QBROKAGE: A Genetic Approach for QoS Cloud Brokering

Gaetano F. Anastasi, Emanuele Carlini, Massimo Coppola, Patrizio Dazzi
Information Science and Technologies Institute
ISTI-CNR
Pisa, Italy
{gaetano.anastasi,emanuele.carlini,massimo.coppola,patrizio.dazzi}@isti.cnr.it

Abstract—The broad diffusion of the Cloud Computing paradigm has fostered the proliferation of a large number of cloud computing providers. In such a context, the need of Cloud Brokers arises for helping consumers in discovering, considering and comparing services with different capabilities as offered by different providers. Also, consuming services coming by different providers, when possible, may help to overcome vendor lock-in problems. While it can be straightforward choosing the best provider when deploying small and homogeneous applications, things get harder if the size and complexity of the application grow up. In this paper we propose a genetic approach for Cloud Brokering, focusing on finding Infrastructure-as-a-Service (IaaS) resources for satisfying Quality of Service (QoS) parameters of applications. By leveraging an implementation of such broker and a Cloud simulator, we performed some experiments for showing that our broker can find near-optimal solutions even when coping with hundred of providers, trying at the same time to mitigate the vendor lock-in problem.

Keywords—Cloud Broker; Cloud Brokering; QoS; genetic.

I. INTRODUCTION

Cloud Computing is nowadays one of the most popular computational paradigm. It has been adopted by many companies and considered by many more others for the unquestionable benefits offered, such as potential cost reductions offered by the pay-per-use model, flexibility and increased availability for the geographic distribution of resources.

However, most Cloud providers force their customers to use proprietary interfaces, virtualization technologies, communication protocols and so on. This may lead to an high degree of vendor lock-in that results from the burdens related to moving data to another provider and the difficulties to modify applications for leveraging other technologies coming from different service providers. For protecting themselves against vendor lock-in, some small to mid-sized businesses (SMBs) may do underinvestment or hesitate to use cloud computing. Recent surveys [7] also point out that some SMBs are forgoing Cloud Computing because of security and trust reasons, being afraid of losing control of their data, worrying about reliability, integrity and compliance with data privacy laws.

In our opinion, at least two orthogonal approaches can be applied for addressing such problems: Cloud Brokering and Cloud Federation [2]. In fact, both approaches can overcome the vendor lock-in problem, allowing users to exploit different providers by means of abstraction API such as Apache Libcloud1 (common case for Cloud Brokering) or by means of a standard platform each provider must be compliant to (common case for Cloud Federation). Even if Cloud Federation may subsume the Cloud Brokering approach, they can be considered orthogonal for the goal: for correctness a Cloud Broker should always consider user profits neglecting provider ones, whilst the Cloud Federation must operate a trade-off between these two apparent discording objectives, for example ensuring fairness in exploiting resources belonging to the federated providers.

Moreover, such approaches can help to overcome the trust problem in adopting Cloud Computing, for example by searching for providers that fit the user security needs in submitting particular applications (e.g. the provider location for ensuring law compliance in data management) or by explicitly addressing Quality of Protection (QoP) as a special case of Quality of Service (QoS), as done in the Contrail approach to Cloud Federation [4]. In such a context, one of the research challenge to be faced regards the problem of scheduling complex applications by respecting user constraints and, at the same time, avoid the issue of vendor lock-in. Another aspect to consider is the number of worldwide providers. For instance, Spamina Inc. lists around 850 providers worldwide on its web site2. While it can be considered acceptable to manually search and deploy an applications on handful of providers, this task becomes unfeasible when the number of providers grows up to hundreds.

In this paper we present QBROKAGE, a Cloud Brokering approach aimed to resolve the scalability and vendor lock-in issues. QBROKAGE exploits only the information that commercial providers are likely to made available for customers, such as Virtual Machine (VM) costs and characteristics in term of storage, memory, etc. Let us consider a scenario in which customers submit applications to QBROKAGE requesting for a deployment solution that meets QoS requirements, that could be formally expressed by Service Level Agree-

1http://libcloud.apache.org/
2http://www.spamina.com/eng/cloud_hosting_providers_list.php
ments (SLAs). Such requirements may involve both non-functional aspects, such as security capabilities of providers, and functional aspects as coming from other specification formats, such as the Open Virtualization Format (OVF [10]). For example, applications requirements may include that VMs need at least a certain quantity of memory, and a minimum number of physical CPUs, along with the exact match of geographic location where to place specific parts of the application. Such requirements are used as constraints by QBROKAGE for choosing a set of Cloud providers that can host the services (appliances) and at the same time guaranteeing the respect of the QoS negotiated for the application as a whole.

The main contribution of this paper is the proposal of a genetic approach for Cloud Brokering, designed and implemented by keeping in mind the following software requirements:

- meeting the heterogeneous QoS requirements of applications;
- finding near-optimal solution according to customers preferences trying at the same time to avoid vendor lock-in;
- supporting providers with different cost models;
- scaling up with hundreds of providers, while maintaining interactivity.

The model proposed in this paper is presented in Section III. The broker architecture and an insight on the algorithm are given respectively in Section IV and Section V. This paper also provide an experimental evaluation of QBROKAGE by means of simulations (Section VI), including a comparison with a state of the art approach, tuning of the genetic algorithm, scalability performances, and the ability to avoid vendor lock-in. In the following section we present our work with respect to the state of the art.

II. RELATED WORK

In the research community there is a wide consensus on the importance that brokers can have on Cloud environments for helping consumers in discovering, considering and comparing services with different capabilities as offered by different providers [12]. The need of brokering mechanism particularly arises in Cloud Federation architectures, such as Intercloud [2], the first approach going in the direction of building a unified platform composed by federated providers that can exchange information through super-entities (e.g. the Contrail approach) or as peers (e.g. the Sky [6] approach). Recently, Cloud Brokering architectures are acquiring importance by themselves, for dealing with providers that are loosely-coupled or not coupled at all. STRATOS [11] is a cloud broker service that permits to deploy and manage cloud applications on multiple providers, based on requirements specified in higher level objectives. This work has many similarities with ours and they are compared in Section VI by considering a scenario that was possible to reproduce. One of the most notable difference between the two works is the formalization of the problem, STRATOS embodies a multi-criteria optimization problem, whilst our work is based on genetic algorithms. Another remarkable difference regards the application description: STRATOS uses a custom XML representation for describing the application as a set of clusters and nodes, whilst our broker accepts OVF [10] description as input and constructs from it a graph representation of the application, in terms of nodes and edges. Since OVF is a promising standard description approach, we believe this capability may help in the adoption of QBROKAGE. Finally, STRATOS supports elasticity allowing for adding VMs to running applications, whilst our work does not address this aspect.

Ngan and Kanagasabai propose a semantic cloud broker [8] based on ontology matching that can cope with semantic interoperability issues caused by different non-standard ways of exposing provider capabilities. Focusing on the same approach, the authors propose a benchmark framework for cloud brokers [9], based on five different level of difficulty. Our work does not use ontology matching since we are focusing on QoS parameters that can be quantifiable and uniformed quite easily. Nevertheless, we believe that our work could be complemented with such approach for considering semantic fuzziness in provider capability exposition.

None of the previous work for Cloud Brokering is based on a genetic approach. However, this approach has been recently leveraged for Cloud Scheduling. In particular, the work by Pop et al. [13] focus on the scheduling of independent tasks based on the reputation of resources. Though their model is quite different from ours, their insight into genetic operators could be leveraged in our work for boosting performances in terms of evolutionary steps for population convergence. Another remarkable work has been done by Zheng et al. [16] that address the problem of resource scheduling in Infrastructure as a Service (IaaS) Cloud. They focused on parallel genetic algorithm for speeding the resource allocation process and improving the utilization of system resources. In our work, QBROKAGE is considered as a service that requires user interaction and thus optimize execution times is not the primary goal. However, we do not exclude to explore parallel genetic algorithm in the future.

This work is the outcome of many efforts. Under the supervision of the authors, a student worked on a first attempt [14] to develop a genetic meta-service for exploiting resources of different providers and allocating VMs on top of them. Based on such implementations, we provide a model and refined the genetic algorithm, including a completely redefinition the fitness function. Moreover, we integrated such implementation on SmartFed [1], a cloud simulator we developed for extending CloudSim [3] with Cloud Federation scenarios. We leveraged SmartFed for performing the experiments presented in Section VI and thus
we also extended the functionalities of SmartFed in order to run the presented scenarios.

III. MODEL AND NOTATION

In this section we model applications, resources and QoS as used throughout the paper. These models are partially derived from a previous work of the same authors [1].

A. Applications

An application \( \alpha \) is represented by an undirected graph \( \langle G_N, E_M \rangle \), where \( G \) represents the set of \( N \) vertexes and \( E \) represents the set of \( M \) edges connecting the vertexes. Each vertex \( g_i \in G \) embodies an appliance each one potentially providing different services, e.g. one appliance provides a firewall and another one provides a back-end database. Each appliance can be composed by multiple correlated VMs (\( \text{vm}_{1..K} \in g_i \)) but for the sake of the presentation we assume that each appliance is composed by only one VM, and thus the terms appliance and VM can be used interchangeably. We will release this assumption when performing experiments in Section VI.

Each edge \( e_{i,j} \in E \) represents an undirected communication path connecting vertexes \( g_i \) and \( g_j \).

B. Resources

In this work we consider resources of IaaS cloud providers. Each provider is modeled with a datacenter \( p_i \) composed by a set of hosts, that can run one or more virtual machines depending on their resources availability. The resources of each datacenter are interconnected by a network characterized by a specific set of features, such as bandwidth, latency and security capability. Depending on its performance capabilities each datacenter can run a different number of applications, i.e. set of VMs. Each datacenter is characterized by a set of resources and exposes its limits, i.e. the maximum amount of resources that can be assigned to a VM in order to be run on that provider.

To catch the heterogeneity of current pricing models adopted by cloud providers, we consider both providers adopting a per-resource cost model and providers adopting a per-VM cost model. In the per-VM case, we model cloud providers as capable of running predefined type of VM provided by customers and charging them for the whole VM used over time. In the per-resource case, we model cloud providers as capable of running each type of VM provided by customers (up to the datacenter limits) and charging them per unit of used resources over time.

C. QoS

We consider a set of QoS attributes \( Q = \{q^1, \ldots, q^S\} \) that can be classified in three categories, as defined in the work of Ye et al. [15]:

- **ascending** QoS attributes, in which higher values are better than lower ones;
- **descending** QoS attributes, in which lower values are better than higher ones;
- **equal** QoS attributes, in which only equality or inequality is meaningful.

When these attributes are considered in specifying the application, they turn into constraints values to be respected by the infrastructure. Thus, for each attributes we denote as \( \gamma_i^j \) the constraint value related to appliance \( g_i \) for the QoS attribute \( q^j \). Instead, we denote as \( \beta_i^j \) the actual value considered for the cloud provider \( p_i \) and related to the same attribute \( q^j \). For the sake of simplicity we may omit the index \( i \) when it will not be strictly necessary to the comprehension, like in the following.

For checking the adherence of constraint values coming from the application to actual values coming from the infrastructure, a set of inequality constraints \( QC = \{QC^1, \ldots, QC^S\} \) is built, whose cardinality is the same of set \( Q \). In particular, we have the following cases:

- \( QC^j = \gamma^j - \beta^j \leq 0 \) in case of ascending attributes
- \( QC^j = \beta^j - \gamma^j \leq 0 \) in case of descending attributes
- \( QC^j = |\gamma^j - \beta^j| - \epsilon \leq 0 \) in case of equal attributes

When \( QC^j \leq 0 \) the constraint is respected, otherwise it is not respected. Moreover, the following \(-1 \leq QC^j \leq 1\) holds by design, as \( \beta^j \) and \( \gamma^j \) are normalized values with respect to the maximum value for \( q^j \).

Also, a weight \( \omega_j \) is associated to each attribute \( q^j \) for modeling a search guided by user preferences and/or implementation of particular policies for some attributes. It is out of the scope of this paper to find a user-friendly way for allowing customers to express such preferences. For simplicity we assume that customers can specify an input vector \( V = (v^1, \ldots, v^S) \) of relative values and each \( \omega_j \) is calculated by equation \( \omega_j = \frac{v^j}{|V|} \) where \( |V| \) is the unitary norm of vector \( V \).

IV. QBROKAGE ARCHITECTURE

This section proposes the architecture of QBROKAGE, depicted in Figure 1, and briefly describes it. Customers sub-
mit applications to QBROKAGE by using the OVF standard format. The OVF format permits to describe applications in terms of appliances, each one composed by a set of VMs inter-connected by virtual networks. Both VMs and networks can be characterized by resources’ requirements. The OVF Parser is the component responsible to parse OVFs given in input, for constructing an application graph as defined in Section III. The application is then enqueued on the Application Queue, which buffers the current request for application mapping and manages the delivery to the Mapping component. For the sake of simplicity, this work only considers an M/M/1 application queue.

The Mapping component computes a set of mapping of the application, considering the properties of the datacenters and the estimation of the current network characteristics, as provided by the Monitoring component. Therefore, the Mapping component works with a representation of application, datacenters and QoS properties, built according to the modelization presented in Section III. This allows for flexibility and the ability to compute complex plans without dealing with burden of contacting datacenters. Once one or more mapping plans are computed, those are passed to the Allocator.

The Allocator component tries to allocate the cloudlet by considering one mapping plan at time. Note that, unlike the mapping component, the Allocator effectively contacts the cloud providers for allocating VMs. This can lead to failures or exceptions, which are properly managed by the component. Indeed, when an allocation defined by the mapping fails, the Allocator has two possibilities: (i) either rolls back and starts over with another plan, or (ii) continues with the next plan on the list, leaving the already allocated VMs in places. The first case requires longer allocation time, but it is needed when the application has functional dependencies among its component. The second case can be useful in cases where such dependency is not present, to speed up the allocation process. In either cases, when all mapping plans are considered and no one of them is resulted correct the allocator raises an error to the mapping component, which in turn contact the user.

This work focuses on the Mapping component and, in particular, the following section will describe the current algorithm leveraged in the such component for computing application mappings in term of VMs and datacenters.

V. GENETIC ALGORITHM

Genetic Algorithm (GA) is a well-known heuristic approach that permits to iteratively find near-optimal solution in large search spaces. This work leverages the GA approach because it is flexible enough to support multiple constraints and the injection of additional constraints with minimal interventions on the algorithm. Clearly, this is crucial for software reuse in the context of Cloud Computing, where QoS models are continuously enriched as providers are beginning to support QoS guarantees previously not addressed, such as soft real-time guarantees for virtualized services [5].

In the proposed algorithm, each solution (chromosome) \( c \) is a vector of length \( N \), representing the allocation mapping of each appliance, i.e. a single VM (see Section III), to a cloud provider. In other words, if \( c(i) = j \) it means that appliance \( g_i \) will be allocated on provider \( j \). For example, Figure 2 shows a chromosome representation where appliance \( g_2 \) is allocated on provider \( p_1 \) and thus \( c(2) = 1 \).

The brokering algorithm follows the canonical GA, as detailed in Listing 1. In particular, the initial population of \( S_p \) individuals is randomly generated and each individual is evaluated by considering better individuals with those having higher fitness values. The population selected for mating, is randomly chosen with a rate \( R_{cross} \) among the total and the crossover strategy is the random one-point crossover. Thus, a number of \( S_p \times R_{cross} \) crossover operations are performed for each generation and each operation produces 2 individuals. Then, mutation is applied with a probability of \( P_{mut} \) to each gene in each individuals. A discussion on the numeric values to be used for \( R_{cross} \) and \( P_{mut} \) is deferred to Section VI. After applying mutation, an elitist selector is used to select the top 90% of the population size, whilst the remaining is obtained by cloning the best selected up to reach the population size.

```java
Listing 1. Canonical GA of QBROKAGE

creating_initial_population();
while (!termination_condition){
    population_evaluation();
    selection_for_mating();
    one_point_crossover();
    random_mutation();
    elitism_selection_for_new_generation();
    termination_condition_eval();
}
```

Fitness function: When evaluating a solution, the proposed algorithm first consider each gene \( g \) in isolation by defining a column vector \( D_g = (Q_{g1}^1, \ldots, Q_g^S) \) as the distance of an allele from constraint satisfaction. According to the problem formulation explained in Section III-B, we consider values \(-1 \leq QC_g^j < 0\) as those satisfying...
the constraint $q^j$ for gene $g_j$, with better solutions in the minimizing gradient direction.

As fitness functions are usually modeled as maximization problems, we define the fitness of each gene

$$l(g) = AD_g$$

where $A$ is a square matrix

$$A = \begin{pmatrix} a_{1,1} & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{S,S} \end{pmatrix}$$

with each element in the diagonal is defined as

$$a_{ii} = \begin{cases} -\omega^i & \text{if } QC^j < 0; \\
0 & \text{if } QC^j > 0. \end{cases}$$

Finally, the fitness function of each chromosome $c$ is defined as

$$F(c) = \sum_{i=1}^{N} l(g_i) * K$$

being equal to a constant defined for awarding those chromosomes with an high number of genes corresponds to allocations respecting constraints.

VI. EXPERIMENTS

This section presents some experiments performed for evaluating our broker, both quantitatively and qualitatively. We first compare our broker against a state of the art approach. Then, we analyze how the parameters of the genetic approach affects results when dealing with an higher number of providers and VMs. After experimentally tuning our broker, we evaluate it when coping with hundreds of providers and we finally measure the number of providers used in acquiring resources, for giving an estimation of the vendor lock-in degree when using our implementation.

QBROKAGE is implemented in Java and uses the JGAP framework (see http://jgap.sourceforge.net/) for performing common genetic operations. The experiments were run in an environment that simulates Cloud Federation and/or Inter-Cloud scenarios. Such environment is provided by SmartFed, which in turn is built upon CloudSim (see Section II). All simulations have been ran on a machine equipped with Java 7, 16GB of RAM, an Intel i5-2550 quad core @3,30 Ghz.

Unless differently specified, presented results are obtained by averaging the output of 20 independent runs of simulation. Another common setup regards the QoS attributes of applications. We considered the following set $Q = \{ \text{costPerVm}, \text{ram}, \text{storage}, \text{location} \}$. In this case $\text{costPerVm}$ is a descending QoS attributes (lower values are better), $\text{ram}$ and $\text{storage}$ are ascending attributes and $\text{location}$ is an equal attributes. We are interested to minimize cost for the user and thus we leverage the weight mechanism already presented in Section III-B for going in the direction of cost minimization.

<table>
<thead>
<tr>
<th>VM Number</th>
<th>CPU</th>
<th>RAM (GB)</th>
<th>Disk (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Balancer</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Web Server</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Database</td>
<td>1</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EC2</th>
<th>Rackspace</th>
<th>Aruba.it</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallVertex</td>
<td>0.085</td>
<td>0.240</td>
</tr>
<tr>
<td>LargeVertex</td>
<td>0.680</td>
<td>0.240</td>
</tr>
</tbody>
</table>

A. QBROKAGE Comparison

QBROKAGE is compared with the state of the art by considering and extending a scenario defined by the authors of the STRATOS broker service [11]. Such scenario, briefly recalled here for the reader’s convenience, consider an application composed by a load balancer, a web server and a database – 4 VMs in total. In our model, such application can be represented by three vertexes, as described in Table I, where the number of VMs and the desired characteristics of the corresponding VM are indicated for each vertex. As the load balancer and the web server have the same requirements, their configuration will be denoted as Small, whilst the database one will be denoted as Large. In the former experiment, the authors of STRATOS consider two providers, Amazon EC2 and Rackspace (RS), that charge customers on a VM basis, at costs indicated in the first two columns of Table II. They show that up to 48% are realized when using the broker, reaching the optimum cost value of 0.495 for the whole application. We repeated such experiment with QBROKAGE and we obtained the optimum cost value at the first evolution step with a population $S_P = 20$, as described in the first row of Table III.

In addition, we extend this scenario with another provider, for leveraging the QBROKAGE capability of considering also providers applying a pricing model that charge customers on resource basis rather than on VM basis. For this experiment, we consider Aruba.it, an Italian provider that allows customers to create VMs by specifying the desired quantity for each resource and thus applies a per-resource pricing model. For obtaining prices indicated in last column of Table II, we use the following prices expressed in $ per hour (for simplicity we assume that the euro/dollar exchange rate is 1 euro equal to 1.3 dollars – see http://www.cloud.it/en/cloud-computing/pricing.aspx): 0.026 for 1 CPU, 0.005 for 1 GB of RAM, 0.0039 for 10 GB of Hard Disk. When considering the three providers in conjunction, QBROKAGE is capable of finding the optimal result for cost minimization, that is 0.493 as resumed in the second row of Table III.
Table III
QBROKAGE RESULTS FOR STRATOS SCENARIOS

<table>
<thead>
<tr>
<th></th>
<th>Optimal Cost</th>
<th>QBROKAGE Cost</th>
<th>QBROKAGE Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>FormerScenario</td>
<td>0.495</td>
<td>0.495</td>
<td>1</td>
</tr>
<tr>
<td>ExtendedScenario</td>
<td>0.493</td>
<td>0.493</td>
<td>3</td>
</tr>
</tbody>
</table>

Table IV
VERTEX REQUIREMENTS AND PROVIDERS COST AND CHARACTERISTICS

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex requirements and providers characteristics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>memory</td>
<td>512 MB</td>
<td>16 GB</td>
</tr>
<tr>
<td>bandwidth</td>
<td>10 KB</td>
<td>10 MB</td>
</tr>
<tr>
<td>disk</td>
<td>4 GB</td>
<td>10 TB</td>
</tr>
<tr>
<td>cores</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>mips per core</td>
<td>1000</td>
<td>25000</td>
</tr>
<tr>
<td>Provider cost (in currency per hour)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>memory (per GB)</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td>storage (per GB)</td>
<td>0.0002</td>
<td>0.0020</td>
</tr>
</tbody>
</table>

Figure 3. Distance to optimum with $R_{cross} = 0.35$ and different values for $P_{mut}$ by varying number of genetic steps.

B. QBROKAGE Tuning

Given that our broker is able to cope with a small number of VMs and providers, we increased the size of the problem and contemporary studied how different parameters affect the behavior of QBROKAGE, in order to properly tune the genetic allocator. In particular, the main purpose of this experiment is to experimentally find reasonable values for $R_{cross}$ and $P_{mut}$, which are the main parameters characterizing the genetic algorithm described in Section V. To this purpose, we consider a case study in which an application is composed by 3 vertex (12 VMs in total) and desired resources must be found among a set of 50 providers, assuming that each provider is capable of accepting an infinite number of VMs. The requirements for each application vertex and the provider characteristics have been generated by following a uniform distribution, in the ranges resumed in Table IV. For simplicity in generating providers, in this case we only use the per-resource cost model. Once generated datacenters and VMs, we compute the optimal solution in terms of cost, i.e. the less expensive allocation that contemporary satisfies the QoS requirements, founding a value of 58.67.

Then, starting from default configuration of JGAP – $R_{cross} = 0.35$ and $P_{mut} = 1/12$ – we first vary mutation probability and then crossover rate, studying how solutions are distant to optimum for an increasing number of evolution steps. Figure 3 shows the distance to optimum (where 1 is the maximum distance calculated in the positive space) with $R_{cross} = 0.35$ and $P_{mut} = \{1/6, 1/10, 1/12, 1/22\}$. Although all the configurations reach the optimum within 120 steps, it can be seen that the curve relative to $P_{mut} = 1/10$ is very close to the optimum starting from 90 steps, much earlier than the other configurations. Further increments of $P_{mut}$ does not yield better performances, as seen with 1/6.

Given this result, we try different crossover rates by using configuration with $P_{mut} = 1/10$. Figure 4 shows the distance to optimum with different values for $R_{cross}$. It can be seen that for $R_{cross} = 0.2$, the distance to optimum is significantly higher than the other configurations, at least up to 60 evolution steps. After that point, all the curves converge to the optimum. The bottom curve, corresponding to $R_{cross} = 0.80$, is the one which yields a measurable advantage with fewer steps. However, increasing $R_{cross}$ leads to increasing execution times, for the higher number of operations to be performed. In fact, by measuring the execution times for $R_{cross} = 0.80$ and $R_{cross} = 0.35$, we found that the latter is always completing around $400ms$ early. As an example, in the case of 100 evolution steps, we obtained, on average, an execution time of $903.05ms$ for $R_{cross} = 0.35$ and $1290.35ms$ for $R_{cross} = 0.8$, with a difference on the distance-to-optimum that is quite negligible. For this reason, we decided to operate a trade-off, using the configuration with $R_{cross} = 0.35$.

To resume, as a consequence of the result shown in Figure 3 and 4 and results gathered for execution times (not completely shown for brevity), we choose $R_{cross} = 0.35$, $P_{mut} = 1/10$ as default configuration for QBROKAGE, to be used in the following experiments.

C. Scalability

In this experiment we studied how QBROKAGE scales in terms of distance-to-optimum and computation time when increasing the number of providers up to 500. This experiment was run with $R_{cross} = 0.35$, $P_{mut} = 1/10$, $S_P = 50$ and 120 evolution steps. Figure 5 shows the results for the computation time. It can be noticed that the time grows sub-linearly with the number of providers and thus it scales well for this parameter. Considering the gathered results,
Figure 4. Distance to optimum with $P_{mut} = 1/10$ and different $R_{cross}$ by varying number of genetic steps.

Figure 5. QBrokage computation time in seconds, population size 50, up to 500 providers.

QBrokage takes from 1.05 (50 providers) to 1.25 (500 providers) seconds to compute the mapping. In our opinion, this increment is not due to the genetic algorithm itself but it is due to the some sorts performed on providers and thus requiring more time when increasing provider number. Although not extremely fast, it can considered still a tolerable delay for an interactive service. The computation time could be reduced by using less evolution steps and thus sacrificing precision (see Section VI-B) and/or by exploiting a parallel genetic algorithm.

The distance-to-optimum obtained for different number of providers has been plotted in Figure 6. When changing the number of providers, we regenerate the application and providers by using ranges described in Table IV. The plot suggests that the considered configuration of QBrokage scales up to 400 providers. Indeed, until 400 providers the distance is always below 0.78% (worst case of 350 providers). To manage a number of providers larger than 400, QBrokage could be configured differently, for instance by increasing $S_P$ and/or the number of evolutions steps. We defer to future work an experimental validation of these intuitions.

D. Vendor Lock-in

In this experiment, we compare QBrokage with a naive approach in term of resilience to the vendor lock-in. The naive approach narrows down the provider number by considering those that meet the given QoS and then tries to map the application by only considering cost minimization. We measure the dispersion of the application by simply averaging the number of used providers in 20 independent runs, for a different number of suitable providers. In this case, the providers are generated in the range described by Table IV, whilst the application is always generated by considering min values of such range, in order to have all the providers as suitable for choice.

The gathered results can be seen in Table V. It can be noticed that QBrokage trades cost-effectiveness for the ability to use multiple providers. The increment of cost is limited, as the maximum difference in cost is +2.6% when considering 200 providers. However, this yields to an increment of +75% (still with 200 providers) in the number of used providers.

<table>
<thead>
<tr>
<th>Suitable providers</th>
<th>QBrokage Cost in currency</th>
<th>QBrokage Used providers</th>
<th>Naive Cost in currency</th>
<th>Naive Used providers</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.082</td>
<td>1.50</td>
<td>0.081</td>
<td>1.0</td>
</tr>
<tr>
<td>100</td>
<td>0.078</td>
<td>2.35</td>
<td>0.077</td>
<td>1.0</td>
</tr>
<tr>
<td>150</td>
<td>0.082</td>
<td>3.20</td>
<td>0.080</td>
<td>1.0</td>
</tr>
<tr>
<td>200</td>
<td>0.075</td>
<td>3.50</td>
<td>0.073</td>
<td>1.0</td>
</tr>
</tbody>
</table>
VII. Conclusion and Future Work

As Cloud Computing becomes a predominant trend, a growing amount of Cloud providers is joining the market, increasing heterogeneity and assortment of offered resources. Therefore, customers may find it hard to select a proper set of providers for acquiring resources needed by complex applications. In this paper we described QBROKAGE, a cloud broker that explores a large number of candidate solutions and choose those that meet the QoS requirements of the application. We proposed a genetic approach to the problem, because in our opinion it provides the needed flexibility for supporting multiple and heterogeneous QoS constraints. Our proposal can deal with hundreds of providers, by preserving at the same the interactivity of the service and it is capable of mitigating vendor lock-in by design, finding sub-optimal solutions.

Cloud applications might require the ability to adjust the amount of computational resources during their life time and thus acquiring more resources at run time. Even if QBROKAGE is able to deal with custom-size applications, its current version does not support elasticity for an already deployed application. Such support is deferred to future work. In the future we also plan to explore the possibility of complementing our work with ontology matching, for supporting non-functional requirements that are usually expressed in natural language.

Acknowledgment

The authors acknowledge the support of the Project FP7-257438 Contrail (Open Computing Infrastructures for Elastic Services).

References


