CLEVER: A CLoud-Enabled Virtual EnviRonment

F. Tusa, M. Paone, M. Villari and A. Puliafito
Università degli Studi di Messina, Facoltà di Ingegneria
Contrada di Dio, S. Agata, 98166 Messina, Italy.
e-mail: {ftusa,mpaone,mvillari,apuliafito}@unime.it

Abstract—A cloud-enabled virtual environment called CLEVER is proposed in this paper. CLEVER aims to design a Virtual Infrastructure management layer to simplify the access and administration of private/hybrid clouds. It also provides simple and easily accessible interfaces to interact with different “interconnected” cloud computing infrastructures and successfully deploy Virtual Machines, that can eventually migrate. The concept of interface is exploited for integrating security, contextualization, VM disk image management and federation functionalities made available from higher level software components. Due to its pluggable design, CLEVER is able to grant high scalability, modularity and flexibility in software components. Due to its pluggable design, CLEVER is able to grant high scalability, modularity and flexibility in the middleware architecture, while fault tolerance requirements are also satisfied. A prototype version of CLEVER has been developed to implement and test its main features, as discussed in the final part the paper.

Keywords: cloud computing, XMPP, fault tolerance, Virtual Infrastructure Management, clusters.

I. INTRODUCTION

Cloud computing is generally considered as one of the more challenging topic in the IT world, although it is not always fully clear what its potentialities are and which are all the involved implications. Many definitions of cloud computing are presented and many scenarios exist in literature. In [1] Ian Foster describes Cloud Computing as a large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet.

In order to provide a flexible use of resources, cloud computing delegates its functionalities in a virtual context, allowing to treat traditional hardware resources like a pool of virtual ones. In addition virtualization enables the ability of migrating resources, regardless of the underlying real physical infrastructure. Therefore, using virtualization and aggregating virtual resources, also mapped in different physical hardware, clouds provide services at three different levels: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) [2].

Another type of classification can be performed assuming Public Clouds, Private Clouds and Hybrid Clouds [2]. Public Clouds such as Amazon EC2 [3] offer a computing infrastructure, publicly accessible from a remote interface, for creating and managing Virtual Machine (VM) instances within a proprietary infrastructure, where many different customers can run and control their own applications.

Recently, interest is growing in open source tools that let organizations build their own IaaS clouds using their internal computing resources [4]: private clouds, in fact, are provided by an organization and offer a dedicated operating environment with high trust level. The computing infrastructure is thus owned by a single customer that controls the applications being executed. The main aim of such clouds is not to provide and sell computing capacity over the Internet through publicly accessible interfaces, but to give local users a flexible and agile private infrastructure to run service workloads within their administrative domains. Private clouds can also support a hybrid cloud model, by adding to the local infrastructure more computing capacity coming from an external public cloud. A private/hybrid cloud can allow remote access to its resources over the Internet using remote interfaces, such as the Web services interfaces that Amazon EC2 uses.

The logical organization of private/hybrid cloud architectures can be analyzed keeping in mind the stack of Fig. 1 where two main layers (the lowest one named Virtual Infrastructure cloud Management and the highest one named High-level cloud Management), representing two different logical levels of cloud management, are depicted: a fundamental component of private/hybrid clouds is represented by the Virtual Infrastructure (VI) management (the lowest layer in the picture) which acts as a dynamic orchestrator of Virtual Environments (VEs). In project as OpenQRM [5] and OpenNebula [6], the VI manager in fact deploys and manages VEs, either individually or in groups that need parallel scheduling on local resources or external public clouds. It automates VE setup (preparing disk images, setting up networking, and so on) regardless of the underlying virtualization layer (Xen, KVM, or VMware). Although the creation of private clouds was already possible with the tools these project provide, other relevant features lack for building hybrid IaaS clouds, such as public cloud-like interfaces, mechanisms for adding such interfaces easily, and the ability to deploy VMs on external clouds.

On the other hand, in the same scenario of private/hybrid clouds, projects such as Globus Nimbus [7] and Eucalyptus [8], which can be considered as cloud toolkits (they mainly deal with the highest layer of the stack of Fig. 1), are instead able to transform existing infrastructure into an IaaS cloud with cloud-like interfaces. Eucalyptus is compatible with the Amazon EC2 interface and is designed to support additional client-side interfaces. Globus Nimbus exposes EC2 and Web Services Resource Framework (WSRF) [9] interfaces and offers self-configuring virtual cluster support. However, although these tools are fully functional with respect to providing cloud-like interfaces and higher-level functionality for security,
contextualization, and VM disk image management, their VI management capabilities are limited and lack the features of solutions specialized in VI management. Such toolkits attempt to span both cloud management and VI management (i.e. they try to implement functionalities belonging on both the stack layers of Fig. 1) but, by focusing on the former, are not able to provide the same functionalities of software written specifically for VI management. Although integrating high-level cloud management middlewares with existing VI managers would seem like the obvious solution, this is complicated by the lack of open and standard interfaces between the two layers, and the lack of certain key features in existing VI managers.

Our work, analyzing all the above mentioned issues, proposes a cloud-enabled virtual environment named CLEVER that could be used on most of the different cloud contexts previously cited: it specifically aims at the design of a VI management layer for the administration of private clouds infrastructures but, differently from the other middleware existing in the literature, it also provides simple and easily accessible interfaces for enabling the interaction of different “interconnected” computing infrastructures and thus the ability of deploying VMs on such heterogeneous clouds. The concept of interface is also exploited for integrating security, contextualization, VM disk image management and federation functionalities made available from higher level software components (i.e. the modules that should lie in the highest layer of the stack of Fig. 1).

As will be described in the following, other key concepts related to our implementation refer to high modularity, scalability and fault tolerance of the whole middleware: thus the software layer is able to easily adapt itself to the physical infrastructure modifications which can occur (even when one of its components crashes or goes down) while the Cloud Computing infrastructure is working. Furthermore, by means of such kind of interface our CLEVER middleware is ready to interact with those high-level cloud components providing dynamically resources scheduling and provisioning.

The rest of the paper is organized as follows: Section II briefly explores the current state-of-the-art in Cloud Computing and existing middleware implementations, while Section III critically analyses the features of such cloud middlewares and motivates the need of designing and implementing a new one. Section IV provides an overview of the CLEVER's features, which are then deeply discussed in Section V where also a logical description of each middleware module is reported. Section VI contributes some details on the CLEVER prototype implementation and finally Section VII concludes the paper.

II. RELATED WORKS

This Section describes the current state-of-the-art in Cloud Computing analysing the main existing middleware implementations and evaluating their main features. A deep comparison among such middlewares and ours will be presented in Section III.

As previously introduced in Section I and already stated in [4], cloud management can be performed at the lowest stack layer of Fig. 1 as Virtual Infrastructure Management: this type of cloud middleware includes OpenQRM [5] and OpenNebula [6].

The project OpenQRM [5] is an open-source platform for enabling flexible management of computing infrastructures. Thanks to its pluggable architecture, OpenQRM is able to implement a cloud with several features that allows the automatic deployment of services. It supports different virtualization technologies managing Xen, KVM and Linux-VServer VEs. It also supports P2V (physical to virtual), V2P (virtual to physical) and V2V (virtual to virtual) migration. This means VEs (appliances in the OpenQRM terminology) can not only easily move from physical to virtual (and back), but that they can also be migrated from different virtualization technologies, even transforming the server image.

OpenQRM is able to grant a complete monitor of systems and services by means of the Nagios tool [10] which maps the entire openQRM network and creates (or updates) its corresponding configuration (i.e. all systems and available services). Finally, OpenQRM addresses the concepts related to High Availability (HA) systems: virtualization is exploited to allow users to achieve services fail-over without wasting all the computing resources (e.g. using stand-by systems).

OpenNebula [6] is an open and flexible tool that fits into existing data center environments to build a Cloud computing environment. OpenNebula can be primarily used as a virtualization tool to manage virtual infrastructures in the data-center or cluster, which is usually referred as Private Cloud.

Only the more recent versions of OpenNebula are trying to supports Hybrid Cloud to combine local infrastructure with public cloud-based infrastructure, enabling highly scalable hosting environments. OpenNebula also supports Public Clouds by providing Cloud interfaces to expose its functionalities for virtual machine, storage and network management.

Still looking at the stack of Fig. 1, other middlewares work at an higher level than the VI Manager (High-level Management) and although provide high-level features (external interfaces, security and contextualization) their VI management capabilities are limited and lack VI management features: this...
type of cloud middlewares include Globus Nimbus [7] and Eucalyptus [8].

Nimbus [7] is an open source toolkit that allows to turn a set of computing resources into an IaaS cloud. Nimbus comes with a component called workspace-control, installed on each node, used to start, stop and pause VMs, implements VM image reconstruction and management, securely connects the VMs to the network, and delivers contextualization. Nimbus’s workspace-control tools work with Xen and KVM but only the Xen version is distributed. Nimbus provides interfaces to VM management functions based on the WSRF set of protocols. There is also an alternative implementation exploiting Amazon EC2 WSDL. The workspace service uses GSI to authenticate and authorize creation requests. Among others, it allows a client to be authorized based on Virtual Organization [VO] role information contained in the VOMS credentials and attributes obtained via GridShib[11].

Eucalyptus [8] is an open-source cloud-computing framework that uses the computational and storage infrastructures commonly available at academic research groups to provide a platform that is modular and open to experimental instrumentation and study. Eucalyptus addresses several crucial cloud computing questions, including VM instance scheduling, cloud computing administrative interfaces, construction of virtual networks, definition and execution of service level agreements (cloud/user and cloud/cloud), and cloud computing user interfaces.

III. MOTIVATION

CLEVER aims to provide Virtual Infrastructure Management services and suitable interfaces at the High-level Management layer to enable the integration of high-level features such as Public Cloud Interfaces, Contextualization, Security and Dynamic Resources provisioning.

Looking at the middleware implementations which act as High-level Cloud Manager [7], [8], it can be said that their architecture lacks modularity: it could be a difficult task to change these cloud middlewares for integrating new features or modifying the existing ones. CLEVER instead intends granting an higher scalability, modularity and flexibility exploiting the plug-ins concept. This means that other features can be easily added to the middleware just introducing new plug-ins or modules within its architecture without upsetting the organization.

Furthermore, analysing the current existing middlewares [5], [6] which deal with the Virtual Infrastructure Management, we retain that some new features could be added within their implementation in order to achieve a system able to grant high modularity, scalability and fault tolerance. Our idea of cloud middleware, in fact, finds in the terms flexibility and scalability its key-concepts, leading to an architecture designed to satisfy the following requirements:

- Persistent communication among middleware entities.
- Transparency respect to “user” requests.
- Fault tolerance against crashes of both physical hosts and single software modules.
- Heavy modular design (e.g. monitoring operations, managing of hypervisor and managing of VEs images will be performed by specific plug-ins, according to different OS, different hypervisor technologies, etc).
- Scalability and simplicity when new resources have to be added, organized in new hosts (within the same cluster) or in new clusters (within the same cloud).
- Automatic and optimal system workload balancing by means of dynamic VEs allocation and live VEs migration.

Table I summarizes the features of CLEVER vs other cloud middleware implementations. CLEVER is able to manage in a flexible way both physical infrastructures composed of several hosts within a cluster and physical infrastructures composed of different “interconnected” clusters. This task is performed ensuring fault tolerance while operations are executed, exploiting particular methods which allow the dynamic activation of recovery mechanisms when a crash occurs. Furthermore, due to its pluggable architecture, CLEVER is able to provide simple and accessible interfaces that could be used to implement the concept of hybrid cloud. Finally, it is also ready to interact with other different cloud technologies supposing that their communication protocol or interfaces are known.

Looking at the table I, we believe the existing solutions lack a cloud VI able to implement all the characteristics of each row: the design of a new middleware able to satisfy all such requirements is the main goal we intend to pursue in this work, as deeply described in Section V.

IV. CLEVER OVERVIEW

Our reference scenario consists a set of physical hardware resources (i.e. a cluster) where VEs are dynamically created and executed on the hosts, considering their workload, data location and several other parameters. The basic operations our middleware should perform refer to: 1) Monitoring the VEs behaviour and performance, in terms of CPU, memory and storage usage; 2) Managing the VEs, providing functions to destroy, shut-down, migrate and network setting; 3) Managing the VEs images, i.e. images discovery, file transfer and uploading.

Looking at Fig. 1, such features, usually implemented in the Virtual Infrastructure Management layer, have been further reorganized according to the hierarchical structure of our physical infrastructure: the lowest tier of the stack has been split into two different sub-layers named respectively Host Management and Cluster Management. Although CLEVER is ready to interact with software components lying in the High-level Management layer of the same stack, our description specifically points out the lowest layers of the picture, describing both the set of components needed to perform VEs management at host level and the set of components required to perform VEs management at cluster level.

Grounding the design of the middleware on such logical subdivision and taking into account the satisfaction of all the above mentioned requirements, the simplest approach to design our middleware is based on the architecture schema
A comparison of CLEVER features vs other cloud middleware implementations

<table>
<thead>
<tr>
<th>Middleware</th>
<th>OpenQRM</th>
<th>OpenNebula</th>
<th>Nimbus</th>
<th>Eucalyptus</th>
<th>CLEVER</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fault Tolerance</strong></td>
<td>Yes, High Availability</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes, CM fault tolerance and XMPP session fault tolerance through the distributed DB</td>
</tr>
<tr>
<td><strong>Modularity</strong></td>
<td>Yes, plug-ins structure</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes, plug-ins structure and heavy modular design</td>
</tr>
<tr>
<td><strong>Cluster Interconnection</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes, by means of the interfaces provided by the CM</td>
</tr>
<tr>
<td><strong>Hybrid Cloud support</strong></td>
<td>No</td>
<td>Yes, differentiated driver to interact with external clouds</td>
<td>Yes, EC2 Backend</td>
<td>No</td>
<td>Yes, differentiated plug-ins through the CM interfaces to support different external clouds</td>
</tr>
<tr>
<td><strong>Remote Interfaces</strong></td>
<td>No</td>
<td>No</td>
<td>Yes, EC2 Web services API and Nimbus WSRF</td>
<td>Yes, EC2 Web services API</td>
<td>Yes, multi-purpose customizable interface through the CM</td>
</tr>
<tr>
<td><strong>Monitoring</strong></td>
<td>Yes, exploiting Nagios</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes, distributed cluster Monitoring performed by each HM</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes, by means of its flexible communication channel and modular architecture</td>
</tr>
</tbody>
</table>

Table I

TABLE I

A comparison of CLEVER features vs other cloud middleware implementations

<table>
<thead>
<tr>
<th>Middleware</th>
<th>OpenQRM</th>
<th>OpenNebula</th>
<th>Nimbus</th>
<th>Eucalyptus</th>
<th>CLEVER</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fault Tolerance</strong></td>
<td>Yes, High Availability</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes, CM fault tolerance and XMPP session fault tolerance through the distributed DB</td>
</tr>
<tr>
<td><strong>Modularity</strong></td>
<td>Yes, plug-ins structure</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes, plug-ins structure and heavy modular design</td>
</tr>
<tr>
<td><strong>Cluster Interconnection</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes, by means of the interfaces provided by the CM</td>
</tr>
<tr>
<td><strong>Hybrid Cloud support</strong></td>
<td>No</td>
<td>Yes, differentiated driver to interact with external clouds</td>
<td>Yes, EC2 Backend</td>
<td>No</td>
<td>Yes, differentiated plug-ins through the CM interfaces to support different external clouds</td>
</tr>
<tr>
<td><strong>Remote Interfaces</strong></td>
<td>No</td>
<td>No</td>
<td>Yes, EC2 Web services API and Nimbus WSRF</td>
<td>Yes, EC2 Web services API</td>
<td>Yes, multi-purpose customizable interface through the CM</td>
</tr>
<tr>
<td><strong>Monitoring</strong></td>
<td>Yes, exploiting Nagios</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes, distributed cluster Monitoring performed by each HM</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes, by means of its flexible communication channel and modular architecture</td>
</tr>
</tbody>
</table>

depicted in Fig. 2 which shows a cluster of $n$ nodes (also an interconnection of clusters could be analysed) each containing a host level management module (Host Manager). A single node may also include a cluster level management module (Cluster Manager). All these entities interact exchanging information by means of the Communication System. The set of data necessary to enable the middleware functioning is stored within a specific Database.

![Fig. 2. General Cloud middleware Architecture](image)

The Communication System is the channel used to enable the interaction among the middleware components. In order to grant the satisfaction of our requirements, it should offer: decentralization (i.e. no central master server should exist) in a way similar to a p2p communication system for granting fault-tolerance and scalability when new hosts are added in the infrastructure; flexibility to maintain system interoperability; security based on the use of channel encryption.

V. ARCHITECTURE DESIGN

We believe that the best solution to implement our proposed solution is to take advantage of the XMPP protocol [12], while the Database has to be implemented in a distributed
fashion. Such design choices lead to the integration within the middleware of an XMPP Server and a distributed DBMS (as will be described in the Section VI). Since the XMPP Server also exploits the distributed database to work, the solution enables an high fault tolerance level and allows system status recovery if a crash occurs.

When the middleware start-up phase begins, each software agent has to establish a connection to the XMPP server. The first agent booted will be the HM that, once connected to the server, will start a timer whose initial value is randomly chosen. When such timer expires, the HM will check the number of CMs available on the cluster. If this number is less than a given value, chosen evaluating the total number of hosts, the HM will start the initialization process of a new CM. In this manner, a set of CM agents will be created, in order to achieve the needed level of fault tolerance. This mechanism will persist while the middleware is up, allowing to maintain a proper number of instantiated CMs (more details will be provided in the following Subsections).

As previously introduced, the first instantiated CM agent will remain in an active state (it will perform the operations about the VEs management) while the others will be in a monitoring state (they will receive the requests and will monitor the number of CM agents present in the cluster). If the active CM agent crashes, in order to maintain the service, another CM will change its own state to active. To select the active CM we adopt an algorithm similar to the one previously described for the CM instantiation.

The following part of this Section provides more details on the internal composition of the software components deployed on the cluster’s hosts (already pointed out in Fig. 2) analysing the Host Manager (Subsection V-A) and the Cluster Manager (Subsection V-B).

### A. Host Manager description

According to Fig. 1, this middleware component belongs to the lowest level of the stack and consists of the set of modules represented in Fig. 3: the red ones, depicted on the left part of the picture (i.e. Image Loader, Hypervisor Interface and Network Manager), represent the “low-level” components exploited from the yellow ones, depicted in the left part of the same picture (i.e. Host Coordinator Migration Manager and Monitor). The Coordinator can interact both with low-level and high-level components while the Migration Manager only communicates with the low-level components.

HM Coordinators, during their interaction, could operate as follows: 1) Streaming the collected information; 2) providing the collected information on demand; 3) sending a specific notification (alert) when a pre-determined condition is verified. All the HM Coordinators have to interact exploiting the persistent XMPP connection made available through the Coordinator CM Interface; the other middleware components, in order to perform temporary peer-to-peer communications, can attend an ephemeral XMPP session connecting themselves to an “utility room”.

1) **Migration Manager:** This module receives requests submitted from the central control of the HM Coordinator, and communicates with Image Loader, Hypervisor Interface and Network Manager in order to accomplish the Virtual Environment migration among the Hosts belonging to the “cloud”.

When a Virtual Environment migration has to be performed, two different hosts will be involved: Source and Destination Hosts (i.e. source and destination HM). The Cluster Manager component (described later in Paragraph V-B), which manages the migration at high level, will interact with the Coordinators of these HMs, forwarding them two different requests: ReqSend to notify the Source HM that a migration of a certain Virtual environment, running on that host, will occur; ReqRecv to notify the Destination HM that a Virtual Environment (already managed by another HM) will migrate to that host.

Such requests are sent to the HM Coordinator and then delivered to the Migration Managers which will interact each other exploiting a temporary XMPP connection in the “utility room”. By means of the established communication channel, all the parameters necessary to execute the migration will be exchanged between hosts. Such parameters will be employed to provide the right migration guidelines to the Image Loader, Hypervisor Interface and Network Manager components. Once the migration has been correctly performed, the Migration Manager will notify the event to the HM Coordinator, in order to propagate the event itself to the other middleware entities.

2) **Monitor:** The main task this module accomplishes, regards resource usage monitoring for a certain host. Collected information are then organized and made available for the HM coordinator which exploits them for managing the node. This entity provides two different types of information: data regarding hardware resources (e.g. the machine architecture) and the available Operating System of the monitored host; data about the host workload including Memory, CPUs and local disk storage usage, obtained interacting directly with the Operating System.
3) **Image Loader:** In our scenario, in order to provide the highest level of flexibility and interoperability among the middleware components, it is necessary to design an Abstract Storage System. We assume that, when a Virtual Environment has to be saved within such Storage System it must be registered: thus, an unambiguous identifier should be created according to the structure (Cluster ID, protocol://path) where Cluster ID is univocally associated to a cluster whose registration has been executed, protocol belongs to a set of different values (FILE, NFS, SFTP, FTP, GFAL etc) and path represents the effective data location. When a file registration is performed, two different situations may occur: the file is located in an external site (thus only its reference will be registered); the file has to be uploaded on the local cluster storage.

Each host of the cluster may share a part of its local storage, in order to create a distributed file system of the cluster (managed by the Storage Manager in the CM). Such system, even exploiting heterogeneous storage methods and data transfer protocols, has to be accessed in a transparent fashion by the middleware components (in particular the Image Loader).

The Image Loader component interacts with the HM Coordinator in order to provide the Virtual Environment image which is needed by the local hosts, in particular by their Hypervisor Interfaces. The Image Loader receives (from the Migration Manager) the identifier of the Virtual Environment which will be instantiated and will be able to supply it to the Hypervisor Interface. In order to assure high flexibility, different plug-ins should be employed, each designed to support a different data access/transfer method.

4) **Hypervisor Interface:** This module acts as a middleware back-end to the host hypervisor: since different virtualization technologies could be employed on each host, a flexible and customizable plug-in structure has to be developed: each plug-in will interact with a different hypervisor. One of these plug-ins, for instance, will interact with the hypervisor exploiting the LibVirt interface, while another one will interact with Vmware through another specific interface. Using this approach, an abstraction layer on top of the hypervisor will be deployed, granting high modularity on the component and thus flexibility.

By means of the “abstract” interface created, the Migration Manager component will be able to employ a common set of commands for managing the Virtual Environments. Such set of commands will include: create, destroy, list and suspend.

5) **Network Manager:** This component, exploiting the Operating System interface, mainly accomplishes two different tasks. The first one refers to the capability of gathering information about the host network state: available network interfaces (including their type and details regarding TCP/IP stack level 2-3); usage state of Transport level resources and available port range (firewall).

The second task the Network Manager should perform, refers to the host network management (OS level), according to the guidelines provided by the HM Coordinator, i.e.: Dynamic creation of network bridges, Dynamic creation of network routing rules and Dynamic creation of network fire-walling rules.

6) **Coordinator:** This is the core of the Host Manager, whose function is the coordination of the HM components. Coordinator represents an interface to the Cluster Manager and communicates with all the other HM components (DBUS/JMS). In particular, it interacts with the Monitor to collect statistics about system state and workload; interacts with Image Loader, Hypervisor Interface and Network Manager to manage Virtual Environment execution (start, stop etc); interacts with the Migration Manager, to forward migration requests coming from the CM (as detailed in the Migration Manager description).

7) **CM initiator:** As previously exposed, when the middleware components are instantiated, the first agent booted is the HM agent. The CM initiator within the HM agent, once connected to the server, will start a timer whose initial value is randomly chosen. When such timer expires, the module will check the number of CMs available on the cluster. If this number is smaller than a predefined value depending on the total number of hosts, the CM initiator will start a new CM. In this manner, a set of CM agents will be created, in order to achieve the required level of fault tolerance.

**B. Cluster Manager description**

Unlike the HM component already presented, the Cluster Manager module lies in the second layer of the reference stack depicted in Fig. 1, thus representing an high level module used to manage the Virtual Environment allocation in the cluster’s hosts. CM accomplishes these tasks interacting with all the HM coordinators instantiated within each host of the cluster exploiting an XMPP communication.

![Fig. 4. Host Manager internal components](image)
cluster manager modules: Performance Estimator for instance, in order to perform its task, needs both historical data on cluster state and an own set of information. As well as the Performance Estimator, other components of the Cluster Manager could require this kind of interaction with the Database. This leads to the necessity of integrating within the Database Manager the skill of performing DML (Data Manipulation) and DDL (Data Definition) operations.

State information are collected from the CM Coordinator interacting with each HM coordinator of the cluster (on each host the coordinator receives such information from the monitor). Once received, such information are forwarded to the Database Manager, which will write them on the database. Data exchange between Database Manager and other CM Components, will be based on XML documents. These latter will be generated by the CM coordinator according to a specific schema, will be parsed by the Database Manager and finally written on the Database despite of the underlying DBMS. Similarly, when information has to be retrieved, they will be extracted from the Database, encapsulated within XML documents and finally filtered exploiting the XQuery language [13].

2) Performance Estimator: The role of this component is mainly related to the analysis of the set of data written on the CM database from the Coordinator, in order to compute and provide a probable trend estimation of the collected measures. Since the Coordinator writes on database the series of measures referred to cluster resources usage (in a specific observation time), the Performance Estimator should be able to predict the future state of cluster usage. Values computed from the Performance Estimator are then formatted according to a specific XML schema, encapsulated within XML documents and thus sent to the Database Manager. This latter will parse the received data and will write them on the Database, adding (if necessary) further data container within the XML structure, exploiting the available set of DDL operations.

3) CM Monitor: Once the CM initiator has been started, a certain number of CMs will be available: the first one instantiated will be in an “active” state, that is it will perform the operations about the VEs managing, while the others will remain in a “monitoring” state, that is they will be able to receive the requests and monitor the number of CM agents present in the cluster. If the active CM agent crashes, in order to maintain the service, a new “active” CM must be present thus an election algorithm has to be exploited. As well as described for the CM initiator (in the HM), each CM monitor will start a timer whose initial value is randomly chosen. When such timer expires, the module will check if an active CM exists: if this event is not true, the CM will elect itself as the “active” CM; otherwise any operation won’t be executed.

4) Storage Manager: As previously described, all the files representing the Virtual Environment, in order to be employed within the Cloud Infrastructure, have to be registered (and if necessary uploaded) within the Cluster Storage System using an unambiguous identifier. The Storage Manager is used to perform the registration process of such files and manage the internal cluster distributed file system.

5) Coordinator: As already described for the HM, the CM coordinator can be considered as the core of the Cluster Manager. As the figure shows, in order to communicate with the other middleware components, the CM coordinator exploits three different interfaces: Client Interface, used to interact with both User Interfaces and high-level control entities (Cloud Brains); External Cluster Interface, used for the interconnecting different CM and thus different clusters; HMs Interface, as previously described, exploited for communicating with all the HM coordinators.

Each of these interfaces will be connected to a different XMPP room, in order to separate different communications. It performs high level operations like Virtual Environment deploying, taking into account the system workload and features of each host (OS, hypervisor, etc.). Moreover, through Client Interface or External Cluster Interface, the Coordinator provides information about the Cluster State, collected through the HM Interface.

VI. IMPLEMENTATION

This section highlights some of the details involved in our CLEVER implementation. In the current status of the work, a primitive prototype, integrating some features of the whole architecture, has been developed: software modules implementing the basic functionalities of the Host Manager and Cluster Manager have been written, allowing middleware components interaction by means of the XMPP protocol: the management of all the problems related to identification, access, monitoring and control of hardware resources in the Cloud Environment have been thus addressed in the current implementation.

As we already stated, CLEVER presents many modules which interact each other through the communication channel in which all controls, commands and data are confined. The communication channel is based on the XMPP protocol whose features refer to P2P communication with TCP/IP connectivity model. This allows an efficient and strongly distributed interaction among all the middleware modules in conjunction with an high degree of scalability.

According to the description reported in the Section V, the CM interface of the Host Manager (3) and the HMs interface of the Cluster Manager (4) have been implemented developing two specific software modules which act as XMPP clients exploiting the Smack [14] set of libraries. By means of such software modules, our middleware components are able to communicate each other exploiting the XMPP facilities provided by the Openfire server [15].

In order to manage VEs allocation, our implementation of the HM components, includes a specific software module whose aim refers to hypervisor interaction: such software practically implements the Hypervisor Interface represented in Fig. 3 linking the Libvirt [16] set of libraries.

Regarding to the identification, it was obtained with the introduction of a representation in which it is possible to uniquely identify each resource of the cloud. A resource is
not only a VM, but also a physical resources, a XMPP client that participates in global communication, storage resource, etc. The main components enabling the XMPP communication are: 1) SERVER which is responsible for managing the connections as XML streams (XML Stream), authorizing the entities involved in such stream and finally routing XML data between appropriate entities; 2) PEER that is the entity which contacts the server to gain the access to any specific XML Stream. It needs the unique ID, the access credentials and the communication scope. Last requirement is performed using the concept (rather consolidated in the chat messaging systems) of “chat-room”. According to our middleware implementation, any middleware module involved in whatever type of communication will be a PEER, belonging to any specific chat-room. (In our early implementation, we configured just one main chat-room, to simplify the work).

Since many resources can simultaneously connect to the SERVER on behalf of each authorized PEER, their own identifier JID will be organized as follows: JID = [node"@"]domain["/"resource]. XML stream permits to drive different data type. This is the reason that leads us to introduce a simple protocol able to bring the whole set of commands and control we need. The XMPP messages are characterized from the following tags:

- **Type**: it represents the command type that is necessary either to send to any different type of targeted resource: vm, host; or to perform a specific action i.e. polling;
- **Command**: it is the command itself and it can either add, remove the VM resource or in case of polling it can be the value of: “requestDetails”.
- **Step**: indicates the request state; it may be “request” or “reply”.

We underline the protocol can carry on any type of request, command and control in a transparent and easy way (i.e. the input argument values for a command). Further effort is being devoted to implement the remaining modules of the proposed middleware. Extensive measurement and tests will then be carried on to quantitatively assess its performance and reliability.

**VII. CONCLUSIONS AND FUTURE WORKS**

In this work we described the design principles and the preliminary prototype implementation of our cloud middleware named CLEVER: unlike similar works existing in the literature, CLEVER provides both Virtual Infrastructure Management services and suitable interfaces at the High-level Management layer to enable the integration of Public Cloud Interfaces, Contextualization, Security and Dynamic Resources provisioning within the cloud infrastructure.

Furthermore, thanks to its pluggable design, CLEVER grants scalability, modularity, flexibility and fault tolerance. This latter is accomplished through the XMPP protocol, which provides P2P communication according to a TCP/IP connectivity model, allowing efficient and strongly distributed interaction among all the middleware modules, in conjunction with an high degree of scalability. Since the current CLEVER prototype implementation is still at a preliminary stage, we are already working to further extend the middleware functionalities according to the reference model described in this paper. Moreover, a set of tests is being executed to obtain a comprehensive set of experimental results to deeply evaluate the behavior of the middleware and its performance.

**REFERENCES**


[2] Sun Microsystems, Take your business to a Higher Level - Sun cloud computing technology scales your infrastructure to take advantage of new business opportunities, guide, April 2009.


