Agent Based Load-balancer for Multi-Cloud Environments

Luca Tasquier1*

Received 14 July 2015; Published online 8 December 2015

© The author(s) 2015. Published with open access at www.uscip.us

Abstract

Load-balancing is one of the central issues in Cloud Computing because it helps in optimal utilization of resources, thus in enhancing the performance of the system. The goal of load-balancing is improving the performance by balancing the load among various resources to achieve optimal resource utilization, maximum throughput, maximum response time and avoiding overload. Unfortunately, generally traditional algorithms are not fully adapted to the Cloud Computing system which requires development of new algorithms or adaptation of those already existing for distributed systems. Moreover, they are not feasible to exploit the extreme elasticity that Cloud brings with it, because they are focused on stable environments where resources are allocated and fixed and do not rapidly increase and decrease. In this work is presented an application aware multi-Cloud load-balancer based on mobile agent paradigm: the architecture enables the load-balancing of the user's application by using agents’ capabilities to monitor the state of the Cloud infrastructure and to detect overload and/or under-utilization conditions. The multi-agent framework will provide provisioning facilities to automatically scale the application to unloaded resources and/or to new resources coming from different Cloud providers; in the same way, agents will be able to deallocate resources if unused, thus leading to a cost saving for the user by adapting the Cloud infrastructure to the runtime application's environment.

Keywords: Load-balancing; Cloud Computing; Mobile multi-agent system

1. Introduction

Cloud Computing emerged as a new deployment model of resources as a service accessible via public or private networks. Load-balancing is one of the central issues in Cloud [1]. It is a mechanism that distributes the dynamic local workload evenly across all the nodes in the whole cloud to avoid a situation where some nodes are heavily loaded while others are idle or doing little work. It helps in optimal utilization of resources, thus in enhancing the performance of the system. The goal of load-balancing is improving the performance by balancing the load among various resources to achieve optimal resource utilization, maximum throughput, maximum response time and avoiding overload. To distribute load on different systems are used generally traditional algorithms like whose used in web servers, but these algorithms do not always give the expected performance with large scale and distinct structure of service-oriented data centers: they are not fully adapted to the Cloud Computing system which requires development of new algorithms or adaptation of those already existing for distributed
systems (Cluster, Grid Computing). To overcome the shortcomings of these algorithms, load-balancing has been widely studied by researchers and implemented by computer vendors in distributed systems [2].

In this work is presented an application aware multi-Cloud load-balancer based on mobile agent paradigm: the architecture enables the load-balancing of the user application by using agents’ capabilities to monitor the state of the Cloud infrastructure and to detect overload and/or under-utilization conditions. The multi-agent framework will provide provisioning facilities to automatically scale the application to unloaded resources and/or to new resources coming from different Cloud providers; in the same way, agents will be able to deallocate resources if unused, thus leading to a cost saving for the user by adapting the Cloud infrastructure to the runtime application's environment. The aim of the presented research is to introduce a novel methodology and to design a framework to address load-balancing in Cloud infrastructures, specifically designed to address this environment, improving the existing techniques by adding dynamic provisioning capabilities to the balancer; moreover, the detection algorithm will take into account the possibility to increase and/or reduce the infrastructure on the basis of the application runtime requirements, thus self-adapting the environment to the application’s needs and exploiting the elasticity and the pay-per-use business model of the Cloud Computing paradigm. Additionally, it will be given to the developer the possibility to design the load-balancing specifically for his/her application, taking into account its peculiarities and characteristics; furthermore, the application will be easily integrated into the load-balancer without having the Cloud infrastructure ready when the application is about to be deployed: the user will have the possibility to create a resource template at deployment time while the infrastructure will be dynamically created by the proposed framework, by composing resources provisioned from different Cloud providers, thus overcoming the vendor lock-in problem. The presented work will leverage on the mobile agent paradigm, of which it will exploit the key characteristics: the framework will adapt itself autonomously, following the runtime requirements of the application and adapting the infrastructure adding and removing resources dynamically thanks to the Cloud elasticity. Moreover, agents mobility will ensure a straightforward adaptation of the application and of the load-balancer to the runtime infrastructure by carrying the necessary functionality with the agent, thus also allowing an in-place monitoring of the resources and minimizing the communication overhead.

Section 2.1 presents the analysis of previous works in the field of the load-balancing on distributed systems, with a focus on Cloud Computing and agent-based models and technologies; moreover, in Section 2.1 are highlighted the weaknesses of the presented approaches and the improvements that the presented work bring with it. In Section 3 the multi-agent load-balancing architecture is described: for each agent composing the framework are detailed the role within the whole infrastructure, the used interfaces, which functionality of the system is addressed and how. The load-balancer functioning is reported in Section 4: the complete events’ flow is detailed, starting from the specification of the client API and of a Graphical User Interface that allow the submission of the application to the load-balancer and describing the algorithms used to detect overload and under-utilization conditions and to dynamic provision the resources. Finally, conclusion and future researches are due in Section 5.

2. Related work

Load-balancing in Cloud is a mechanism that distributes the excess dynamic local workload evenly across all the nodes [3]. It is used to achieve a high user satisfaction and resource utilization ratio [4], making sure that no single node is overwhelmed, hence improving the overall performance of the system. In [5] authors propose a content aware load-balancing policy named as workload and client aware policy (WCAP). It uses a unique and special property (USP) to specify the unique and special property of the requests as well as computing nodes. USP helps the scheduler to decide the best suitable node for the processing the requests. This strategy is implemented in a decentralized manner with low overhead. In [6] is presented a scheduling strategy on load balancing of VM resources that uses historical data and current state of the system. This strategy achieves the best load bal-
ancing and reduced dynamic migration by using a genetic algorithm. In contrast with this, the work presented in [7] proposes a central load-balancing policy for VMs that balances the load evenly in a distributed virtual machine/cloud computing environment. In [8] the authors propose a load-balancing virtual storage strategy that provides a large scale net data storage model and Storage as a Service model based on Cloud Storage. Storage virtualization is achieved using an architecture that is three-layered and load-balancing is achieved using two load balancing modules. A lock-free multiprocessing load-balancing solution that avoids the use of shared memory is proposed in [9], in contrast to other multiprocessing load-balancing solutions which use shared memory and lock to maintain a user session. It is achieved by modifying Linux kernel. Another load-balancing algorithm within Cloud environments is proposed in [10], where a load-balancing algorithm for dynamically scalable web services is presented; this algorithm provides large scale load-balancing with distributed dispatchers by, first load-balancing idle processors across dispatchers for the availability of idle processors at each dispatcher and then, assigning jobs to processors to reduce average queue length at each processor. In [11] is discussed a two-level task scheduling mechanism based on load-balancing to meet dynamic requirements of users and obtain a high resource utilization. It achieves load-balancing by first mapping tasks to virtual machines and then virtual machines to host resources.

The Cloud elasticity allows the possibility to change dynamically over time the system configuration, and so the distributed monitoring must adapt itself quickly to the new requirements. That is why we propose an agent based architecture to address the load-balancing. In [12] authors claim that an approach based on software agents is a natural way to tackle the monitoring tasks in the aforementioned distributed environments. The proposal considers a divide-and-conquer cooperation structure with agents that take into account the delays they experience and the environmental delays (i.e., processing and network delays). The usage of mobile agents leads to a number of advantages: mobile agents are used to perform the monitoring tasks efficiently and effectively, thus reducing the network traffic and carrying the tasks to wherever the required data can be obtained with a good performance; moreover, agents adapt to the current situation by taking into account the delays experienced by their cooperative agents and the environmental delays (i.e., correlation and network delays) for deciding when they should perform their tasks. Furthermore, the loose coupling among the agents increases the fault tolerance of the system, as it leads to a "graceful degrading" of performance when some of the agents fail to perform their tasks in time.

The efficiency of the agent paradigm, and in particular of mobile agents, within a distributed environment is also highlighted in [13]. In this work, the authors studies an optimal control of mobile monitoring agents in artificial-immune-system-based (AIS-based) monitoring networks. Both stationary and mobile agents are exploited within the proposed framework: stationary agents are those staying in the sensor nodes where they are created, such as knowledge-base agent and the clonal selection agent; mobile agents are those created during the system operation for monitoring purpose. The authors claim that in a mobile agent-based monitoring network, a remote user can dispatch mobile monitoring agents to sensor nodes in the network. These monitoring agents equip with data analysis and damage diagnosis algorithms and can roam over the network and perform damage diagnosis at sensor nodes where they visit.

2.1 Improvements with respect to the state-of-the-art

In the state-of-the-art presented in Section are described load-balancing techniques applied to Cloud Computing environments: the main common problem of these methodologies is that they are not explicitly designed for Cloud infrastructures, but they use resources like a cluster of nodes, without being aware about the Cloud environment: they are general purpose load-balancing techniques applied to several resources, without exploiting the advantages of Cloud Computing paradigm. As stated in [2], one of the main challenges of load-balancing in Cloud is automated service provisioning: in fact, a key feature of Cloud Computing is elasticity, by which resources can be allocated or released automatically according the runtime application requirements and thus exploiting the pay-per-use business model. The proposed methodology is designed to take advantage of this characteristic by adapting the Cloud infrastructure to the application's requirements, adding and releasing resources dynam-
ically. Moreover, the presented approach allows the dynamic provisioning of resources coming from different Cloud providers, thus enabling a multi-Cloud load-balancing and overcoming the vendor lock-in issue.

Another improvement that the proposed solution brings with it comes from the usage of the mobile agents paradigm: agents technology provides asynchronous mechanisms that could represent the best choice for effective programming of Cloud, due to the unpredictable behaviour of the network; furthermore, as authors demonstrate in [14], the advantage of mobile agents as compared to static intelligent agents is that the former allows for an easy programmability of remote nodes by migrating and transferring functionality; mobile agents are particularly useful for tasks involving distributed information retrieval and processing. Agent autonomy has been typically exploited in order to provide adaptable, fine-tuned and fault tolerant network management solutions that minimize the need for user interaction [15]. Better performance are identified in terms of reduced network management traffic, decentralization of processing and better response times unaffected by network latencies.

3. Architecture

The proposed architecture is mainly composed by three agents:

1. **Executor**: it represents the application running on the multi-Cloud environment;
2. **Provisioner**: this agent is in charge of managing the Cloud infrastructure by adding and removing resources;
3. **Monitor**: it is responsible of monitoring overload and/or under-utilization conditions of a certain resource.

An additional agent called **Controller** is in charge of giving to the user an overview about the current state of the Cloud environment. Moreover, each agent has mobility capabilities in order to migrate themselves autonomously on the multi-Cloud infrastructure.

The proposed architecture differs with a classical object-oriented approach due to the fact that each part of the load-balancer has its own behaviours and has not structural dependencies with other agents. The collaboration among them is based on events sent in an asynchronous way, exchanging data without blocking calls and reacting autonomously to them and to the runtime environment in order to change their state and their behaviours: moreover, each part of the architecture is able to migrate or to clone its functionalities thanks to the agents' mobility capabilities, without explicitly install the required environment on the selected resource.

The description of each agent will be detailed in the following sections. One of the main strength of the proposed solution is the distributed way in which the agents are deployed: there is not a single centralized control center to implement the multi-Cloud load-balancing but each group of agents is responsible of the belonging resource, using their own intelligence to collaborate with the other agents in order to implement the necessary actions to optimize the Cloud infrastructure according to the application's load. A simple scenario is depicted in Figure 1 while all the load-balancing mechanism will be described in Section 4.

As shown by the figure, to each agents' set is related a database that is used to stores local information about the specific resource, such as monitoring information, endpoint of the linked resources and so on.

3.1 **Executor**

The **Executor** is in charge of embedding the user's application: it manages its workflow by acting on a common abstract interface. An example about the connection between the agent and the application is depicted in Figure 2 (a).
The developer that wants to use the proposed framework has only to properly implement the provided interface and to compile the implemented one with his/her own application in order to generate a shared library; this library is uploaded by the Executor in its execution environment and it is loaded at runtime after its boot or at the end of migration’s operations. After that the Executor is able to run the particular application by simply using the common interface without being aware about its implementation: the correct override and usage of the interface is in charge of the application’s developer that is the only one which knows about its own executable. This flexible approach leads to a complete decoupling between the Executor and the application that can be bound in a straightforward way. The Executor interface allows the user to control and manage his/her application through the agents by using a plugin approach and thus allowing the control of the application’s workflow via agent’s technology. The framework provides a set of native interfaces in order to embed the application in the Cloud load-balancer supporting different programming languages: the developer has only to implement the interface related to the application’s programming language and merge overridden interface and native application in a shared library that can be loaded at runtime. The Executor will control the application by invoking the operations overridden by the user without being aware about the specific application: in this way it is possible to use the agent based architecture to manage the native application in a transparent way with respect to the upper layer. The interface is composed by actions and notifications: the actions are the functionality that are application-aware and have to be implemented by the developer in order to allow the management through the Executor; the notifications are events provided by the interface to the user thanks to which the application can notify some events to the related Executor. The interface is described in Table 1 while the class diagram representing the dependencies among the Executor agent, the interface and the application is depicted in Figure 3.
3.2 Provisioner

All the management of the multi-Cloud infrastructure is delegated to the Provisioner (Figure 2 (b)): it is responsible of finding new resources in case of necessity by brokering the best available resource to host the application. It is also in charge of contacting the available Cloud providers by using a number of preconfigured modules that are invoked via a common provisioner interface. Each module implements the actions to address the information retrieval from the specific Cloud provider, as well as the buying and management of the selected resource. The Provisioner, by using these modules, is also able to install the required software on the resource in order to enable the running of the agents technology on the new computational unit. Each module is able to authenticate with the provider, to create resources of specific types and to perform operations on them. This module also generates proposals for based on the specifics of the provider and on the requested resource. The module extends the functionality of an agent interface which is intended to implement the common aspects on each provider. Such aspects include persistence of proposals and resource information, credentials management or the deployment descriptions. The interface allows the agent to manage the providers and the resources in an uniform and agnostic way: it is organized in requests and callbacks in order to allow the an asynchronous management of the resources. More details are provided in [16].

When there is the necessity of adding a new resource, the Provisioner starts a new provisioning transaction by using the ideal resource description provided by the user at boot time; each provider module takes this description and translates it into a provider-aware request. After the reception of the response, the provider module translates the offer in a uniform way in order to be processed by the Provisioner, which collects all the answers and brokers the best according to the application requirements. Afterward the selected resource is autonomously acquired by using the user’s provided credentials, it is started and the necessary software is loaded in order to enable the agent based load-balancer. A notification about the new resource is sent to the user interface in order to inform the developer about the Cloud infrastructure’s reconfiguration.

The feasibility of this agent-based approach to provision Cloud resources has been discussed in [17] [18]. It is presented a collection of agent-based services for the provisioning of Cloud resources at infrastructure level, its service interface provides methods for orchestrated executions of agents that implement a scalable solution for provisioning and management of Cloud resources. Figure 4 shows the performance of the provisioning opera-
Table 1: Executor interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>action</td>
<td>this operation allows the Executor to start the application from scratch</td>
</tr>
<tr>
<td>started</td>
<td>notification</td>
<td>the developer can use this notification within its application to notify its successful start</td>
</tr>
<tr>
<td>stop</td>
<td>action</td>
<td>it is possible to define the operations that have to be performed in order to stop the whole application by calling this action</td>
</tr>
<tr>
<td>stopped</td>
<td>notification</td>
<td>the application can notify when the application successfully stops by using this notification</td>
</tr>
<tr>
<td>suspend</td>
<td>action</td>
<td>the user has to implement this action to allow the suspension of the application, also saving its status through one or more files</td>
</tr>
<tr>
<td>suspended</td>
<td>notification</td>
<td>when the suspension of the application occurs, the developer can notify the event by using this notification; moreover, he/she can pass to the upper layer the status files by using this operation</td>
</tr>
<tr>
<td>resume</td>
<td>action</td>
<td>it is possible to define the operations have to be performed in order to resume the execution of the application by overriding this action: status files can be retrieve within the resume() parameters</td>
</tr>
<tr>
<td>resumed</td>
<td>notification</td>
<td>once the application is resumed from a suspension, the resumed() notification can be rised</td>
</tr>
<tr>
<td>split</td>
<td>action</td>
<td>this is an important operation that has to be implemented: by using this action the developer can define how the load of his/her application must be splitted in case of a load-balancing operation</td>
</tr>
<tr>
<td>splittered</td>
<td>notification</td>
<td>it notifies the end of the splitting operations; eventually status files can be provided to the upper layer by using this operation</td>
</tr>
<tr>
<td>unify</td>
<td>action</td>
<td>this action can be overridden in order to implement to reduce and unify the sub-results of two parallel runs of the application; status files can be used within this operation</td>
</tr>
<tr>
<td>unified</td>
<td>notification</td>
<td>the end of the unification's operations can be notified to the Executor by using this feature</td>
</tr>
</tbody>
</table>

The experiments have been conducted varying the number of vendors and of the concurrent provisioning requests and demonstrate the feasibility of the agent-based provisioning methodology. It is possible to evince that the total time to execute the provisioning operations does not depend on the number of vendors and it increases linearly with the concurrent requests, which demonstrates a good scalability of the solution. Moreover, the performance could be affected by the efficiency of the algorithm used to choose the best resources, which is not related to the agent architecture and collaboration protocols and that can be substituted and embedded in the Provisioner like a plugin.

3.3 Monitor

The Monitor is the agent that embeds the overload and/or under-utilization policies. It runs specific algorithms to detect critical conditions and, on the basis of its computation, decides if a reconfiguration of the multi-Cloud infrastructure is needed in order to best perform the user’s application. Preliminary results about agent-based monitoring of heterogeneous Clouds are drawn in [19] [20]. The result of these works is a framework to address
the monitoring of Cloud infrastructure, by giving to the user the possibility to check the state of the resources, even if acquired from different vendors. The proposed environment offers high elasticity and extensibility by giving a high level of customization of the performance indexes and metrics; the framework uses agents mobility to perform the measurements on the selected resources, while a centralized collector computes performance indexes and QoS parameters. A prototype implementation of this framework has been presented in order to prove the feasibility of the approach.

The expertise and partial results of these works are used in the proposed load-balancer in order to efficiently monitor the Cloud infrastructure: in fact, inspired by the aforementioned architecture, it has been designed the agent that is in charge of implementing the measurements on a resource. In fact, the Monitor checks resource
parameters and decides to run overload or under-utilization detection algorithms; it also computes metrics about the current load of the computational unit in order to answer to requests coming from other agents about eventually split the application’s load. A detailed description of the whole monitoring algorithm is described in Section 4.1.

4. Load-Balancer

The proposed multi-Cloud load-balance relies on the agent based architecture described in Section 3. On the developer’s side, the activities that he/she has to do in order to enable the load-balance to manage its application are the following: first of all, the developer has to choose the Executor interface that best matches with its needings in terms of programming language; after that, he/she must implement the interface in order to allow the application management by the Executor and must merge the extended interface with the application, providing a shared library which can be loaded at runtime. At this time, the application is ready to be loaded into the load-balance and the user can specify the requirements of the computational unit that best fit with the application’s needings: these parameter could be the Operating System, the number of CPUs, the CPU speed, the memory amount and so on. After this specification, the developer can insert the parameters (such as endpoint, credentials, etc.) of each Cloud provider he/she would use to deploy his/her application: he/she can choose from a number of Cloud providers that represents the available provider modules manageable by the Provisioner; this ends the preliminary operations and the user can deploy his/her application. All these operations are supported by a client API, of which the core classes are depicted in Figure 5. This API has been designed in order to allow the development of client applications: it exposes a set of operations that an application can use to interoperate with the load-balancing framework. The user can create an Application object that contains all the information useful for the load-balancer and that can be passed to the agents at deployment time using the ApplicationDeployer entity. The creation of the Application object and the deployment via the ApplicationDeployer class can be done by using user clients: in Figure 6 is proposed a graphical implementation of a user’s client application.

![Figure 5: Client core API](image)
In the "Application chooser" section, the user can load the application shared library, while using the "Resource definition" section it is possible to specify the constraints that the Provisioner must follow to choose the computational units. The "Available Clouds" section allows the developer to configure the Cloud providers that will eventually host his/her application: the inserted parameters will be stored in the Provisioners at the application’s deployment.

When the application deployment starts, a Controller and a Provisioner are created: the Provisioner takes the Resource definition from the Application deployer and contacts the configured providers to retrieve offers; the brokering module of the agent takes into account the user’s preferences to choose the best resource together with the "minimum price" policy: the Provisioner always tries to choose the offer that matches the application's requirements minimizing the cost. After the brokering phase ended, it acquires the selected resource and notifies to the Controller the information about the new computational unit (endpoint, provider, cost, service level agreement, etc.). The Provisioner starts the resource and, after its boot, loads the agent framework and the local database on it: at this point, an Executor and a Monitor are created on the user's host: the Executor tries to embed the shared library in itself in order to automatically transport the whole application on its migration to the selected resource. If the library size exceeds the maximum amount of data that the agent can bring with it, the Executor sends the application to the resource by using alternative technologies, such as Secure Copy (SCP) and so on. After this initial stage, the Provisioner, the Executor and the Monitor migrate on the resource. At the end of the migration, the Executor writes the shared library if it brought the application within its environment and loads it, then it calls the native start() action to boot the application and forward the started notification to the Controller which stores this information in case the user asks for the application's state. All the deployment actions are depicted in Figure 7.

During the execution of the application, the Monitor could detect a resource overload: in this case, the Provisioner starts a new provisioning transaction. The first thing the Provisioner does is searching for nodes already running that are available to accept part of the computation workload: each agent's group has not the complete knowledge about the whole Cloud infrastructure, but it only knows which the neighbouring nodes are. The nodes retrieval works as follows:
1. the applicant Provisioner (aka root applicant) asks for computational power only to its neighbours, if any;

2. each neighbour Provisioner evaluates the request and asks the Monitor to computes the resource state (AVAILABLE or BUSY);

3. if available, the neighbour answers to the applicant with AVAILABLE, providing also the Provisioner i-

dentifier within the return message; if it is busy, forwards the request to its neighbours, if any, thus

becoming the applicant for the neighbour;

4. if a node has not neighbours and it is busy, it answers to the applicant with BUSY;

5. if an applicant receives an AVAILABLE, it stops the search and forwards it to its applicant; if it receives a

BUSY from all the neighbours, it answers BUSY to its applicant;

6. if the root applicant receives an AVAILABLE, it stops the search, stores the Provisioner identifier as new

neighbour and contacts it for splitting the workload; if it receives BUSY from all the neighbours, then it

starts an external provisioning transaction.

If the Provisioner received an AVAILABLE, the Executor invokes the suspend() native action and forwards the

suspension request to the neighbour Executor; this one, when receives the suspended notification, forwards it

to the applicant Executor that, after received the two notifications (its and of its neighbour), invokes the split() action and sends the state files produced by the splitted notification to the neighbour. This one receives the state files and uses the unify() to reduce the results; on the unified reception, the neighbour calls the resume() and asks to the applicant to resume its execution. If the Provisioner received BUSY from all the neighbours, it starts a provisioning transaction as described for the deployment. After acquired the resource and loaded both the agent framework and the database, it stores it as a new neighbour and asks to the Executor to suspend the

application. After that, the Executor calls the split() and clones itself, the Provisioner and the Monitor. The three

Figure 7: Application deployment
clones migrates on the new resource and the both the applicant and the new Executor calls the \textit{resume()} action to resume the application execution.

As Cloud Computing business model is based on the "pay-per-use" paradigm, under-utilization detection has the same importance of the overload condition checking in order to exploit the elasticity brought by the Cloud environment and to not pay for idle resources if the application’s requirements change at runtime or the workload has been overestimated at design time. For this reason, when an under-utilization of a resource is discovered by the related Monitor, the Provisioner asks for nodes that can accept its load without leading themselves to an overload condition: the nodes discovery follows the same flooding algorithm described above. If no resources are discovered, the applicant resource continues its execution and restarts the idle detection algorithm described in Section \ref{sec:monitoring_algorithm}. If a node is found, it asks for a suspension of the application execution to the target Executor and sends it the state files representing the elaboration results of the applicant Executor. The target Executor uses the \textit{unify()} native action to unify the results, receives the \textit{unified} notification and asks the Provisioner to shutdown and deallocate the applicant node. After this operation the Executor runs the \textit{resume()} action to restart the application. The deletion of the node has not to be notified to the other nodes: they will become aware about the node deletion only when an old neighbour tries to contact it and receives a connection rejection; at this point it will delete the node from the list of the neighbours.

As it is possible to understand, the usage of the agent based paradigm in the multi-Cloud environment leads to a complete scalable approach for balancing the applications in the Cloud, taking into account the high elasticity of this computing paradigm. Moreover, the overdescribed event-driven algorithm reduces the communication through the network because messages are exchanged only when it is strictly necessary. The Controller can ask for the state of each resource by using the same messages’ flow described for the provisioning phase. Each node answers to the Controller with its application’s execution state and other information (e.g. resource endpoint, provider, etc.), and forwards the request to its neighbours. In this way a user can take under control the whole infrastructure when needed.

4.1 \textit{Monitoring algorithm}

After the application deployment, the Monitor starts checking the resource behaviour in terms of CPU usage and determines whether the overload or the under-utilization detection should start.

For what concerning the \textbf{overload detection}, the Monitor sets a threshold about the percentage of CPU usage that brings to the starting of the overload detection. Denoted by $OL_{\text{threshold}}$ this threshold, the overload detection starts if $CPU_{\text{usage}} > OL_{\text{threshold}}$. The overload detection starts setting a temporal window $T_{od}$ in which the values of $CPU_{\text{usage}}$ are read with a sampling period $T_s$. At the end of $T_{od}$ the Monitor evaluates the number of samples $n_{od}$ in which $CPU_{\text{usage}}$ overcomes $OL_{\text{threshold}}$: if the number of samples overcomes the 50% of the total samples evaluated in $T_{od}$, the Monitor deduces that the resource is overloaded. The overload condition is expressed by Equation \ref{eq:overload}.

\begin{equation}
OVERLOAD = \frac{n_{od}}{NO_{tot}} > 0.5, \text{ with } NO_{tot} = \left\lfloor \frac{T_{od}}{T_s} \right\rfloor
\end{equation}

If the overload detection is negative, $OL_{\text{threshold}}$ is proportionally increased by the number of times in which the overload detection is started (denoted as $M_{ol}$), thus better tuning the threshold and to not overburden the Monitor with continuous overload detection (Equation \ref{eq:ol_threshold}).

\begin{equation}
OL_{\text{threshold}} = OL_{\text{threshold}} + M_{ol}
\end{equation}

If the overload detection succeeds, the Monitor asks to the Provisioner to start a new provisioning transaction, as described in Section \ref{sec:provisioning}. 


Regarding the idle detection, the Monitor sets CPU\_usage threshold value ID_{threshold} and a temporal window T_{id} in which it observes the CPU\_usage behaviour: if the CPU\_usage overcomes the overload threshold OL_{threshold} and the Monitor starts the overload detection algorithm, T_{id} is reset and ignored. If the Monitor is not in overload detection, it samples the values of CPU\_usage with a sample time T_s: at the end of T_{id} the Monitor evaluates the number of samples n_{id} in which CPU\_usage is under ID_{threshold}: if the number of samples overcomes the 50\% of the total samples evaluated in T_{id}, the Monitor states that the resource is under-utilized. The idle condition is expressed by Equation 3.

\[
IDLE \Rightarrow \frac{n_{id}}{N_{tot}} > 0.5, \text{ with } N_{tot} = \left\lfloor \frac{T_{id}}{T_s} \right\rfloor
\]  

(3)

If the idle detection succeeds, operations described in Section 4 are implemented, otherwise the Monitor resets T_{id} and runs the idle detection algorithm again.

Based on the current behaviour of the resource in terms of load, the Monitor is also in charge of evaluating the current state of the computation unit that the Provisioner has to forward in case of requests coming from neighbours. If the resource is in overload detection, it answers BUSY to all the requests; if the agent’s group has started the resource deletion operations, the Provisioner still answers BUSY to the requests; if the computational unit is in the normal execution state and the Provisioner receives a split request, the Monitor stops the temporal window T_{id} and saves its temporary value T_{id\_temp}. After that, it evaluates IDLE_{temp} as described in the Equation 4:

\[
IDLE_{temp} = \frac{n_{id\_temp}}{N_{temp}}, \text{ with } N_{temp} = \left\lfloor \frac{T_{id\_temp}}{T_s} \right\rfloor
\]  

(4)

where n_{id\_temp} is the number of samples in which CPU\_usage is under ID_{threshold} evaluated within T_{id\_temp}. On the basis of the IDLE_{temp} evaluation, the Monitor computes the state as in the Equation 5:

\[
\begin{align*}
\text{state} &= \text{AVAILABLE, if IDLE_{temp} > 0.5} \\
\text{state} &= \text{BUSY, if IDLE_{temp} \leq 0.5}
\end{align*}
\]  

(5)

After this evaluation the T_{id} counter is reset and, if the state evaluation has been AVAILABLE, the Monitor sets a flag which means that the resource can receive some load from a specific resource. If the flag is set and another request arrives, the Provisioner immediately answers with BUSY because it already "promised" its availability to another resource. If during T_{id} does not arrive a split request from the applicant resource, the flag is reset and the node becomes available to other resources asking for computational power.

5. Conclusion

Load-balancing is a challenging field within the Cloud Computing paradigm. Generally traditional algorithms do not always give the expected performance with large scale structures. Moreover, they are not feasible to exploit the extreme elasticity that Cloud brings with it, because they are focused on stable environments where resources are allocated and fixed and do not rapidly increase and decrease.

In this work is presented a methodology to implement an application aware multi-Cloud load-balancer and the design of the load-balancing framework based on mobile agent paradigm: the architecture enables the load-balancing of the user’s application by using agents’ capabilities to monitor the state of the Cloud infrastructure and to detect overload and/or under-utilization conditions. The usage of a native interface allows the agents to manage the user application in a transparent way with respect to the agent technology. Agents are able to communicate with each other the resources’ state and thus implementing provisioning and load-balancing capabilities by using an highly scalable allocation and deallocation algorithm for the computational units.
The presented approach exploits the Cloud elasticity addressing the automatic service provisioning, that is one of the main issues in load-balancing within Cloud environments and allows to overcome the vendor lock-in problem by managing resources coming from different providers without the user's intervention. The mobile agent framework self-adapts to the runtime states of the application and of the execution environment: each agent monitors each resource and contacts other agents only if needed; the messages’ flow is designed to minimize the communication overhead and to implement a scalable and fault-tolerant solution, without having a centralized load-balancer. Furthermore, the agents’ mobility allows the migration of the monitoring environment and of the application state within the agent, thus exploiting the agent's reactive and proactive capabilities to strengthen new resources with the needed functionalities or to release them without losing the application execution results. The feasibility of the proposed architecture has been demonstrated by detailing results coming from previous works regarding agent-based provisioning, management and monitoring of Cloud infrastructures.

Since the presented work focuses on the methodology and design of the novel load-balancer for multi-Cloud environments, future works will mainly deal with the implementation of the proposed architecture and algorithms, together with the data evaluation of the presented approach with respect to well-known load-balancing algorithms. After that, the next step will be the design and implementation of new algorithms for load-balancing to embed in the monitoring actions in order to relate more rules and check also higher-layer parameters, such as service levels and Quality of Service parameters.

References


