. 321–328.

[39] O. P. Leads, Opennebula, (accessed Feb 15 2012).
URL http://opennebula.org/

[40] U. of Chicago, Nimbus, (accessed Feb 15 2012).
URL http://www.nimbusproject.org/

[41] P. Mousumi, S. Debabrata, S. Goutam, Dynamic job scheduling in cloud computing based on horizontal load balancing, International Journal of Computer Technology and Applications 2 (5) (2011) 1552–1556.

[42] A. M. Q. Protocol, Amqp 1.0 specification, http://www.amqp.org/resources/download (accessed Oct 21 2011) (October 2011).

[43] Microsoft, Microsoft windows server 2003 tech center, (accessed Feb 16 2012).
URL http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/31a2f39c-4d59-4cba-905c-60e7af657e49.mspx?mfr=true

[44] A. S. Foundation, Apache core features, (accessed Feb 16 2012).
URL http://httpd.apache.org/docs/2.0/mod/core.html

[45] R. inc., http://www.rightscale.com (accessed Oct 27 2011) (2009).

[46] A. Chandra, W. Gong, P. Shenoy, Dynamic resource allocation for shared data centers using online measurements, in: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, SIGMETRICS '03, ACM, New York, NY, USA, 2003, pp. 300–301.

[47] E. Caron, F. Desprez, A. Muresan, Forecasting for grid and cloud computing on-demand resources based on pattern matching, Cloud Computing Technology and Science, IEEE International Conference on 0 (2010) 456–463.

[48] M. Finger, G. C. Bezerra, D. R. Conde, Resource use pattern analysis for predicting resource availability in opportunistic grids, Concurr. Comput. : Pract. Exper. 22 (2010) 295–313.

[49] S. Islam, J. Keung, K. Lee, A. Liu, Empirical prediction models for adaptive resource provisioning in the cloud, Future Generation Computer Systems 28 (1) (2012) 155 – 162.

[50] M. Rausand, A. Høyland, System reliability theory: models, statistical methods, and applications, Wiley series in probability and statistics: Applied probability and statistics, Wiley-Interscience, 2004.

[51] T. Tobita, H. Kasahara, A standard task graph set for fair evaluation of multiprocessor scheduling algorithms, Journal of Scheduling 5 (5) (2002) 379–394.

[52] D. G. Feitelson, Workload modeling for computer systems performance evaluation (February 2011).
URL `http://www.cs.huji.ac.il/ feit/wlmod/`

[53] N. Vicari, Modeling of internet traffic: Internet access influence, user interference, and tcp behavior, Ph.D. thesis, University of Würzburg (4 2003).