Creating a high availability, load balancing cluster using cloud technology

Volanis Stamatis, Ftylitakis Nikos, Kainourgiakis Giorgos

Abstract
Some of the most challenging and critical issues regarding modern web systems are flash crowds, denial of service attacks and server infrastructure failure management. In order to address these issues, the architectural approach of designing large scale web sites infrastructure, should focus on high availability and scalability of these systems. In this report we will make an effort to provide a solution and more specifically we will present our work on creating a high availability, load balancing cluster using cloud technology.

1. Introduction
In order to describe the problematic state that large scale web sites eventually facing, it is useful to describe at start two core concepts, flash crowds and denial of service attacks (DoS attacks). The term Flash crowds refers to the situation when large number of people require service in a very short period of time. These requests have an direct impact to web site's servers load, overall network traffic, and eventually resulting denial of service to new clients requests or increment to overall system response time. The outcome of this, is user frustration and a direct reduction to web site's credibility, events that could be critical for business or whatever the website's purpose is.

DoS attacks in the other hand are client requests from machines that intent maliciously to congest the server infrastructure, attempting to result eventually denial of service or increment of response times. This behavior is everyday life for web sites administrators and yet another problem that they have to deal with. One infrastructure design that could answer to this issue is the ability of the system to scale on
demand, so that it can cope with load. That means that somehow new servers should be added when the number of requests rise (scale up), but also in contrast the number of active servers also should be reduced (scale down), when the load congestion is low. Scalability is a core concept for large scale systems if quality of service and high availability are concerned, in order have efficient use of resources.

Another issue that should be addressed is internal server failures. This problem affects also the high availability of the web service, since in large scale systems a load balancing block is common and the possibility of failure also exists. This means that an internal load balancer failure will result the overall failure of the system, resulting denial of service to new users. That means that the system infrastructure should have a failover mechanism that the load balancer but also servers can fail without affecting the performance or the availability of the system. In the next section we will discuss and propose an architecture that addresses to these problems.

2. Architecture

2.1 Clusters

In order to provide an elegant solution to these issues and especially to server failover handling, we considered using an high-availability cluster, so that we could insure that when a failure occurs to the load balancer, a redundant server will continue to serve the website. The main idea is that the redundant server will always monitor the load balancer, since they are part of the same cluster, in order check for failures, and if this is the case to continue load balancers work. This is the classic implementation of a load balancer for high availability, in terms of server failure.

Our perspective in failover handling doesn’t follow the classic approach, but we rather choose to assign the role of load balancer to any of the cluster’s members. The idea is this the first server that appears in the cluster automatically becomes load manager and all the other servers, that join later, are candidate load managers in case of failure. A novel solution to the load manager authority transition problem is to use priority numbers to all cluster members. The server with the higher priority automatically becomes the new load balancer, in case of failure, by acquiring an extra ip address from the ex-load balancer, while still serves its open sessions.

The deployment of a cluster also responds relatively easily to a much needed load monitoring mechanism, since the cluster members have the ability of exchanging messages. For that, the load balancer at any given time, is aware of the exact state of any cluster member, which means that the load manager is aware of the load for any given server making the load balancer capable of applying any desirable load balancing strategy. New sessions are sent to the least congested server, in order to balance the load equally. More details about the systems and load balancing design are presented later.

2.2 Cloud environment
For the implementation of scalability we need a system that can create, manage and terminate easily virtual machines (VMs), in order to distribute load on the system, and also to have the potential to scale itself easily. We decided to deploy a cloud infrastructure, since it provides a simple API for virtual machine and storage management, and also is capable to scale itself. The load balancer can scale up the cluster by starting new VMs and distribute the load to a greater number of servers. In contrast when the average load is low, the load balancer stops sending new sessions to particular VMs and eventually when their load becomes zero, can terminate their operation.

2.3 System's design

Initial state Every node of the system is running on a cloud virtual machine, it's part of the same cluster and has a number that we call preference. The initial number of nodes in system's architecture is two, the load balancer and a simple serving node. The load balancer is a role, so that any node of the cluster can eventually be a load balancer and also has the highest preference. The simple node is communicating with the load balancer and transmits load metrics, which is the essential input for load balancing strategy. When the load of the simple node is low the state of the node is "green", which implies that the node can serve more sessions.

Increased load When the load of a simple node raises above a specific threshold, then the state of the node becomes "red", which means that the node is congested and the load balancer shouldn't assign new sessions to it. At this point the load balancer should start a new node, so that can continue to redirect new session to it. The same procedure is applied at any case that a node's state becomes "red", which means that the load balancer start a new node. This strategy was one of many that we could implement, by tweaking some parameters of the system. The preference that the load balancer assigns to new nodes, decreases every time it starts one, so if the load balancers preference is 100 the first's node is 99, the second is 98, the third is 97 and so on.

Internal server failure The case when a node terminates its operation for any reason, is called internal server failure. When a simple node fails, we decided that the load balancer should'n start a new node, because if the load increases to some node, the load balancer will scale up anyway. The critical situation is when the load balancer fails. In this case the cluster node with the highest preference becomes the new load manager, which means that it continues to serve it's open sessions and at the same time it becomes load balancer.

Decreased load When a node's load decreases the state of the node eventually becomes "white". There many parameters that could be defined, when a node becomes "white", like open connections or overall load. In this case the load balancer chooses to terminate the operation of the node, in order to scale down the system and ensure that manages efficiently the resources of the system.

2.4 Tools
2.4.1 Eucalyptus Cloud

Our cloud infrastructure is based in Eucalyptus Cloud, an open source implementation of a private cloud computing platform, which is compatible with Amazon EC2 and S3 services. The selection of Eucalyptus was based in authors's already existing familiarity with the platform and also the compatibility with Amazon web services, which means this scaling platform can easily integrate to Amazon itself. We will make an effort to present a short description of Eucalyptus's elements and functionality in order to provide a background in terminology that will be used later.

2.4.2 Eucalyptus Concepts

A key concept of cloud computing are Machines Images (MIs). These images are ready to run operating systems configurations, that can easily be deployed and run in VMs. The running images are called Machine Instances and they can easily run or terminate from the Eucalyptus platform API. Another concept is Image Bundling, which means that any user can create custom images that can use in the future, as cloud MIs. This is critical because the cloud user most often desires his own instance configuration and can achieve it with image bundling.

2.4.3 Eucalyptus Elements

Eucalyptus architecture contain a set of five modular simple elements that can easily scale, in order to meet owner’s needs. These elements are:

- Cloud Controller (CLC), provides the user interface of the cloud, communicates with Cluster Controllers and manages new instance allocations and information.
- Walrus Storage Controller (WS3), stores user data and machine images.
- Elastic Block Storage Controller (EBS), is used for the creation of persistent block devices that can be mounted on running machines in order to gain access to virtual hard drive.
- Cluster Controller (CC), mainly receive requests from Cloud Controller to allocate MIs and then decides which Node Controller will use which MI to create running instances. It also manages any virtual networks that instances use and routing their traffic.
- Node Controller (NC), is the cloud element that the VMs actually run as running instances from the MI that the user intents to use.

2.4.4 Linux HA

The cluster implementation is based in the Linux-HA project, which maintains a set of building blocks for high availability cluster systems including a Cluster Communication layer and a Cluster Resource Manager (CRM). The two core block of Linux-HA project are Heartbeat which is the cluster messaging block and Pacemaker which is a cluster resource manager that the project uses for its cluster implementation.
2.4.5 Heartbeat

Heartbeat provides communication and cluster membership layer of Linux-HA, which is implemented as a daemon process that executes in cluster members and provides cluster infrastructure services. This allows a cluster member to know about the presence or absence of peer processes on other cluster members and to easily exchange messages with them.

2.4.6 Pacemaker

In order to be useful to users, the Heartbeat daemon needs to be combined with a cluster resource manager which has the task of starting and stopping the services (IP addresses, web servers, etc) that cluster will make highly available. The cluster resource manager typically associated with Heartbeat is Pacemaker. In the following figure we present the overall systems architecture.

![Figure 1: System architecture](image)

3. Deployment and Configuration

3.1 Eucalyptus Cloud

The project was implemented over an Eucalyptus Cloud Infrastructure, which was deployed for testing purposes in the Computer Center of University of Crete. Eucalyptus services was installed in a 6 core Intel Xeon X5650 Cpu , 4Gb ram Server. The server was configured as both Cloud (CLC), Cluster (CC), Node...
(NC) and Storage (SC) controller. Eucalyptus and his components was compiled from their latest sources available on the websites of the relevant projects. The Operating System used is Ubuntu Server Maveric (x86_64). The components that were installed are:

Eucalyptus cloud v2.0.2 (CLC, CC, NC, SC)

OpenSSL Vtun. VTun provides the method for creating Virtual Tunnels over TCP/IP networks and allows to shape, compress, encrypt traffic in that tunnels.

- libvirtl virtualization api. Libvirt allows you to access hypervisors running on remote machines through authenticated and encrypted connections.
- iscsid Open-iscsi daemon. The iscsid implements the control path of iscsi protocol, plus some management facilities.
- tgtd SCSI Target Daemon. Used to provide iscsi target service to a network.
- kvm QEMU Emulator. QEMU is a generic and open source machine emulator and virtualizer.
- ISC Dhcpd and dnsmasq. DHCP and caching DNS server.
- bridge utils. Ethernet bridging administration
- iproute/iproute2/iptables

Eucalyptus storage functionality is implemented using iSCSI, which is a layer 3 technology, over layer 2 AOE (ATA-over-Ethernet) which is default. The main reason for this choice was that AoE is not as scalable as iSCSI since AoE is not a routable protocol. Also, iSCSI offers user based access and encryption, which are two good reasons for using while offering professional solutions. The KVM is using libvirt's virtio for attaching iSCSI volumes and network cards with the VMs. The DHCP server is configured to allocate free ip addresses to the VMs, while dnsmasq is used for relating hostnames with ip addresses. The ip addresses are allocated from an 11 addresses pool using MANAGED mode. This implies that, for a new VM instance, a private ip is allocated which is bridged with the real ip. The virtual and real interfaces are bridged and iptables rules are introduced in order to map the real ip to the private and vice versa.

The Eucalyptus Cloud has enabled WS_Security for providing secure connections between all Eucalyptus Controller, a feature which was not needed in our case since we are using a single server solution, but it is a must choice for security in a more generic case that, for example, Nodes are seperated from CLC and CC.

3.1.1 VM's images creation

After configuring and initializing Eucalyptus, we bundled various i386 and x86_64 images for testing purposes. The distributions that were tested are: Debian 5.0 x86_64 and i386, Ubuntu Server Maverick x86_64 and centos 5.3 x86_64. All three were bundled without problems and we chose Ubuntu Server Maverick for deploying our project. We use a shared volume for storing configuration files and steps, so the first step for creating the working images is to attach a volume to the "clean" VM:

```bash
  euca-attach-volume -i i-3E170839 -d /dev/vda vol-5F090651
```
Inside the volume, there is the latest version of cloud-init tools (0.6.1) which is need for booting an altered image, latest versions of euca2ools and their dependencies(1.3.1) and needed scripts and configuration files for setting up the system. For Ubuntu, there is a need to use rm -rf /etc/udev/rules.d/70* for removing udev files which are attaching a specific mac address with an interface. Packages that are installed through Ubuntu's apt-get: python-dev libssl-dev swig help2man unzip rsync make wget curl heartbeat pacemaker ldirectord libmailtools-perl.

After installing the packages, we extract our project files as additional resources at /usr/lib/ocf/resource.d/heartbeat directory. Needed configuration files are /etc/heartbeat/ha.cf which is the heartbeat configuration file, /etc/heartbeat/authkeys which contains the SHA1 hash used for authentication between cluster nodes, /etc/heartbeat/ldirectord.cf which is the loadbalancer configuration file, the /etc/sysctl.conf which includes the required kernel settings and /etc/rc2.d/S21heartbeat which is the node startup script.

The above procedure is the only procedure needed for creating a cluster node for our project. After creating the image, we need to create a new Eucalyptus bundle from it. Through inside the instance, we use:

- euca-bundle-vol -c \${EC2_CERT} -k \${EC2_PRIVATE_KEY} -k \${EC2_PRIVATE_KEY} -u \${EC2_USER_ID} --ec2cert \${EUCALYPTUS_CERT} --no-inherit --kernel eki-5EE62385 --ramdisk eri-D7221860 -d /mnt -r x86_64 -p team4proj -s 2048 --exclude /mnt
- euca-upload-bundle -b team4proj -m /mnt/team4proj.manifest.xml
- euca-register team4proj/team4proj.manifest.xml

where kernel eki-5EE62385 is the same kernel with the initial clean instance and eri-D7221860 is a ramdisk created with similar procedure with /boot/initrd.img file. No-inherit states to reject previous cloud configuration.

The startup script initiates the node to be a part of the cluster. If the node is the first one to run, it automatically becomes the load balancer of the cloud. Any next nodes, even if the first node is terminated and another one took its place, are becoming simple cluster nodes. So the init script handles different configuration at startup for load balancer or simple nodes. After the initial configuration, any modifying configuration while our cluster is operational, is executed through heartbeat and pacemaker scripts.

Load Balancer initiates two extra virtual interfaces, one acquires ip 10.10.10.1 which is used for private cluster communication and the other acquires the private ip address 172.19.1.30. The cloud controller routes the real external ip of the service to the private ip address 172.19.1.30 for the incoming connections and masquerades it with the real ip for the outgoing connections. It also initiates the heartbeat cluster configuration, needed one time only.

A simple node also initiates two extra interface, one virtual that acquires a unique ip at the subnet 10.10.10.0/24 and the other is a second loopback interface with the virtual ip 172.19.1.30/32. All nodes are configured with kernel ip forwarding enabled and ARP advertisements declined kernel based. The purpose
of the above network configuration concerning the private ip 172.19.1.30, is that the nodes that don't really have this private ip should be able to see traffic destined to it and respond. Note that all subnets are used for testing purposes only. It can be easily extended in a larger cloud infrastructure with more available VMs.

3.2 Heartbeat

Heartbeat is responsible for providing the cluster infrastructure (communication and membership). It is initiated through S21hbinit init script. It reads /etc/heartbeat/hb.cf for its startup settings, which include cluster nodes, communication topology and which features are enabled. In our case, Heartbeat uses UDP broadcasting for its messages, but also supports many communication methods including multicast, unicast and serial line. Considering that the nodes are able to communicate which other, the autojoin any directive states that all nodes can try to authenticate to each other. Heartbeat also supports autojoin none which requires all nodes to be stated inside the ha.cf file. We also configure various other time settings like deadtime which configures how much time should decide a node dead, initdead which is used to set the time that one starting node is declared dead (for loading purposes), keepalive which sets the interval between heartbeat packets et. al. Another useful configuration directive is the api_auth directive, which defines what uids/gids are allowed to use specific, user created applications that connect to Heartbeat cluster.

On Load Balancer's first run, heartbeat is also initialized. When a node (including Load Balancer first node) joins Heartbeat cluster, there is a membership created bound to a specific UUID, which is randomly created while running Heartbeat for first time. The cluster configuration is propagated within cluster nodes, so every node has whole configuration everytime. If a node becomes unresponsive or is terminated, all other cluster nodes are eventually, in timeouts that are defined at ha.cf. The Heartbeat cluster includes a quorum voting technique, which uses each cluster member to identify itself by voting, in the form of a simple write to a shared vote, or quorum disk. The combination of heartbeat and quorum disk minimizes the risk of split-brain cluster states. Split-brain clusters occur when the two nodes think they are both in the correct cluster state, so both access the shared storage. Split-brain states create the highest risk of database corruption, so this is the functional core of the cluster.

The Load Balancer startup script also initate resources configuration needed to achieve our project goals. Heartbeat uses Pacemaker to utilize the resource functionality. Pacemaker and its resources we deployed and used are discussed later in this report. Heartbeat recognizes the following resource categories:

- primitive: Basic resources, configured to run to a single cluster node.
- master/slave: Resource that runs to all cluster nodes with a master/slave characteristic.
- clone: Resource that runs simultaneously to all cluster nodes.
- group: Grouped resource, used to group one or more of the above categories in order those
resources to run simultaneously.

Each resource has its own properties that are initiated through Load Balancer's first run and can be altered any time through any node from the cluster. Such properties include resource monitor interval, timeout and resources' attributes. For resources that do not run on all nodes, Heartbeat uses a preference system that defines the order of the nodes that will try to run the resource. The order is fully defined in the configuration, can be changed realtime and is followed by every cluster node.

3.3 Pacemaker Resource Agents

As described above Pacemaker is a resource manager component whose main interaction point is the Resource Agents (RA). But first of all let's define what a resource is. As a resource we can consider any kind of service or information that can be acquired at will from the system. That could be the invocation of a program that executes an operation, the execution of a shell script or even simply the extraction of a single value from the system. Each resource usage is wrapped by a Resource Agent to create a uniform and standardized procedure of usage.

The standardization is defined and provided by the Open Cluster Framework (OCF). OCF project's purpose was to define standard clustering APIs for some basic capabilities and was founded at the early 2003. Nowadays it's a part of Linux Foundation and is supported by a number of companies such as Red-Hat and SuSE. The development is mainly held by the Linux-HA project whose part is Heartbeat and Pacemaker that were used in our project. As the OCF specification defines, a Resource Agent has to have some specific characteristics. First of all it has to implement 5 basic operations:

- start: enable or start the given resource
- stop: disable or stop the given resource
- monitor: check whether the given resource is running (and/or doing useful work), return status as running or not running. It's also the main periodic function of the resource where an interval can be set.
- validate-all: validate the resource's configuration
- meta-data: return information about the resource agent itself (used by GUIs and other management utilities, and documentation tools)

Secondly, each of those operations is obligatory to be idempotent (because of multiple checks and simultaneous calls during startup and membership changes to Heartbeat). Lastly most resource agents are coded as shell scripts, this, however, is by no means a necessity. The defined interface is language agnostic.

The benefit from having a Resource agent is the global maintenance. A call to any function is probed to the whole cluster (if needed, it's a configurable feature), thus synchronizing all the nodes. Also any change
in the Resource Agent parameters is also probed. This makes easier the creation of management tool and the maintenance by only one individual. So complying with the specifications, we created 2 custom Resource Agents apart from the 2 already implemented that were used in our project. Those four Resource Agents will be analyzed in the next paragraphs.

3.3.1 SysMonitor

The SysMonitor was created to run at every node to monitor its workload. For this purpose we created metrics to extract average workload during the period of the interval. Afterwards the metrics are combined with configurable logic rules to conclude to a 3 state model that describes the workload of the node.

3.3.1.1 Metrics

This resource agent runs every a certain interval defined at the time that the Resource Agent is set to the Pacemaker. On every run it calculates workload metric that will result the overall state of the node. Specifically the metric comes from CPU, Ram, Upload speed, Download speed, number of nodes in the Heartbeat membership.

3.3.1.1.1 CPU

At every run of the Resource Agent, it extracts the CPU info from the file: /proc/stat. The file tells you the cumulative CPU time that's been used in various categories, in units of jiffies. A jiffy is a very small amount of time which in computer systems is used to signify the amount of time passed for operations held by the CPU. From this file we are mainly interested in 3 values:

- utime: user mode jiffies
- stime: kernel mode jiffies
- tim_total: total time passed in jiffies

So from these 3 values we can extract the time the CPU was working by summing user mode time and kernel mode time. Consequently subtracting that number from the total time we also have the idle time of the CPU. On each run, we hold the previous values of the working and total time. Thus we extract the average workload in presentence of the total time passed from the previous run.

3.3.1.1.2 Ram

Ram values are extracted using the command "free". Among the values of the command's output there are a) total Ram of the system b) amount of Ram used in total, c) amount of Ram allocated for buffering, d) and the amount the is used as shared. At any time the real available amount of Ram can be calculated by subtracting from the total used ram the sum of Buffered and Shared together. If then divided by the total system’s Ram we have the presentence of available Ram.
3.3.1.3 Network

Using "ifconfig" command we can acquire the total number of inbound and outbound traffic measured in bytes which is divided in sections according to the network interface that we use. On each run we save the current values and on every subsequent call the current values are subtracted from the previous and afterwards divided by the interval time between the calls. This operation generates the average upload and download speed during that period.

3.3.1.4 Number of Nodes

The number of nodes is extracted using the Pacemaker Tools. Generally, Pacemaker messages are probed in XML format. Thus using its tools we can acquire information from it. Querying the "cibadmin" tool we get the state of all Pacemaker parameters, nodes, and membership. This info combined with the Cleaning Tool that we will discuss later ensures us that at any given time we will be seeing valid information for the nodes.

3.3.1.2 Overload and Underload Rules

At the beginning of the project we wanted to define somehow rules, combine them with the metrics resulting a categorization of the state of the nodes. The first implementation had hardcoded limits that arbitrarily selected and set to 80% of Ram or CPU. Although this could be a working example, this approach was not even acceptable because various users define differently according their needs their overload and underload conditions.

The above issue led us to the creation of a small Boolean logic language with number comparisons and predefined variables:

- Predefined variables:
  - cpu: holds values between [0,100] which is the presence of cpu workload
  - ram: holds values between [0,100], the presence of ram currently used
  - download: expressed in Kbytes/s
  - upload: likewise “download”
  - numOfNodes: the integer number expressing the number of nodes currently in Heartbeat Membership

- Comparison Operands: >, >=, ==, <=, <

- Boolean Operands: &&, ||

- Parenthesis support

It was created using Antlr Workstation. A platform where LL grammars can be created and thereon it gives to the developer the option in which programming language it will be used generating the appropriate
files. In our case all the programs were written in C++ so we used the C version of the generated Parser and Lexer. The use of the three above leads us to the creation of a small but useful language able to express from simple logic, to complex rules parameterized according to the current number of nodes the system has at that time.

One important aspect of this is that the user of our system runs his Virtual Machines in a Cloud Environment thus paying as much as the number of VMs used. Hence, by creating rules that apply to a certain number of nodes you can create upper limits to the number of nodes leading to user defined restriction of the cost.

An example of the grammar expressing the overload state can be the following:

- \((\text{cpu} < 80 || \text{ram} < 80 || \text{upload} > 100 \&\& \text{download} > 9000)) \&\& \text{numOfNodes} < 10\)

This examples defines that a node will be considered overloaded if CPU workload exceeds 80% or Ram exceeds 80% of Upload speed combined with download speed exceed the above values. But all this logic is limited by a maximum number of nodes needed.

3.3.1.3 Combining them all together

At this point we have the metrics and a Rule language. These two gave use the ability to introduce the concept of 3 node health states. White state will be representing the underload state of a node, Green the normal state and Red the overload state.

Now putting it all together a user of our system defines the Underload Rule and the Overload Rule expressed in the above language and passed as arguments to the Resource Agent definition (under the keywords underloadRule and overloadRule correspondingly). SysMonitor Resource Agent has one more parameter named after “nic” which sets the network interface name used.

By initializing the Resource Agent to our system it starts running according to the interval defined in it. On every run extracts the metrics, checks the metrics with the define rules. First it runs the check if the node is Overloaded. If it fails it continues with Underload Rule so if that fails too it means that the node is at Green state.

These state values are saved as Pacemaker attributes. Each node can define any new value it needs to which is addressed by string key and contains string value. Any such attribute belongs to the node it created it. So the name of the attribute and the value is accompanied by the node it and is probed to all the nodes currently in the cluster.

This feature was used to probe the values we wanted. The attribute that defines the state of a node is called “isLoaded”. This attribute is later on read by the CloudManager Resource Agent (will be discussed below). Along with that value we probe the nodes Virtual IP used to characterize a node as load balancer and the Cloud IP used by the Node Cleaner deamon to terminate any zombie nodes from the Cluster.

To sum up, using this Resource Agent we achieved to provide user configurable rules for which are the overload and underload conditions, the state definition and the probing of the values to the whole...
3.3.2 CloudManager

CloudManager is the Resource Agent that was created as the main component that will be taking actions to the cloud state and configuration. Although it will be running at every node, only the node that is currently load balancer will be able to take actions and execute scripts. In more detail it’s once more a Resource Agent so hence that it also have start, stop and monitor functions with configurable interval times. Also it is initialized and defined using 3 arguments. The load balancer IP (lbip), the minimum number of nodes (minNodes) and the maximum number of nodes (maxNodes) At every call it is responsible to acquire the list of the nodes along with their states. Then checks if any of the nodes had changed state. If any change is found, the program calls an appropriate external script with parameters that characterize the node.

The notion of external scripts was introduced to also make the actions taken for cloud management configurable. 3 scripts were created, one for each health state a node can have. That leeds us to instructionScriptRed, instructionScriptGreen, instructionScriptWhite, 3 bash scripts that are called with 6 arguments.

- the name of the node
- its heartbeat UID
- the IP in cloud
- minimum number of nodes
- current number of nodes
- maximum number of nodes

So when a node changes state, the script corresponding to that state script is called with the above arguments. All this is enough for the user to make his own decisions as to what will be the behavior of the system.

3.3.2.1 Node Cleaner

As mentioned previously, CloudManager acquires the list of nodes and their values. By default Heartbeat doesn’t delete the configuration of offline nodes. So that list was inaccurate and we needed to fix this issue to continue. For that reason we created Node Cleaner. A program that runs before CloudManager and cleans Heartbeat registry from invalid membership registrations. Thus it keeps in the system only the necessary information for the live nodes. At the deletion of any invalid node, using the Cloud IP saved by the SysMonitor Resource Agent, checks if that node is still running at the Eucalyptus system. If any is detected, it is terminated.
3.3.3 IPaddr

This script was one of the already implemented Resource Agents that were part of Heartbeat. It manages IP addresses and can add an IP alias, or remove one. This is used for managing the IP failover of the Load Balancer. Our approach was that a node becomes the Load Balancer whenever it has a specific IP (10.10.10.1) so we needed a way to handle over that IP to other nodes when the current Load Balancer fails. When configuring, we define the IP that we want to handle, in which network interface it will be placed and most importantly a preference list. The preference list defines each node’s priority to take over that IP.

3.3.4 ldirectord

A Load Balancing daemon that uses ipvsadm to route the connections to the nodes. Monitors the health of the real servers by periodically requesting a known URL and checking that the response contains an expected response. Uses scheduling algorithms such as round robin (rr), weighted round robin (wrr) where new connections are assigned to each real-server in turn and weighted least connections (wlc).

4. Evaluation and testing

In this section we will provide some cases that we have tested in order to evaluate the functionality of our system. Each test is considered individual since if they are applied to the same system by any order they have the same results. Before we begin our test we made one implementation of the CloudManager ‘s scripts that as mentioned are called for every node that changes state. In our case, the script for the White state was set to just echo the fact that it was called along with its arguments. The script that caught up most of our interest was the one for the Red state when a node gets overloaded. Of course this is just a user case, our user case and is applicable to any changes needed.

The Red script when called, instantly initiates the creation of a new Virtual Machine. When that happens, extracts the current time and saves it as an attribute to the node that became the reason for this script to be called. Lastly it sets the weight of the node for the rooting purposes to 0, thus preventing any new connections to be done to than node. Having the time of that incident that took place, we can and do prevent the case were that node goes from Overloaded to Normal and back again in sort amount of time. In our case we force it not to create any new Instance for the same overloaded node unless a minute passes, the node becomes normal and the get overloaded again.

Lastly the Green script does 2 operations. Firstly, just like White Script, echos the fact that it was called along with its arguments. Its most important operation was that it restores the weight of the node to the rooting table. If a node goes from White state to Green it’s not important since it makes no difference. But if a node that was previously in Red state (overloaded) and finally calms down to Green (normal) it restores its rooting that was previously blocked form the Red script.

Also for the purpose of our experiment we setup an HTTP Apache server to the initial image. That
was a choice as much for augmenting the result as for testing the ipvsadm which is a Level 4 kernel load balancer. Here we have to make a note that since it is in kernel mode, it can be used with other than HTTP protocols. That Apache server was used to serve a working site of ours.

So to begin our test, we run a new Eucalyptus instance. Upon startup all its initialization takes place, Heartbeat and Pacemaker are initiated, connected and configured with our selected Resource Agents. Now we are ready to proceed to the tests.

4.1 Terminate a node

First test was to create new nodes and at an arbitrary point of time to terminate one or more of them. The challenge here was to prove that any registration of offline nodes was cleaned from the Heartbeat registry. Responsible for that operation is Node Cleaner so it was time to validate its operation. The test was held successfully, resulting a cleaned Heartbeat configuration as many time as we killed any node.

4.2 Terminate the load balancer

Next test, the termination of the Load Balancer node. The expected behavior was to watch IP failover mechanism defined in IPaddr Resource Agent to reassign the IP to the node next in the preference list. As soon as that node takes the IP, it would mean that it was automatically selected as the new Load Balancer. That results that from now on that node would be responsible to take the maintenance actions for the cloud though CloudManager Resource Agent. Once again the test was successfully completed with all the expected outcomes. As a continuation of that test was to reinitiate the dead node. That resulted the previously dead Load Balancer to take back its IP, become Load Balancer again and continue where it stopped.

4.3 Overload a node

Final test was to overload any node and watch our system expand through the Red script instantiation. To do that we overloaded the CPU of a random node, regardless if it was a Load Balancer or not. Since the CPU topped to 100%, it changed its state to Red. CloudManager RA successfully detected the change of the state and called Red script. That indeed resulted the creation of a new Virtual Machine instance. At the same time, the rooting was disabled which could easily be tested if we tried to refresh the web page of our site. Because we can see which node served the request we saw that the overloaded node was no longer serving the requests. At the same time, when stopping and restarting the heavy workload before the end of the minute of waiting, no other node was created. Finally when stopping the workload permanently, the rooting is restored.

4.4 Notes
At this point we have to emphasize that at any test that was held, there was no packet lost. Except from the case where the Load Balancer fails where we have to way for 3 seconds for the IP to be considered dead and another 2 seconds for the rooting.

5. Discussion - Future Work

In this project we had the opportunity to study about a recently rising section of Informatics which tries to take advantage of the huge increase of processing power as well as all server and network resources in general. Taking as an example the recently deployed and successful cloud infrastructures provided by Amazon EC2 or Google App Engine, there are many benefits for individuals, small or large organizations to invest on cloud infrastructures for their IT needs. Reduced cost (for individuals or small organizations mostly), flexibility, mobility, automation, scalability, availability and even energy saving are some of the benefits of cloud computing. All those cloud characteristics are constantly improving since more and more applications and solutions are introduced. In this work we are using opensource software to deploy a Highly Available, Scalable, Load Balancing cluster over the opensource edition of the Eucalyptus Cloud. Heartbeat is used for providing the cluster infrastructure and Pacemaker for managing the cluster resources. We also introduce our own Pacemaker resources using OCF specification for the scripts and C++ for deploying the programs required. Heartbeat API is also used for creating the utility that informs all nodes about its state and gathers the other nodes information. The required software and configuration is contained in new Eucalyptus images which are used for our testbed. All test cases produced expected results.

There are some possible adjustments to both architecture and deployment, in order to improve functionality or usability of the setup. First of all, we can introduce a "loadbalancerNumber" attribute, to support multiple Load Balancers in the cluster. This could be a minor change in the deployment since the new Load Balancers will have the same characteristics as the current Load Balancer, but combining multiple LBs will introduce more diversity in the cluster, since a single Load Balancer can eventually become very busy. We can also use a dns approach in order to split the requests to all Load Balancers in the cluster.

As another addition, we could utilize the already implemented DRDB resource agent, which uses DRDB that introduces network mirroring of blockdevices, similar to raid 1. As an alternative, a cluster filesystem as GFS (Global FileSystem) can be used in order to provide a fast, scalable and high throughput filesystem for our nodes.

Another problem we faced was to minimize the configuration of our whole setup. There are utilities like hb_GUI, written in python, that already allow cluster administrator to monitor, configure and administer the cluster, the cluster nodes and its resources. As a next step, it could be modified to support our developed resources and inherit our introduced attributes and states. This may require further improvement and cleanup of our scripts, in order to make sure that everything is administered from the GUI.
6. References
[3] Linux High Availability project (Linux-HA), http://www.linux-ha.org/wiki/Main_Page