UNIVERSITÀ DI PISA

FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI

CORSO DI LAUREA SPECIALISTICA IN TECNOLOGIE INFORMATICHE

Tesi di laurea

OCTOPUS: EXPERT CLOUD MANAGEMENT FOR VIRTUAL MACHINES

Candidato

Marco Mura

Relatore

Dott. Antonio Cisternino

Controrelatore

Prof. Laura Ricci

Anno Accademico 2011/2012

Contents

Introduction

Virtual Machines have become a standard unit of resource allocation for cloud environment, and virtualization has been used for implementing cloud infrastructures because of its natural ability to decouple the physical hardware from logical servers. The increasing density of computational power allows packing a significant number of virtual machines on a single node, leading to an exponential growth of the number of logical servers to be manages by the cloud infrastructure. To investigate smart policies to administer such a large number of virtual machines we have developed Octopus, a lightweight system for scheduling virtual machines on a cluster of hypervisors.

Octopus has been implemented using Microsoft Hyper-V in order to exploit the WMI interface to control the hypervisor programmatically with the F# programming language. The original goals was:

1. to provide simple users a virtual machine manager to manage only their virtual machine, with total independence from the system administrator that doesn't need to care about hosted virtual machines, and in total security because each user can see only its virtual machines and not the others' ones. Moreover, users can create their virtual machines from a set of pre-configured templates, in order to have a virtual machine ready to be used in just few minutes;
2. to design a system capable of moving virtual machines across different computing nodes in order to optimize the workload and pack computations to save energy by turning off nodes

More recently, our investigation has focused on the possibility of using expert systems to express complex policies and to govern this ever-increasing set of virtual machines. For this reason, we have embedded the CLIPS expert system inside Octopus in order to rely on a full rule-based expert system engine to define the resource management policies: the Octopus code asserts facts about VMs in the CLIPS systems and rules access system primitives exposed as functions invokes by triggered rules. The well-known RETE algorithm ensures a fast execution of policy while ensuring the ability to define policies that may even contain conflicting rules.

Octopus has been presented to the Microsoft workshop “Cloud Futures 2010” in Redmond, WA, USA, in the International Supercomputing Software (IIS) 2010 in Hamburg, Germany, and more recently to the GARR workshop “Calcolo e storage distribuito” in 2012 in Rome, Italy. In all the presentations attendees have shown interest in the technology and possibly its deployment.

1 State of the art

1.1 Virtual Machine

A virtual machine [1], generally understood as hardware virtualization [2], is a software implementation of a machine that can execute programs like an emulated physical machine.

The concept of hardware virtualization was born in the 1960s, with the purpose to partition large mainframes for better utilization. The first implementation of a virtual machine was designed and realized by IBM with its experimental computer M44/44X. Based on an IBM 7044 the M44/44X simulates multiple 7044 using both hardware and software capabilities. This machine indeed implements a machine virtualization very similar to the one we know today.

With its evolution, the concept of virtual machine has been divided into two main categories: the complete system platform virtualization, capable of running an operating system (and all the software compiled to run on it), and the process virtualization [3] (or application virtualization), designed to emulate only single applications.

Process virtual machines are used to execute programs compiled for a different architecture, interpreting or re-compiling the compiled instructions in a version compatible with the current architecture. The most common use of a process virtualization is implemented by the Java Virtual Machine [4] [5] and the .NET Common Language Runtime [6] [7], which executes programs compiled for an intermediate language that is different to the native assembly language one.

This thesis regards the implementation of hardware virtual machines manager, so process virtual machines are not further discussed.

The complete system platform virtualization, or hardware virtualization, implements system virtual machine that emulates an entire hardware architecture. Sometimes a system virtual machine is used with the purpose of providing a platform to run programs where the real hardware is not available, for example to execute software available only for obsolete hardware; other times it is used with the purpose of providing multiple instances of the physical architecture for a better computing resource allocation or for running different operating system simultaneously.

The hardware virtualization is managed by a software called “hypervisor”; the hardware can be entirely emulated, so the hypervisor runs as a process of a host operating system, or the emulation can be hardware assisted (in modern CPU, this technology is called Intel VT-x [8] or AMD-V [9]) to provide virtual machines direct access to the hardware and better performance. In this last case, virtual machines operating system run side-by-side with the host operating system, which generally works also as hypervisor.

1.2 System Virtual Machine Managers

Currently there are many hypervisors which support hardware virtualization, for both hardware-assisted and software-based virtualization. The most common are Microsoft Virtual PC, Microsoft Hyper-V, Parallels for Mac, Oracle VM VirtualBox, VMWare (Server, ESX, Workstation…) and Xen.

All of them was born with different purposed, different host/guests supports and different features, but evolving they now offer similar properties. Virtual PC (now replaced by Hyper-V) and VirtualBox are dedicated for desktop environments, Xen is only for server environments, and Parallels and VMWare

offers both desktop and server products. Hyper-V was only for server environments, but with Windows 8 it replaced the Virtual PC product.

Aside from the environment where they are meant to run, all of them supports both 32 and 64 bit virtual machines, with hardware-assisted support, and with a graphical interface which allows users to connect to the virtual machines and using them, like they are using a real hardware.

The common limitation of all this software, from a point of view of this thesis, is that they have an interface that manages all the virtual machines hosted on that hypervisors, and none of them are meant to organize them and offer different view per-user. They are meant to be used only by administrators.

The idea for this thesis is exactly to create a virtual machine manager that allows simple users (and not administrators) to create, manage and use their virtual machine independently. Users will be able to see only their virtual machines, and cannot interfere with the other's one. The server administrator can manage them all, but he doesn't need to: he is required only for maintain the server host and he doesn't need to care about user's virtual machines.

Also the virtual machine interface, meant to be used by simple users which didn't care too much about low levels properties, in particular its allocation, could use more than one host server to allocate virtual machines. Indeed, developing the idea for this virtual machine manager, we decided to make it a scheduler of virtual machines in a cluster of hypervisors, which is capable of deciding where a virtual machine should be allocated. Users will be able to use virtual machines through the virtual machine manager interface, which will provide them the network information for connecting to the requested virtual machine.

During the implementation of this thesis, software with a similar idea born on the market. These are the Microsoft System Center Virtual Machine Manager

(VMM) and the IBM Platform Cluster Manager (former Platform Cluster Manager from Platform Computing).

Anyway, both of them are thought to create a central point administration for system administrators in order to manage all the virtual machines from different hypervisors in a single interface, and again none of them are meant to be used by simple users.

2 Octopus: a virtual machine scheduler

2.1 The project

The purpose of this thesis was designing and implementing a Virtual Machine Manager that manages (schedules) virtual machines from different hypervisors, creating a single environment where the virtual machines can be automatically migrated in case of needs from one hypervisor to another.

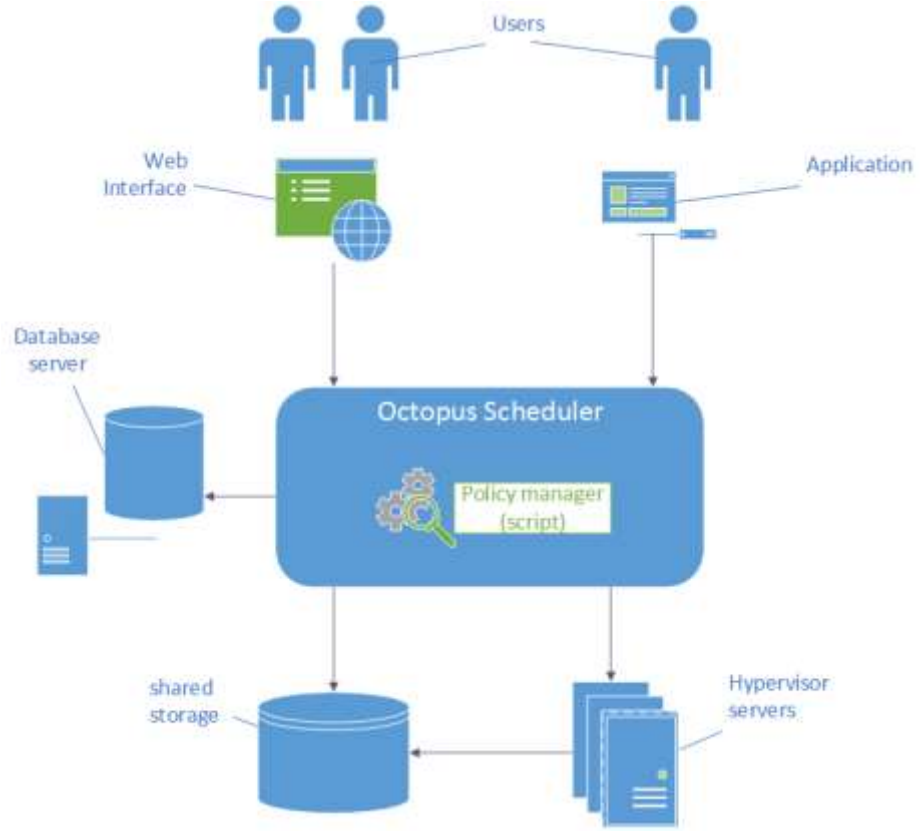
Also, we imagined to design the Virtual Machine Manager to be used by users, and not only to administrators. Users will see only their virtual machines, and each user will be able to fully administrate them: deleting a vm, change its settings, suspending/stopping a vm, etc; according to possible quotas.

Users will also have the ability to create virtual machine using preconfigured templates. In this way, they will be able to instantiate a fully working operating system in few minutes, instead of creating an empty virtual machine and waiting for the time needed the Operating System to be installed. A preconfigured VM could be useful to create basic virtual machine fast and easily (with just the OS and drivers), test environments or any other kind of machine you need to use as soon as you can.

With this precisely feature, where users can easily create a virtual machine usable in a very short time, and can fully administrate it suspending or deleting it whenever he wants, we wanted to create the concept of “Disposable Virtual Machine”: a virtual machine that you can create when you need it, even for a short period of time, and then throw it away.

We decided to name this Virtual Machine manager “**Octopus**”.

In this thesis will be often referring to Octopus sometimes as a Virtual Machine Manager.



The hypervisor servers are several third-party virtual machine managers, independently, with their own private storage space and a connection to a large shared storage space (i.e., a NAS or SAN).

The Policy manager is a script running inside the Octopus Scheduler. It operates in order to react as events, like “a user want to create a virtual machine” or “a hypervisor node need to be shut down”. It makes actions, like turning on or off virtual machines, hypervisors, migrating virtual machines between hypervisors, etc.

Virtual Machine can be migrated between hypervisors easily, thanks to live migration features native of the hypervisor itself [10] [11] [12] [13]. If live migration features are not supported, VM can still be migrated in a slower way

moving all the data from one hypervisor to another; in this scenario, the virtual machine disks must be allocated in the shared storage, in order minimize the transferred data and take as less time as possible.

Summarizing, having a Virtual Machine manager that manages several hypervisors server and allow users to administrate their virtual machines have different benefits:

- it results in a better resource allocation of the entire cluster
- it allows a better power management of the cluster, turning off the unused nodes
- it lighten the cluster administrator workload, avoiding him to manage the users' virtual machines
- it's adaptable and scalable, allowing the cluster administrator to fit the scheduler policy through a script language

2.2 Power consumption

According to a report dated 2007 of the U.S. Environmental Protection Agency (EPA) [14], in 2006 the energy used by the USA's servers and data centers is estimated about 61 billion kilowatt-hours (kWh), which is about 1.5% of the total of the nation's electricity consumption, for a total electricity cost of about $4.5 billion.

In the same document, it states that this energy consumption is more than doubled between 2000 and 2006, and estimates that in 2011 will double again for an electricity consumption of more than 100 billion kWh.

According to recent studies, in particular to a report of the 2011 [15], the electricity used in USA data centers in 2010 was significantly lower than predicted by the EPA's 2007 report, and between 2005 and 2010 it increased by only about 36% instead of doubling.

These numbers explain the rising attention that energy efficiency is getting nowadays.

Although in 1992 the U.S. EPA launched the Energy Star [16] in order to incite hardware manufacturer to develop devices that consumes less energy, using techniques of high energy efficiency and power savings states (like sleep states), the computer hardware consumption is still significant.

Virtualization was always been a key for energy savings and Green Computing [17] [18] [19], and this is also a key argument for this thesis.

This thesis is going to illustrate how to better use a server cluster improving its efficiency keeping powered on only the nodes that are really needed, and keeping powered off the other ones.

Node powers will be managed by Octopus policies, implemented via script. We are going to show an example of power management keeping powered on only the nodes where there are virtual machines allocated into, but administrator can personalize or rewrite the entire script to adapt the Virtual Machine manager "Octopus" to any scenario.

2.3 Security considerations

Giving a user the ability to create their own virtual machine by themselves and using them without any administrative interaction could became a security issue.

It is necessary that the Virtual Machine Templates that are configured to be used in the system will be “secure”. The definition of secure depends on the administration of course… It could be a closed/unconnected cluster, so anything could be secure, or it could be a cluster in a big corporate connected with the rest of the network, so source to attack to the data from the internal.

The Octopus Administrator could configure the Virtual Machine Templates to be already joined on the Active Directory domain with can enforce security policies. It could also avoid that users will use the Virtual Machines with administrator privileges, providing them only an accessible “simple user” account.

Any other considerations about the security risks about running a computer inside a network, especially when you have administrator account, are already widely discussed in literature [20] [21] [22].

About the Octopus design, the potential risks are:

- abuse of the system, allocating too much resources and taking down nodes for overloading
- unauthorized use of the Octopus functions (i.e., a user can operate on other users’ virtual machines)
- stealing other user’s credentials hacking the database or sniffing the communications

The first point is handled by the cluster administrator. It's his duty to adjust the policy script in order to prevent users to allocate too much resources, and especially to avoid a node overloading.

The second one is implemented using an authentication system, that prevents users to call any Octopus function without a valid authentication token. In the Octopus Scheduler interface, every function takes as first parameter an authentication token, and if it's not valid or the associated user doesn't have the right to perform that operation, the function fails.

About the third point, we know that handling password represent a very delicate point in a software, and implementation of password management requires refinement. To prevent users to hack the database and read/use the stored credentials, for instance, it should be enough storing an HASH of the password instead the password in clear. In this way, the password can be matched with the HASH, but the HASH cannot be used to authenticate someone. Also transmission via TCP/IP must be encrypted in a Socket Secure Layer (SSL) to prevent users from sniffing data (intercepting the traffic) or to execute man-in-the-middle attacks.

We decided to not focus too much on this third point, to focus on the other innovative aspects, and also to facilitate debugging during the software development. Anyway, this kind of improvement doesn't impact at all in the Octopus architecture design and could be performed easily as a future development.

2.4 Technologies used

The main decision of we had to take about something was with no doubts about the hypervisor to use. Theoretically it would be even possible to use different kind of hypervisors simultaneously, but this is not on this scenario at this time.

We chose to use Microsoft Hyper-V because its ease to use via code, through its Windows Management Instrumentation (WMI) interface. Thanks to the Hyper-V WMI, in fact, it's possible to execute with a function any possible Hyper-V feature, and the WMI infrastructure allows applications to execute its functions in both local and remote computers.

The Hyper-V live migration feature became available only with the Hyper-V Server 2008 R2, released in 2009. We starting developing Octopus in 2008, so we didn't integrate the live migration feature. For this reason, migration in implemented in the "slow" way and the shared storage is an essential requirement.

Also, because of the Hyper-V live migration requires the Failover Clustering feature, which is available only on the Enterprise or Datacenter edition of Windows Server, we decided to not add this feature when was released. We'd like to see Octopus running on the cheapest configuration possible, and for the records it exists a Windows Server edition called "Microsoft Hyper-V Server 2008 R2" which is totally free.

Because of this, we decided to use the .NET Framework for programming the software. The .NET Framework was not only for manufacturing coherence (same Microsoft products), it have a native interface with WMI [23] that, added to

the advantages of a high level programming language makes it the best choice for this project.

More specifically, we chose to write all the code in F#, a pretty young programming language developed by Microsoft and recently became part of the .NET language. It's a multi-paradigm programming language, mainly functional, inspired by ML. We decided to use this because, aside that every language in the .NET Framework are virtually equivalent, the implementation of this software using a functional paradigm would have been more interesting from a software engineering point of view.

And for the database server, we chose Microsoft SQL Server. This is not really supported by any strong argumentation since any other database server would be the same. This is was just because the other choices was software from Microsoft.

2.4.1 Expert systems and CLIPS

As already mentioned, Octopus users are administrators of their virtual machines; they can create, suspend, change and delete them anytime. However, to prevent users to abuse of the system (creating too much virtual machines) there are policies that it needs to be implemented.

Speaking of Virtual Machine, the primary resource limit is the main memory (RAM) allocation: usually hypervisor hosts runs out their amount of main memory before finishing the disk space.

So, an allocation policy could be one that limits users to allocate more than a fixed amount of main memory, regarding of the number of virtual machine he wants to create. But this depends of the kind of virtual machine he wants to create, too. In an environment of "Disposable Virtual Machine" we think about virtual

machine that are not CPU intensive; maybe allocated for their special configuration to test a particular software or to create a small console for office purpose. But in a scientific environment where virtual machines are used for intensive calculations, CPU quotas are necessary too.

Since we cannot predict what kind of policy the Octopus administrator will need in its scenario, we first decided to create a script language to express simple policy. The language would be composed by events, actions and objects (users, hosts, virtual machines) with their properties.

With this language for instance, we would have an event for a new virtual machine creation where we could check the virtual machines already allocated by the user who is requesting the new creation, calculate if with the new virtual machine the quota limits would be violated, then deny or allow the request.

But except for quotas, there are other thing we decide to automatize. In an optical of power consumption, we would like to have an environment which is capable of use only the resource it needs and turn off the unused ones.

So, sometimes, Octopus should monitor the resource utilization of its nodes, try to defragment the virtual machine allocation, migrating them from an host to another, and shut down the unused hosts.

Before implementing the script language described above, we realized that in computer science it has been already invented something perfect to handle this problem: an expert system.

An expert system is a computer system that emulates the decision-making ability of a human expert. They are designed to solve complex problems by reasoning about knowledge, like an expert, and not by following the procedure of a developer as is the case in conventional programming. An expert system has a unique structure, different from traditional programs. It is divided into two parts, one fixed, independent of the expert system: the inference engine, and one

variable: the knowledge base. To run an expert system, the engine reasons about the knowledge base like a human.

We chose the expert system CLIPS, an implementation started in 1985 from the NASA and became a public domain software later. CLIPS is probably the most widely used expert system tools because it is fast, efficient and free. CLIPS incorporates a complete object-oriented language for writing expert systems and its user interface closely resembles that of the programming language LISP.

Like other expert system languages, CLIPS deals with facts and rules. Facts represent the system knowledge, which can be asserted changing the state of the system. For example, a fact can be the number of hypervisors available, or a virtual machine allocated.

The CLIPS command to assert facts is *(assert)* and its syntax is:

```
(assert <element>+)
```

For instance, to describe the hardware resources for the node HYPERVIS1, we could assert:

```
(assert (node-name HYPERVIS1))
(assert (node-isreachable HYPERVIS1 true))
(assert (node-isavailable HYPERVIS1 true))
(assert (node-ipaddress HYPERVIS1 192.168.1.11))
(assert (node-memorysize HYPERVIS1 16384))
(assert (node-memoryfree HYPERVIS1 10654))
(assert (node-cpucount HYPERVIS1 4))
(assert (node-cpuload HYPERVIS1 21))
(assert (node-diskfree HYPERVIS1 154312))
```

which create the fact that it exists a node named HYPERVIS1, which is reachable (turned on) and available, has 16Gb of total memory and about 10Gb free, 4 CPU with a total load of 21%, and about 150Gb of free disk.

And to describe a virtual machine, we could assert:

```
(assert (vm-name ed7ba470-8e54-465e-825c-99712043e01c
         VMMARCO1))
(assert (vm-owner ed7ba470-8e54-465e-825c-99712043e01c
         mura))
(assert (vm-hostname ed7ba470-8e54-465e-825c-99712043e01c
         VMMARCO1))
(assert (vm-server ed7ba470-8e54-465e-825c-99712043e01c
         HYPERVIS1))
(assert (vm-cpucount ed7ba470-8e54-465e-825c-99712043e01c
         2))
(assert (vm-memorysize ed7ba470-8e54-465e-825c-
         99712043e01c 4096))
(assert (vm-ipaddress ed7ba470-8e54-465e-825c-99712043e01c
         192.168.1.121))
(assert (vm-cpuload ed7ba470-8e54-465e-825c-99712043e01c
         6))
(assert (vm-state ed7ba470-8e54-465e-825c-99712043e01c 1))
(assert (vm-disksize ed7ba470-8e54-465e-825c-99712043e01c
         81920))
```

which create the fact that there is a virtual machine (with a GUID = "ed7ba470-8e54-465e-825c-99712043e01c") named VMMARCO1, with the hostname VMMARCO1, owned by the user "mura", running on the host HYPERVIS1, it has allocated 4Gb of main memory and 80Gb of disk space, and it has 2 CPU with a total load of 6%.

As the Facts are the knowledge of the system, the Rules represents the actions the system must take depending on its knowledge.

The CLIPS command to define rules is *(defrule)* and its syntax is:

```
(defrule <rule-name> [<comment>]
         [<declaration>]                ; Rule Properties
         <conditional-element>*
         =>
         <action>*
)
```

and the defined rule will be fired only if all the conditional elements are satisfied.

For instance, we could have a rule which states that, if a node have less than 1Gb of free memory, it should be set as unavailable. A possible rule for this scenario could be:

```
(defrule detachnode
         (node-name ?name)
         (node-isreachable ?name true)
         ?avail <- (node-isavailable ?name true)
         (node-memoryfree ?name ?mem)
         (test (< ?mem 1024))
         =>
         (retract ?avail)
         (assert (node-isavailable ?name false))
)
```

Note the tokens that begin with a question mark: these are variables, declared locally in the rule to be referred from a conditional point to another, or to an action.

In this case, we use “?name” to iterate all conditions to the same node name, “?mem” to store its free memory and then test it, “?avail” to store the fact “node-isavailable” itself in order to be retracted later.

Rules are “activated” (ready to run, or to be fired) when all the facts in their declaration part is asserted, and at least one of those fact must be asserted after the last execution of the rule itself. This is both for an efficient implementation of

the CLIPS's Rete algorithm, and to avoid that a rule will be fired forever without any fact change in the knowledge.

Let's see an example. Let our CLIPS environment be populated with the set of Facts for the node HYPERVIS1 we have previously seen.

Using the CLIPS command *(agenda)* we can see, without firing them, which Rules are activated and will run at the next *(run)* command. The command *(agenda)* doesn't return any rule because the declaration "(test (< ?mem 1024))" with ?mem taken from "(node-memoryfree ?name ?mem)" is not satisfied.

Now, assume we retract the fact "node-memoryfree HYPERVIS1 10654" and we assert the fact "node-memoryfree HYPERVIS1 900". Now the output of *(agenda)* is:

```
CLIPS> (agenda)
0      detachnode: f-1,f-2,f-3,f-10
For a total of 1 activation.
CLIPS>
```

It is telling us that the rule "detachnode" is activated, satisfied by the facts 1, 2, 3 and 10.

The execution of the command *(run)* will fire the detachnode, marking the facts 1, 2, 3, and 10 "already checked". In the case of the detachnode, a successive *(run)* wouldn't fire the rule again anyway because in its action the commands

```
(retract ?avail)
(assert (node-isavailable ?name false))
```

would have made impossible its activation removing a fact required in its declaration "node-isavailable ?name true". But even if without this, successive

executions of *(run)* wouldn't make the rule "detachnode" firing twice because, the second time, there wasn't any of the fact in its declaration list asserted new.

In CLIPS is also possible to define functions with the following syntax:

```
(deffunction <name> [<comment>]
    (<regular-parameter>* [<wildcard-parameter>])
    <action>*
)
```

that allows to write scripts easily, defining functions that performs generic operations or controls logic. For instance, we could define the following functions:

```
(deffunction getAllVmForOwner (?ownerName)
  (bind ?toReturn (create$))
  (bind ?factName vm-owner)
  (bind ?factz (find-all-facts ((?x ?factName)) (eq
        (nth$ 2 ?x:implied) ?ownerName)))
  (loop-for-count (?i 1 (length$ ?factz))
    (bind ?g (fact-slot-value (nth$ ?i ?factz)
         implied))
         (bind ?toReturn (append$ ?toReturn (nth$ 1
         ?g)))
  )
  (return ?toReturn)
)
```

This function return a list (in CLIPS called multified values) of all Virtual Machines GUID for the ones owned by the specified parameter. In particular:

- remember the syntax of vm-owner:

  ```
  (vm-owner <vmId> <owner>)
  ```

- create an empty multifield value with *(create$)*
- retrieve all the facts named ?factName (= vm-owner) which satisfy the condition "the second value of the fact must be equal to ?ownerName"

- iterate the retrieved facts and, for each one, append to the multifield value its first value
- return the created multifield value

Functions like *(bind)*, *(create$)*, *(find-all-facts)*, *(eq)*, *(nth$)*, *(loop-for-count)*, *(fact-slot-value)*, *(return)* are just some of the functions pre-defined in the CLIPS implementation. A full list is available on its specification.

The function *(append$)* was not available, and it was defined in this way:

```
(deffunction append$ (?m ?e)
  (insert$ ?m (+ (length$ ?m) 1) ?e)
)
```

The functions *(create$)*, *(nth$)*, *(length$)*, *(insert$)* and the new defined *(append$)* are used to create and manipulate "Multifield values". The Multifield is a CLIPS type that contains more than a value of different types, and it can be used as a list.

Some examples:

```
CLIPS> (create$)
()
CLIPS> (append$ (append$ (append$ (create$) uno) due)
         tre))
(uno due tre)
CLIPS> (length$ (append$ (append$ (append$ (create$)
         uno) due) tre)))
3
CLIPS> (nth$ 2 (append$ (append$ (append$ (create$)
         uno) due) tre)))
due
```

2.4.2 Microsoft Hyper-V

Microsoft Hyper-V is a Microsoft hypervisor initially developed only for Server products (Windows Server) but now available also in the workstation version of Windows 8 (only in some editions).

It allows creating and managing hardware-assisted virtual machines, using an architecture called Type-1 or Bare metal. The hypervisor runs directly on the hardware at the CPU ring -1 (a special ring introduced by Intel and AMD to support hardware virtualization), then the host operating system and all the virtual machines run at the CPU ring 0 that allows guests operating system to run unmodified in the virtualized environment.

Isolation between virtual machine (and the host OS) is implemented in terms of partition. The main partition is the one where the host OS is running and it have direct access to the hardware. The main partition, when a virtual machine is executed, spawns a child partition that does not have direct access to the hardware resources but it views a virtual view of them, in term of virtual devices. Any request to the virtual devices is redirected via the VMBus to the devices in the main partition, which will manage the requests.

Hard disks are virtualized using the VHD (virtual hard disk) image format, also natively supported on the recent editions of Windows.

It allows to store, in a single file, an entire virtualized hard disks with partitions information and file allocation tables.

Like a real hard disk, the Virtual Hard Disks are fixed in size; however, it can be allocated statically, using since its creation all the virtual hard disk size, or dynamically, which starts with the disk information only and grows up progressively when new data is written.

The Virtual Hard Disks can also be organized hierarchically: a virtual hard disk can inherit all the data from a parent VHD; in this case, the child VHD will contains only the “changed” data other than the new ones.

Thanks for parent/child VHDs, Octopus can implement the virtual machine templates.

A template is just a VHD with an operating system configured, drivers and applications, and a small Octopus service executed on the OS start to specialize the installation changing its hostname, configuring the user account and eventually configuring the networking.

In order to instantiate a virtual machine using that template, Octopus will instantiate a child VHD inheriting from the template one; so, all the changes needed to specialize the virtual machine are written in the child VHD and the virtual machine itself store its data in its child VHD.

This approach allows saving a lot of time instantiating a virtual machine (the operating system is already installed, so other application we want to have in the virtual machine): it requires only the time to create the virtual machine metadata on the hypervisor and to execute the code to specialize the operating system information.

Initially, it saves also a lot of space, because the changed needed to specialize the virtual machine are very small compared to the full operating system installation. However, the more the virtual machine is used, the more space its child VHD will occupy. Compared to standard allocations of virtual machine, this approach consumes at most a surplus of the size of the template, but in a scenario where virtual machine are used for few operations then discarded (disposable virtual machine), the data allocated will be substantially reduced.

Hyper-V also, since the 2008 R2 version released in 2009, supports the live migration of virtual machines [24]. It means that virtual machines can be migrated

between Hyper-V server maintaining network connections and uninterrupted services running in the virtual machine.

However, the Hyper-V 2008 R2 live migration have strict requirements:

- each node must have the Failover Clustering feature, which is a feature available only on the Enterprise or higher edition of Windows Server.
- each node must have a network adapter that carries Cluster Shared Volumes (CSV) communication. Client for Microsoft Networks and File and Printer Sharing for Microsoft Networks must be enabled in the network adapter properties to support SMB, which is a requirement for Cluster Shared Volumes.
- it is required that the storage configuration and hardware on the failover cluster be identical and that the cluster nodes used for live migration have processors by the same manufacturer

2.4.3 Windows Management Instrumentation (WMI)

Windows Management Instrumentation (WMI) is the infrastructure for management data and operations on Windows-based operating systems. It is possible to write WMI scripts, like in VBScript or Windows PowerShell, or applications to automate administrative tasks on remote computers, but WMI also supplies management data to other parts of the operating system and products.

One of the product that expose its management interface through WMI is Microsoft Hyper-V, which is also another reason for choosing it as the supported virtualizator in Octopus.

The most important features of WMI are:

- .NET management interface

  because the System.Management namespace relies on the existing COM/DCOM plumbing, the created WMI provider and its set of WMI classes becomes automatically available to all .NET applications independently of the language used.

- Remoting capabilities over DCOM and SOAP

  WMI, other than the DCOM transport, offers SOAP transport since Windows Server 2003 R2 through the WS-Management initiative led by Microsoft, Intel, Sun Microsystems and Dell. This initiative allows to run any scripts remotely or to consume WMI data through a specific set of interfaces handling SOAP requests/responses. The advantage for the WMI provider developer is that when he exposes all his features through WMI, Windows Remote Management/WS-Management can in turn consume that information as well.

- Support for queries in WQL

  WMI offers support for Windows Management Instrumentation Query Language (WQL), which is a SQL-like implementation of the CIM Query Language (CQL), a query language for the Common Information Model standard from the Distributed Management Task Force. It is a subset of the standard ANSI SQL with minor semantic changes.

  For instance, to query all system Drives on a computer that have less than 2Mb of free space:

```
SELECT * FROM Win32_LogicalDisk WHERE FreeSpace <
        2097152
```

- Eventing capabilities

  WMI offers the capability to notify a subscriber for any event it is interested in. WMI uses the WMI Query Language (WQL) to submit

WQL event queries and defines the type of events to be returned. The eventing mechanism, with all related callbacks, is part of the WMI COM/DCOM and automation interfaces.

Thanks to the .NET management interface, it is possible to call directly from the .NET code any WMI function. For instance, a C# code equivalent to the WQL query mentioned before "*SELECT * FROM Win32_LogicalDisk WHERE FreeSpace < 2097152*" (with a variable for specifying the free space) is:

```
IEnumerable<ManagementObject> GetLogicalDisks(ulong
        freeSpace)
{
    var mc = new ManagementClass("root\\CIMV2",
        "Win32_LogicalDisk", null);
    ManagementObjectCollection moc = mc.GetInstances();

    foreach (ManagementObject mo in moc)
    {
        if (mo["FreeSpace"] != null)
        {
            ulong f = (ulong)mo["FreeSpace"];
            if (f < freeSpace)
                yield return mo;
        }
    }
}
```

The information about the Win32_LogicalDisk class can be found at the Microsoft MSDN site:

http://msdn.microsoft.com/en-us/library/windows/desktop/aa394173.aspx .

It is also possible to execute direcly WQL queries through the .NET code: this is useful to retrieve only the instances that satisfies specified conditions instead of checking that condition on the received instances. Equivalent of the code as before but using WQL is:

```
IEnumerable<ManagementObject> GetLogicalDisks(ulong
        freeSpace)
{
    string query =
        "SELECT * FROM Win32_LogicalDisk WHERE
        FreeSpace < " + freeSpace;

    using (var s = new ManagementObjectSearcher(new
        ObjectQuery(query)))
    {
        var moc = s.Get();
        foreach (ManagementObject mo in moc)
            yield return mo;
    }
}
```

Another example involves the Hyper-V management interface, and it enumerates the existing virtual machine in the specified node (null for localhost):

```
IEnumerable<ManagementObject> GetVirtualMachines(string
        node)
{
    string scope;
    if (node == null)
        // localhost
        scope = "root\\virtualization";
    else
        scope =
        String.Format("\\\\{0}\\root\\virtualization",
        node);

    var mc = new ManagementClass(scope,
        "Msvm_ComputerSystem", null);
    ManagementObjectCollection moc = mc.GetInstances();

    foreach (ManagementObject mo in moc)
    {
        string systemType = (string)mo["Caption"];
        if (systemType == "Virtual Machine")
            yield return mo;
    }
}
```

The information about the class Msvm_ComputerSystem can be found at the Microsoft MSDN site:

http://msdn.microsoft.com/en-us/library/cc136822.aspx .

2.4.4 Microsoft .NET Framework and F# language

We decided to implement Octopus using the .NET Framework, and in particular using the F# language. We chose the .NET Framework because its ease

of interaction with the Windows Operating System and with Microsoft Hyper-V, both trough WMI.

The .NET Framework is a software framework developed by Microsoft that runs primarily on Microsoft Windows, but it exists a free and open source implementation called Mono capable of running all the mainly used systems (Linux, Android, BSD, iOS, OS X, Windows, Solaris, ...). It includes a large library and provides language interoperability across several programming languages. Programs written for the .NET Framework execute in a software environment (as contrasted to hardware environment), known as the Common Language Runtime (CLR), an application virtual machine that provides services such as security, memory management, and exception handling. The class library and the CLR together constitute the .NET Framework.

The language F# have become recently an official language of the .NET framework. It is a multi-paradigm programming language, which means that its syntax allows programmers to use it with different programming paradigm. In the case of F#, it supports functional programming, imperative and object-oriented programming. Its main programming paradigm is the functional programming.

It is born as a ML implementation in .NET; in fact, its syntax is very similar to the ML's. Like ML (and generally other functional languages), it is a strongly typed language that use intensive type inference; that means that data types do not need to be explicitly declared by the programmer, they will be deduced by the compiler during compilation. Also, like ML, it's a functional language with eager evaluation. Eager evaluation, in contrast with lazy evaluation (used by the functional language Haskell for example) forces the evaluation of sub expression immediately when evaluated instead of waiting for its utilization.

For functional programming, it also provides several constructs like tuples, records, lists, discriminated unions… and common functions typical of functional programming, like map, reduce, filter, etc.

Like other functional languages, iterations are usually expressed using recursion, recursive functions that invokes themselves. Usually, in imperative programming paradigm, recursion is avoided because it increases considerably the call-stack size. On the contrary, in functional languages, recursion is often used being careful to respect the tail-recursion requirements: if the recursive function do not execute any instruction after calling itself, the recursion can be expressed like a iterative while, avoiding building new stack entry for each call.

This is an example of the function “factorial”, which calculates the number factorial, in both non-tail-recursive and tail-recursive implementation:

Non tail-recursive

```
let rec factorial n =
    match n with
    | 0 -> 1
    | _ -> n * factorial (n - 1)
```

Tail recursive

```
let factorial n =
    let rec f n t =
        match n with
        | 0 -> 1 * t
        | _ -> f (n - 1) (n * t)
    f n 1

(* the recursive function is wrapped inside
a normal one because the recursive one
requires the "tail" parameter, which we want
to hide to the caller *)
```

An example of the F# code using WMI interface to query Hyper-V for the existing virtual machine, in other words the F# equivalent code for the C# provided before, is:

```
let getVirtualMachines node =
    let scope =
        match node with
        // localhost
        | None | Some(null) -> "root\\virtualization"
        | Some(n) -> sprintf
         "\\\\%s\\root\\virtualization" n

    let mc = new ManagementClass(scope,
         "Msvm_ComputerSystem", null)
    let moc = mc.GetInstances()

    [ for _mo in moc do
        let mo = _mo :?> ManagementObject
        let systemType = string (mo.["Caption"])
        if systemType = "Virtual Machine" then yield mo
         ]
```

that have type:

```
val getVirtualMachines : string option -> ManagementObject list
```

that means that it's a function which takes a string option and returns a ManagementObject list.

The option type in F# is used when an actual value might not exist for a named value or variable. An option has an underlying type and can hold a value of that type, or it might not have a value. The value "None" is used when an option does not have an actual value. Otherwise, the expression "Some(...)" gives the option a value.

3 Implementation

The implementation of this thesis was designed in two different projects: one for the implementation of a library that wraps all the WMI calls directed to Hyper-V, in order to allow an easily Hyper-V management via both scripts (i.e., using the PowerShell) and applications. This project is called **Hyper-F**.

Hyper-F is mainly represented by a single dll library which exports all the functions and data structs required to manage an Hyper-V server. However, because of some operation cannot be executed remotely, it was necessary to implement a Windows service that must be installed on every Hyper-V server to execute locally those particular operations.

The other software is **Octopus** itself, and it's composed by several projects:

- The Octopus Scheduler

  this is a Windows Service that runs in a centralized "headnode" and coordinates all the Hyper-V servers, scheduling their virtual machines. It also contains the CLIPS environment, implemented as an open source library called CLIPSNet and available at the page

  http://sourceforge.net/projects/clipsnet .

- The Octopus VM Deployer

  the deployer is a Windows Service or linux script that must be installed in a VM Template in order to detect a new deployment and execute the operation to specialize that instantiation.

- The Octopus Web Interface

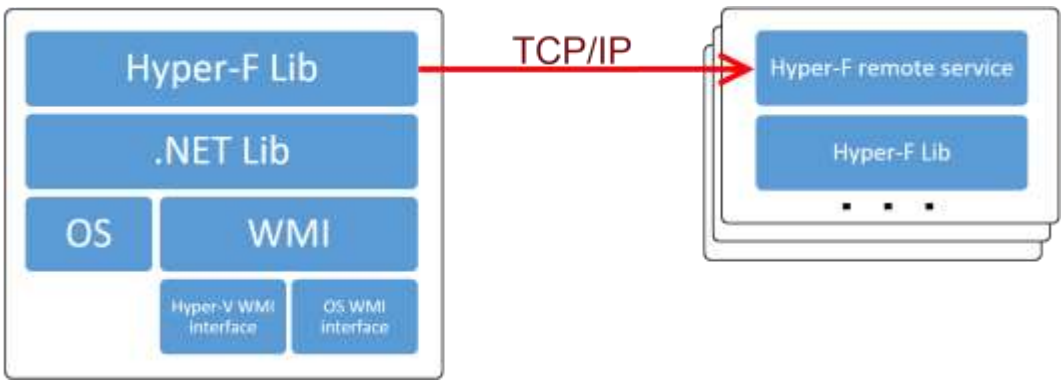
  this is the web interface used by administrators and users to create and manage virtual machines.

- other projects, like a setup project to automatize some operations during the software installation, a small GUI application to execute the configuration of a fresh installed Octopus service, and some libraries to share common code between projects.

3.1 Hyper-F

As mentioned before, the implementation of the functions that calls the WMI functions to manage Microsoft Hyper-V has been developed separately in a .NET library, defining a small API (application programming interface) to manage local and remote servers.

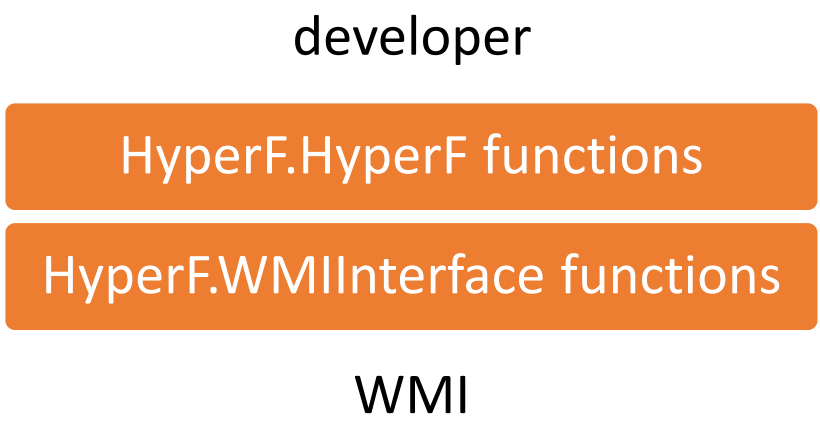
Other than a library, Hyper-F is also consisted by a Windows Service application, installed in every Hyper-V server we want to manage and it is used to execute locally all the function we cannot execute remotely, either because of Hyper-V limitation or because security issues.



Most of the WMI functions exposed by Microsoft Hyper-V are wrapped in a .NET function, which other than invoking the relative WMI function, performs security and stability checks, and hides the WMI generic ManagementObject type

to the developer using a safer wrapped object. In fact, all the WMI functions returns a single or a collection of ManagementObject as a result, which, depending on the function, they can represent Virtual Machines, Virtual Hard Disks, Virtual Ethernet cards, etc.

So the library is designed in 2 main layers:



The HyperF.WMIInterface layer is the only one that have direct access to the WMI interface for both OS and Hyper-V functions, and it exposes the functions that calls the WMI functions. They include only basic correctness check, in order to avoid that the WMI function call could result in a unpredictable result. An example of one of this function is the following, which allows to create a new virtual machine:

```
/// <summary>
/// Creates a new virtual machine with default settings:
/// One single-core processor, 512Mb of RAM, no Hard disk,
/// no NIC, no other stuff
///
/// #Example:
/// #Creates a new VM called " MYVM01" in the server NODE03
/// #  new_VM (Some "NODE03") "MYVM01" null
/// #Example:
/// #Same as above, but it stores the VM data in the specified path
/// #  new_VM (Some "NODE03") "MYVM01" @"\\NODE08\VirtualMachines"
/// </summary>
let new_VM server (newVmName:string) (path:string) =
  let getVirtualSystemGlobalSettingDataInstance
          (scope:ManagementScope) =
    let settingPath = new
          ManagementPath("Msvm_VirtualSystemGlobalsettingData")
    use globalSettingClass = new ManagementClass(scope, settingPath,
          null)
    use globalSettingData = globalSettingClass.CreateInstance()
    globalSettingData.["ElementName"] <- newVmName
    if path <> null then globalSettingData.["ExternalDataRoot"] <-
          path
    globalSettingData.GetText(TextFormat.WmiDtd20)

  let scope = new ManagementScope((getVirtualizationManagementPath
          server), null)
  use virtualSystemService = Utility.getServiceObject scope
          "Msvm_VirtualSystemManagementService"
  use inParams =
          virtualSystemService.GetMethodParameters("DefineVirtualSy
          stem")
  inParams.["ResourcesettingData"] <- null
  inParams.["Sourcesetting"] <- null
  inParams.["SystemsettingData"] <-
          getVirtualSystemGlobalSettingDataInstance scope

//http://msdn.microsoft.com/en-us/library/cc136786(VS.85).aspx
  use outParams =
          virtualSystemService.InvokeMethod("DefineVirtualSystem",
          inParams, null)

  let code = outParams.["ReturnValue"] :?> uint32
  if code = uint32(Utility.ReturnCode.Completed) then
    new ManagementObject(string outParams.["DefinedSystem"])
  else if code = uint32(Utility.ReturnCode.Started) then
    Utility.jobCompleted outParams scope
    new ManagementObject(string outParams.["DefinedSystem"])
  else
    failwithf "Define virtual system failed with error %d" code
```

This functions creates an empty virtual machine with default settings: 512Mb of RAM, 1 CPU, no hard disks and no Ethernet cards. The result of the function is a ManagementObject that represent the new Virtual Machine just created.

This function is not exposed outside the library; instead, it is wrapped on another higher-level function, defined inside the HyperF.HyperF layer, that hides the ManagementObject functions and returns a VirtualMachine object, which is an Hyper-F type:

```
module LocalOnly =

  (* ... *)

  let new_VM_To newVmName path =
    let vmobj = WMIInterface.new_VM None newVmName path
    new VirtualMachine(vmobj)
```

Note that this function doesn't have any parameter to specify the Hyper-V server.

In the higher-layer HyperF.HyperF, some functions are defined inside a submodule called LocalOnly. Those are the functions that it cannot be called remotely, or during tests we found that calling them remotely can cause runtime issues (generally security issues). To execute this kind of functions we use the Windows Service "Hyper-F remote executer". So, to create an empty virtual machine, developers will call this function:

```
let new_VM_To server newVmName path =
  match server with
  | None -> LocalOnly.new_VM_To newVmName path
  | Some(s) when isLocal s ->
              LocalOnly.new_VM_To newVmName path
  | Some(s) ->
              let ret =

          execute_remote_HyperF_function_withReturn s
                    [| "new_VM_To"; newVmName; path |]
              VirtualMachine.Deserialize(ret)
```

This function checks for the “server” parameter: if None, then it’s meant to be executed locally, if Some but the specified server is the local one, execute it locally, else call the remote Hyper-V service to invoke that function locally with the specified parameters. In this last case, the result is serialized in a string and deserialized to be retuned as object.

We have seen an example of the functions that allows to create a default empty virtual machine… of course there are other functions to fully customize the virtual machine:

```
let set_VMMemory (vm:VirtualMachine)
        (newMemoryAmountMb:int) =
  WMIInterface.set_VMMemory vm.ManagementObject_VM
        (int64 newMemoryAmountMb)

let set_VMCPUCount (vm:VirtualMachine) newCpuCount =
  WMIInterface.set_VMCPUCount vm.ManagementObject_VM
        newCpuCount

let add_VMNic (vm:VirtualMachine) (switch:NetworkSwitch
        option) isLegacy (macAddress:string option) =
  match switch with
  | None -> WMIInterface.add_VMNic
        vm.ManagementObject_VM None isLegacy macAddress
  | Some(s) -> WMIInterface.add_VMNic
        vm.ManagementObject_VM (Some
        s.ManagementObject) isLegacy macAddress



let add_VMHardDiskToController
        (controller:DiskController) address vhdPath =
  if address >= int(controller.Limit) then failwith
        "The specified Address is out of the controller
        limit"
  let drive =
    let d = try get_VMDrivesByController controller |>
        Array.find (fun x -> x.Address = address)
            with _ -> add_VMDiskDriveToController
                          controller address
    if (get_VMDiskByDrive d).IsNone then d
    else failwith "There is already a disk drive in the
        specified Address"
  add_VMHardDiskToDrive drive vhdPath
```

etc…

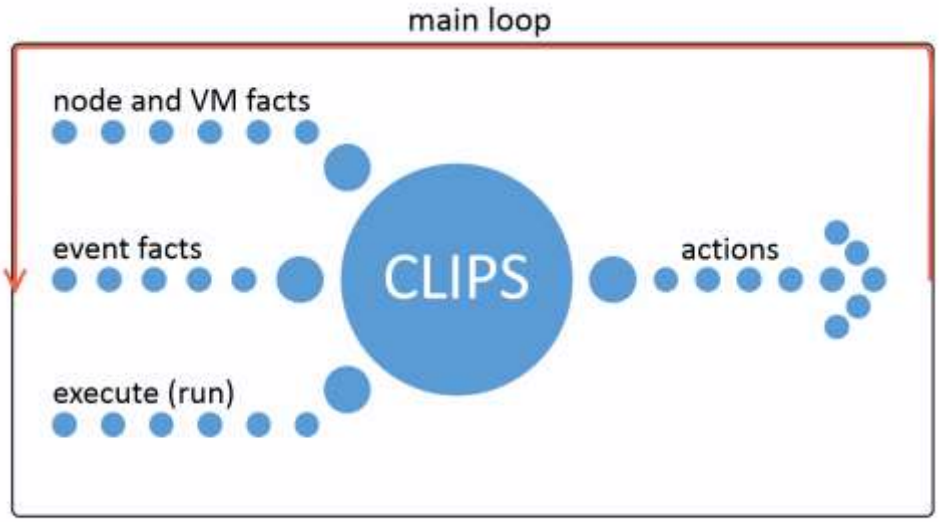
3.2 The CLIPS environment

The CLIPS environment is instantiated and controlled by the Octopus Scheduler module. This chapter maybe have should written inside the “Octopus Scheduler” one, but it’s so much important for the thesis that we preferred to dedicate it more visibility.

As already described before, the Virtual Machine Manager is controlled by the expert system CLIPS in order to make decisions autonomously, and to implement policies that limits users using and creating the virtual machines.

For every main loop iteration (the interval between each iteration is about 60 seconds), the Octopus Scheduler checks the status for every Hypervisor Hosts and for every Virtual Machines. Then, all the performance values collected are asserted as Facts inside the CLIPS environment. During these 60 seconds, several events could occur: an user asks for virtual machines, a virtual machine finishes its initialize configuration, a user stop one virtual machine, etc… Any of this event is asserted to the CLIPS environment in order to let a policy to decide what to react for that event.

Concluding the iteration, the Fact “triggeringRun” is asserted and then the *(run)* command is ran.



The Fact “triggeringRun” is used to have a fresh fact asserted just before the next run, in order to have Rules fired at every *(run)* or when the other conditions don’t changes.

An example of this is used by the initialization and finalization script, implemented with these two Rules:

```
(defrule initRun
  (declare (salience 1000))
  (triggeringRun)
  =>
  (bind ?nodes (host-list))
  (bind ?len (length$ ?nodes))
  (bind ?*hostsMemory* (create$))
  (loop-for-count (?i 1 ?len)
    (bind ?node (nth$ ?i ?nodes))
    (bind ?freeMem (getNodeValue host-memoryfree ?node))
    (bind ?*hostsMemory* (append$ ?*hostsMemory*
         (create$ ?node ?freeMem)))
  )
)
```

```
(defrule postRun
  (declare (salience -1000))
  ?f <- (triggeringRun)
  =>
  (retract ?f)
)
```

Before explaining what they do, the first thing you should note is the special declaration "(declare (salience <num>))". It is used in CLIPS to declare the priority for the Rules to be fired. Rules with the same priority (default is 0) will run in an unpredictable order, but thanks to the salience you can ensure some Rules will fire after or before others. Rules with higher salience will be fired first.

The first Rule, initRun, is used to initialize the multifield ?*hostsMemory* (this is a global variable[1]) which contains the main memory available for every hosts. This is keep in a structure because, during several virtual machine allocations, it must be changed in order to be up to date between an allocation and the next.

The last Rule, postRun, is only used to clear the Fact "triggeringRun".

When the CLIPS command *(run)* is executed, depending on the knowledge is asserted, some Rules that implements the Octopus policies may run.

We decided to implement a script to show the following policies and automations:

- Deny to any user to own more than <maxVmPerUser> virtual machines
- Defragment the hosts utilization migrating the virtual machines from one host to another one, with the purpose of using as few nodes as possible.
- Keep the hypervisor hosts turned off when unused

[1] The syntax to define a global variable is `(defglobal ?*name* = value)`

3.2.1 Policy to implement quotas

Let's see the script that implement the policy that denies users to own more than 5 virtual machines.

First of all, we need to concretize the <maxVmPerUser> parameter. We have defined it as a global variable in CLIPS, in order to be easily configurable without modifying anything else. So we defined:

```
; the maximum number of VM that a user can create
(defglobal ?*maxVmPerUser* = 10)  ; 10 virtual machines
```

Now, let's assume there is a user who want to instantiate a new virtual machine, using the template with ID = 1 which describes a virtual machine with 2 CPUs, 4Gb of ram, a 60Gb of disk and "Windows Server 2008" operating system.

After received the request, the scheduler will assert in the CLIPS environment a fact described in the following syntax:

```
(event-creating-vm
        <request id>
        <owner name>
        <number of vm>
        <number of CPU per vm>
        <ram size per vm>
)
```

so, in our example, it will assert the following fact:

```
(assert (event-creating-vm sampleId mura 1 2 4096))
```

Now, in this policy we have defined the following Rule that checks the number of Virtual machines already owned by that user and allows or denies it to create more:

```
(defrule createVmRule
  ?f <- (event-creating-vm ?reqId ?owner ?numVm ?numCpu
        ?ramSize)
  =>
  (bind ?alreadyCreated (countVmPerOwner ?owner))
  (if (> (+ ?alreadyCreated ?numVm) ?*maxVmPerUser*)
   then
    (deny-newvm ?reqId (str-cat "no more than "
        ?*maxVmPerUser* " VMs per user"))
   else
    (bind ?nodes (assignNodesForNewVm ?numVm ?numCpu
        ?ramSize))
    (if (< (length$ ?nodes) ?numVm)
     then
      (deny-newvm ?reqId "insufficient resources for
        this request")
     else
      (create-newvm ?reqId ?nodes) [2]
    )
  )
  (retract ?f)
)
```

this rule takes any “event-creating-vm” fact and assign the variables

- ?f, which described the fact itself
- ?reqId ?owner ?numVm ?numCpu ?ramSize, which contains all the event-creating-vm parameters

then, with no other conditions, execute the rule. In the actions of the rule:

- It will count the virtual machine already owned by the user who is requesting the creation, using the function *(countVmPerOwner)* defined in this script.

[2] actually the function here is not “create-newvm” but “internal-create-newvm”, needed to handle the event we are going to create a virtual machine in a turned off host. We are going to see this later, and the end of the 3.2.3 chapter.

- Compares the value (?alreadyCreated + ?numVm) to the value in the global variable ?*maxVmPerUser* :
    - if it's greater, deny the creation of the new Virtual Machine(s) calling the function *(deny-newvm)*, that notifies the scheduler that the request is denied
    - if it's not, then select a list of nodes, using the function *(assignNodesForNewVm)*, where the new virtual machines can be allocated, and notifies the scheduler, using the function *(create-newvm)*, to proceed creating the new virtual machines.
- Lastly, it undefines the initial fact "event-creating-vm" stored in ?f .

Note that this rule uses some user-defined functions, like *(countVmPerOwner)*, *(assignNodesForNewVm)*, *(create-vm)* and *(deny-vm)*.

The first one, find all facts named "vm-owner" which have as second parameter the value ?owner; then returns the number of facts found.

The second one iterates the available nodes looking for a way to fit all the virtual machines requested, depending on the hosts' available main memory. If it succeeds, it return the list of the nodes where the allocate the single Virtual Machines (a succeeded request for 3 VM always returns 3 nodes), if it fails return an empty list.

The *(create-vm)* and *(deny-vm)* are actually implemented in F#, inside the Octopus scheduler code. This is possible thanks to a feature of the CLIPSNet library we use, which allows to declare a .NET function and define a CLIPS function which invokes the .NET code. The code in F# to do that is the following:

```
type DelegateStrStr = delegate of CLIPSNet.DataType *
          CLIPSNet.DataType -> unit
type DelegateStrMulti = delegate of CLIPSNet.DataType *
          CLIPSNet.DataTypes.Multifield -> unit

module clipsFunctions =
  let clips_create_newVm (reqId:CLIPSNet.DataType)
          (nodes:CLIPSNet.DataTypes.Multifield) =
    let reqIdStr = string reqId
    let nodeArray = [| for x in nodes -> string x |]
    let requestGuid = new Guid(reqIdStr)
    FunctionsCore.allowRequestOfNewVm requestGuid nodeArray

  let clips_deny_newVm (reqId:CLIPSNet.DataType)
          (reason:CLIPSNet.DataTypes.String) =
    let reqIdStr = string reqId
    let requestGuid = new Guid(reqIdStr)
    FunctionsCore.denyRequestOfNewVm requestGuid


let mutable env : Environment = null

let initEnv () =
  if env <> null then env.Clear(); env.Dispose()
  env <- new Environment()
  new UserFunction(env, new
         DelegateStrMulti(clipsFunctions.clips_create_newVm),
               "create-newvm") |> ignore
  new UserFunction(env, new
         DelegateStrStr(clipsFunctions.clips_deny_newVm),
               "deny-newvm") |> ignore

  let clipsDefaultFile =
      try (getFile getServiceBinPath
          "clips-default.clp").FullName
      with _ ->
          failwith
            "Clips file \"clips-default.clp\" not found."
  env.Load(clipsDefaultFile)
  (* etc etc *)
```

The DelegateStrStr and DelegateStrMulti represent the types of the functions that we will export on the CLIPS environment: the first one is a function that takes two generic CLIPS DataType (that we use as string parameters) and returns void, the second one is a function that takes a generic CLIPS DataType

(again, used as a string), a CLIPS type Multifield (that is the clips representation of a record: in this instance, we use it as an array of strings) and returns void.

The module clipsFunctions contains all the functions that will be exported inside the CLIPS environment, just for arrange the code for better organization.

The function initEnv initializes the CLIPS environment: it creates a new environment with "new Environment()" and exports a .NET function using the constructor UserFunction specifying the environment to use, the function wrapped in its delegate type-representation and the CLIPS function name.

It also contains the code to load the CLIPS script "clips-default.clp" that contains the policy that must be preloaded on the CLIPS environment, like the *createVmRule* rule that we have seen.

After the initialization, the CLIPS environment will contain the *(create-vm)* and *(deny-vm)* as a functions with these syntax:

```
(deny-newvm ?reqId ?reason)
(create-newvm ?reqId ?nodeList)
```

and, every time one of those function will be called, actually a .NET code is executed and the .NET return value will be the return value in CLIPS.

3.2.2 Policy to defragment the servers' VM allocation

The idea of this policy is to compact the utilization of the Virtual Machine in order to use as few nodes as possible. So, if the entire load of a node can be moved to other nodes, we do it to have a node unused.

This policy fits well, and do the best, with the next policy: the one that shut down the unused nodes. Combined, Octopus will consume as little power as possible.

This is the Role that implement this policy:

```
(defrule defragmentNodesAllocation
 (declare (salience -99))
 (triggeringRun)
 =>
 (bind ?nodes (host-list))
 (loop-for-count (?i (* (length$ ?nodes) -1) -2)
   (bind ?ii (* ?i -1))
   (bind ?node (nth$ ?ii ?nodes))
   (loop-for-count (?j 1 (- ?ii 1))
     (bind ?vms (getAllVmOnNode (nth$ ?ii ?nodes)))
     (loop-for-count (?k 1 (length$ ?vms))
       (bind ?vmId (nth$ ?k ?vms))
       (if (< (getVmMemorySize ?vmId)
        (_getGlobalNodeMemFor (nth$ ?j ?nodes)))
        then
          (migrate-vm ?vmId (nth$ ?j ?nodes))
          (_setGlobalNodeMemTo (nth$ ?j ?nodes) (-
        (_getGlobalNodeMemFor (nth$ ?j ?nodes))
        (getVmMemorySize ?vmId)))
          (_commitGlobalNodes)
          (return)
       )
     )
   )
 )
)
```

It is declared with a low salience so other Rules, like “createVmRule”, can be fired first and “defragmentNodesAllocation” can work on an up to date environment.

The implementation is the following:

i. let be ?len = (length$?nodes), therefore the number of the hosts
ii. for each node ?n1 from the index ?len to 2
iii. -- for each node ?n2 from the index 1 to (?len – 1)
iv. -- -- for each VM ?v instantiated in the host ?n1
v. -- -- -- it checks if ?v can fit in ?n2, if so it migrates ?v from ?n1 to ?n2

In other words, this algorithm try to compact the hosts utilization moving the Virtual Machine from the hosts with higher index to the hosts with lower index.

3.2.3 Policy to shut down the unused hosts

This policy, combined with the previous one, can be called the “green policies”.

The idea is very simple: assuming that the cluster is under-loaded most of the time, and thanks to the previous policy that tries to keep the load compacted in few nodes, we shut down and keep turned off the empty hypervisor hosts and we turn them on only when needed.

The implementation is simple as its idea. The Rule that implements this policy is the following one:

```
(defrule checkNodesToShutdown
  (declare (salience -100))
  (triggeringRun)
  =>
  (bind ?nodes (host-list))
  (loop-for-count (?i 1 (length$ ?nodes))
    (bind ?node (nth$ ?i ?nodes))
    (if (getNodeValue host-isreachable ?node)
      then
        (bind ?numVm (length$ (getAllVmOnNode ?node)))
        (if (eq ?numVm 0) then (shutdown-node ?node)
         (break))
    )
  )
)
```

Note that the salience is a little bit lower that the "defragmentNodesAllocation", so it can turns off the nodes that the defragmentation has just cleared.

For each reachable Hypervisor Node, it just check the number of Virtual Machines allocated in that node. If zero, shut down the node.

We need to make a consideration now. Now that we can have powered off nodes as well as powered on nodes, during the creation of a Virtual Machine we need to take care of this to turn on a node we need to use for an allocation.

For this purpose, the function *(create-vm)* used in the Rule createVmRule must be replaced with the function *(internal-create-vm)* defined as follows:

```
(deffunction internal-create-newvm (?reqId ?nodes)
  (bind ?allNodesReachable TRUE)
  (loop-for-count (?i 1 (length$ ?nodes))
    (bind ?b (isNodeReachable (nth$ ?i ?nodes)))
    (bind ?allNodesReachable (and ?allNodesReachable
        ?b))
    (if (not ?b) then (poweron-node (nth$ ?i ?nodes)))
  )
  (if ?allNodesReachable
    then (create-newvm ?reqId ?nodes)
    else (assert (delay-create-vm ?reqId ?nodes))
  )
)
```

This function checks if all the nodes chosen for that particular request are turned off, inferred because reachable.

If all nodes are already turned on, just redirect the call to the *(create-vm)* we already know.

If some of the nodes are not reachable, it calls *(poweron-node)* on them to turn it on. Then, instead of calling *(create-vm)*, it asserts “delay-create-vm ?reqId ?nodes” to delay the creation of this request to the next *(run)*, hoping that the nodes we are trying to turn on will be turned on.

To handle this new Fact, there is a new Rule of course:

```
(defrule checkDelayedVmCreation
  (declare (salience 1))
  (triggeringRun)
  =>
  (bind ?delayedCreations (getFacts delay-create-vm))
  (loop-for-count (?i 1 (length$ ?delayedCreations))
    (bind ?ff (nth$ ?i ?delayedCreations))
    (bind ?f (fact-slot-value ?ff implied))
    (bind ?allNodes (create$))
    (bind ?allNodesReachable TRUE)
    (loop-for-count (?j 2 (length$ ?f))
      (bind ?b (isNodeReachable (nth$ ?j ?f)))
      (bind ?allNodesReachable (and ?allNodesReachable
        ?b))
      (bind ?allNodes (append$ ?allNodes (nth$ ?j ?f))
    )
    (if (not ?b)
      then
       (poweron-node (nth$ ?j ?f))
    )
  )
  (if ?allNodesReachable
    then
      (create-newvm (nth$ 1 ?f) ?allNodes) (retract
        ?ff))
  )
)
```

The salience for this Rule is greater than 0, the default salience. It means that this Rule will be fired before firing all the “createVmRule” activations.

It retrieve all the “delay-create-vm” Facts and, for each of them, checks the available status of their nodes. If are all reachable finally it calls the *(create-vm)* function, otherwise it tries again to power the unreachable nodes up, and leaves the Fact asserted. The next execution of *(run)* will fire this rule again, and so on until all nodes are powered on.

3.2.4 Considerations on policies

The policies illustrated here have the purpose to demonstrate the expressivity of the expert system CLIPS and the versatility of the Octopus scheduler. They are not optimized and, most important, they don't cover all the possible scenarios. This would be quite impossible, too.

But every system administrator who decide to configure Octopus for his cluster can easily adjust / rewrite the policy using all the events raised from Octopus to CLIPS and all the command exposed from Octopus in CLIPS.

This is a list of the events that Octopus raises:

```
(event-creating-vm <reqId> <user> <numVm> <numCpu>
        <ramSize>)
(event-created-vm <vmId>)
(event-deleted-vm <vmId>)
(event-suspended-vm <vmId>)
(event-paused-vm <vmId>)
(event-starting-vm <vmId>)
(event-started-vm <vmId>)
(event-migrated-vm <vmId> <hostSource> <hostDest>)
```

This is a list of command that Octopus exposes on CLIPS:

```
(deny-newvm ?reqId ?reason)
(create-newvm ?reqId ?nodeList)
(migrate-vm ?vmId ?nodeDest)
(suspend-vm ?vmId)
(shutdown-node ?vmId)
(poweron-node ?vmId)
```

3.3 The Octopus Scheduler

The octopus scheduler is a Windows Service application entirely written in F#. As all windows services, it runs and stay in background all the time the Windows operating system is up, even if there isn't any user logged in.

It is consisted to a main loop that periodically checks all the Hyper-V servers to control their availability and performance counters, and all the virtual machines instantiated in those servers. In details, it monitors the RAM memory utilization and the CPU load.

3.3.1 The creation of a Virtual Machine

We have just seen how the creation of a virtual machine is handled in CLIPS: the fact "event-creating-vm" is asserted and a set of rules and functions defined in CLIPS decide if the user is allowed to create the virtual machine(s) or not.

What happens next, if the CLIPS allows the user to create the virtual machine(s), is that CLIPS calls its function "create-newvm", which calls the F# function "clips_create_newVm", which calls the function allowRequestOfNewVm:

```
val allowRequestOfNewVm : System.Guid -> string array -> unit
```

This function takes 2 arguments: the GUID that represent the virtual machines creation request, and a string array that contains the nodes selected by CLIPS where the new virtual machines will be allocated. The GUID is assigned by Octopus when the user request a creation of virtual machine through the Octopus

Service, and the entire request is stored into the SQL Database until CLIPS processes that request.

When allowRequestOfNewVm is called, for each virtual machine requested, Octopus execute a routine of virtual machine creation that consists of this following steps:

1. check the correctness of parameters, configuration data and reliability of the assigned Hyper-V server. Eventually, handle faults.
   Let's name the shared storage path as "saveVmPath".
2. create a new empty virtual machine into "saveVmPath", with default RAM size and CPU count, using the Hyper-F function "HyperF.new_VM_To".
3. Hyper-V actually stores the virtual machine inside a subdirectory names "Virtual Machines", and then inside another subdirectory named as the GUID of the virtual machine itself. So the real path of the virtual machine is (saveVmPath + "\Virtual Machines\" + vm.Guid).
   Let's name this path as "realVmPath".
4. create a new differencing Virtual Hard Drive, deriving from the one represented by the selected Virtual Machine Template, into the path "realVmPath".
5. create a Virtual Floppy Disk, which contains the deployer configuration file (used by the Octopus Deployer to specialize the virtual machine instance), into the path "realVmPath".
6. assign to the virtual machine the RAM memory amount and the CPU count values, chosen by the user.
7. add to the virtual machine a legacy Network Adapter, with a MAC address chosen by Hyper-V.

8. connect the Virtual Hard Disk and the Virtual Floppy Disk to the virtual machine.
9. assign a GUID to identify the Virtual Machine, in order to hide to the user the GUID used by Hyper-V.

   Finally, add the virtual machine record to the SQL database, setting its state to “Configuring”.

Note in the point 7, we add a Legacy Network Adapter. Hyper-V have a flag “legacy” for network adapters that identify if the NIC driver will be a general one (legacy = true) or if it can use specialized driver that takes advantage of hardware support (legacy = false), which requires the Operating System to have that particular driver. Of course, the non-legacy NIC has better performance than the legacy one.

During the creation of a virtual machine, Octopus use a legacy NIC because it doesn’t know if the Virtual Machine template uses an Operating System that contains the specialized driver of not, and the legacy one works in all operating systems. So, it’s the Octopus Deployer, which knows that, to execute a command through the Octopus Service to eventually upgrade the NIC from legacy to non-legacy.

For the records, at the moment of this thesis is written, Hyper-V have the integration services drivers for:

- Windows (XP Service Pack 3, Vista SP2, 7+)
- Windows Server (2000 SP4, 2003 SP2, 2008+)
- Linux Red Hat Enterprise Linux 5.7, 5.8, 6.0-6.3 x86 and x64
- Linux CentOS 5.7, 5.8, 6.0-6.3 x86 and x64

After the execution of allowRequestOfNewVm, the Virtual Machines requested are allocated on the selected Hyper-V server and are ready to be configured.

The Octopus Scheduler have a task that periodically (every 30 seconds) checks the status of Virtual Machine configurations. It is used to avoid to execute more than 1 configuration at time for every Hyper-V server, and to starts the configuration process for new virtual machines. So, after a virtual machine creation, this task will run the configuration process.

The configuration process is actually performed by the Octopus Deployer, so the Octopus Scheduler task just starts the virtual machine in Hyper-V using the Hyper-F function “HyperF.start_VM”.

3.4 The scheduler’s database

The Octopus Scheduler uses several database tables to store persistent and volatile information:

- the **User** table, which is used to handle users authentication
- the **HyperVNodes** table, which is used to store information about the hypervisors nodes
- the **VMTemplates**, used to store the virtual machine templates information
- the **VirtualMachineRequested**, used to store the Virtual Machine creation requests by users, waiting to CLIPS to decide if the request can be allowed
- the **VirtualMachineData**, used to stored all the persistent information for a virtual machine, even after its deletion

- the **VirtualMachinePendingCreation**, used to store temporary the information to create a new virtual machine after a CLIPS approval
- the **VirtualMachineActive**, used to store all the Virtual Machine information about its current lifetime
- the **GeneralConfig** table, used to store various settings

The first tables we are going to see are related to authentication. We have a "Users" table where username, password, authentication token, and other user information are stored:

Column name	Data Type	Allow Nulls
ID	int	No
Username	nvarchar(30)	No
Firstname	nvarchar(20)	Yes
Lastname	nvarchar(20)	Yes
FullName	nvarchar(50)	Yes
Password	nvarchar(50)	No
AuthToken	uniqueidentifier	No
AuthTokenExpire	datetime	No

Table 1 - Database table Users

For security reason, we know we should store only the hash of the password, or the password encrypted, and not the password itself in clear. But for the purpose of this thesis, which is realizing a virtual machine scheduler with a scriptable policy, and also to make debugging more easy, we overlooked this security issue.

This is a very simple way for authenticate users; there are other well-known better system of authentication, i.e. which allows users to have more active sessions (with more authentication tokens), but again this was not in the purpose of this thesis, and can be discussed in future development of the software.

After a user logs in, Octopus creates a string (at now, a GUID) which represent its authentication token. It has an expiration of 8 hours, set on the row "AuthTokenExpire". The authentication token represent a valid identity to call any Octopus function.

When a user logs out, Octopus clear its authentication token and set its expiration to a date in the past.

Regarding the hypervisors node, we have the HyperVNodes table:

Column name	Data Type	Allow Nulls
ID	int	No
HostName	nvarchar(65)	No
IpAddress	nvarchar(15)	No
ExternalNIC	uniqueidentifier	No
IsReachable	bit	No
CpuCount	int	No
MemorySize	bigint	No
LastCpuLoad	real	Yes
LastMemoryFree	bigint	Yes
LastCounterUpdated	datetime	No

Table 2 - database table HyperVNodes

This table stores all the information used by Octopus about the hypervisors it manages. The HostName, IpAddress, ExternalNIC, and IsReachable are using to reach and manage the node in the network; the CpuCount and MemorySize should be static (unless the hardware changes) and represents the resource capacity of the node. LastCpuLoad and LastMemoryFree are the last values read from the performance counter the Octopus scheduler in its main loop. LastCounterUpdate of course represent the time that the counters are read the last time.

For managing the Virtual Machine Templates we use the VMTemplates table:

Column Name	Data Type	Allow Nulls
ID	int	No
IsDeleted	bit	No
Name	ncarchar(65)	No
Description	ntext	No
CpuCountMin	int	No
CpuCountMax	int	No
MemorySizeMbMin	int	Yes
MemorySizeMbMax	int	No
VhdPath	nvarchar(256)	No
ParentTemplateID	int	No

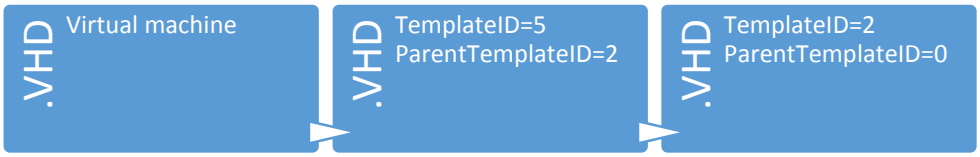
Table 3 - database table VMTemplate

This table contains both descriptive information like Name and Description, used to show to users what template they could need, and technical information like CpuCountMin and Max, which is the range of CPU cores supported / suggested

for this operating system or this specific installation, the same applies for the memory size with MemorySizeMbMin and Max, the VhdPath column to specify the physical path of the template file .vhd, and the ParentTemplateID.

This last column is usually 0, indicate that the template depends by nothing but its own vhd. But, in the case of a virtual machine is “templatized”, a new template will be created setting this field as the template id that virtual machine had.

Technically, the column ParentTemplateID indicates that this template has a differencing disk from the specified template. Consequently, a virtual machine instantiated with a template (for instance, with ID=5) that derives from a parent (for instance, with ID=2) will have this chained differencing disk:



Of course, once a VHD is created as “differencing disk” from another one, the parent vhd cannot be changed anymore. It becomes a read only disk. All it’s information are inherited in its children vhd, which contain only the differences from the parent.

About the Virtual Machine management, we have several tables.

One is dedicated to the user requests for new virtual machine creations, and it’s called VirtualMachineRequested:

Column Name	Data Type	Allow Nulls
ID	int	No
RequestID	uniqueidentifier	No
NumberOfVm	int	No
OwnerName	nvarchar(50)	No
MachineNames	nvarchar(64)	No
Password	nvarchar(128)	No
Description	ntext	Yes
CpuCount	int	No
MemorySize	bigint	No
VmTemplateID	int	No
Allowed	bit	Yes

Table 4 - database table VirtualMachineRequested

This table is used to handle users requests to new virtual machines. After a user asks for one o more new virtual machines, the request is stored in this table (which contains all the information to handle the request – again, the security about the password was not an issue during the development of this thesis –) and then the event “event-creating-vm” is asserted in CLIPS.

If the expert system CLIPS decides to allow this creation, the column Allowed changes from NULL to TRUE and will be created as much as many records in the table VirtualMachinePendingCreation depending on the number of virtual machine requested (column NumberOfVm).

Therefore, the VirtualMachinePendingCreation table is described as follows:

Column Name	Data Type	Allow Nulls
ID	int	No
VmDataID	int	No
HostServer	nvarchar(64)	No
Password	nvarchar(128)	No

Table 5 - database table VirtualMachinePendingCreation

The data contained in this table are very basic since the main information about virtual machines are kept in the table VirtualMachineData. The column VmDataID is the foreign key to the record to VirtualMachineData. We will describe this table shortly.

In the meantime, we describe the table VirtualMachineActive:

Column Name	Data Type	Allow Nulls
ID	int	No
VmDataID	int	No
OctopusGuid	uniqueidentifier	No
HyperVGuid	uniqueidentifier	No
CurrentServer	nvarchar(64)	No
StoredPath	nvarchar(256)	No
HyperVState	int	No
HyperVHealthState	int	No
HyperVHeartbeat	int	No
LastStateCheck	datetime	No
IsCheckpointed	bit	No
IpAddress	nvarchar(15)	No

Table 6 - database table VirtualMachineActive

This table contains all the information regarding the current virtual machine session. Some of these are meant to change often, like the several Hyper-V states and heartbeat, other ones instead describes this instance and doesn't change never, like the GUIDs and the StoredPath.

The CurrentServer field contains the current hypervisor node that is hosting the virtual machine. Every time the virtual machine is migrated, this field must be updated. This field is also important to know which is the right Hyper-V server that hosts the virtual machine, in order to provide Hyper-F the correct parameter to its management.

Note the presence of two GUIDs: the HyperVGuid and the OctopusGuid. The HyperVGuid is the GUID assigned by Hyper-V to represent univocally the virtual machine in the system. This GUID is necessary to Hyper-F to invoke correctly the WMI functions to manage the virtual machine. Nevertheless, this GUID is never shows outside Octopus and, even if users don't have to handle GUIDs manually, they (their clients) operates only with the OctopusGuid. The octopus scheduler always translate the HyperVGuid to OctopusGuid and viceversa to let "the outside" managing the virtual machines without knowing the real Hyper-V GUIDs.

Also, the Hyper-V GUID could virtually change depending on the kind of migration performed between systems, for this reason it's better to avoid users to work on the real GUID.

The last two tables we have seen have the field VmDataID, which is a foreign key for the field ID of the VirtualMachineData, described as follow:

Column Name	Data Type	Allow Nulls
ID	int	No
OctopusGuid	uniqueidentifier	No
OwnerName	nvarchar(50)	No
MachineName	nvarchar(64)	No
Description	ntext	No
CpuCount	int	No
MemorySize	bigint	No
OctopusState	int	No
MacAddress	nvarchar(12)	No
VmTemplateID	int	No

Table 7 - database table VirtualMachineData

All the information contained in this table are meant to live after the Virtual Machine deletion, as a history of the virtual machines created in the system. All the fields are pretty self-descriptive, or we already have seen them in other tables.

The OctopusState is the only one which deserves some additional words. It contains a state which represent what Octopus is doing with this virtual machine. It could be “Normal”, then the relevant state is the Hyper-V one, “Configuring” when the virtual machine is just created and during the specialization phase (see The Octopus Deployer), “Deleting” or “Deleted”, just before and after deleting the virtual machine, and some eventual errors like “CreationFailed”, “DeletionFailed”, “MigrationFailed”, “CheckpointingFailed”, and “GenericError”.

The last table we are going to describe is the GeneralConfig one:

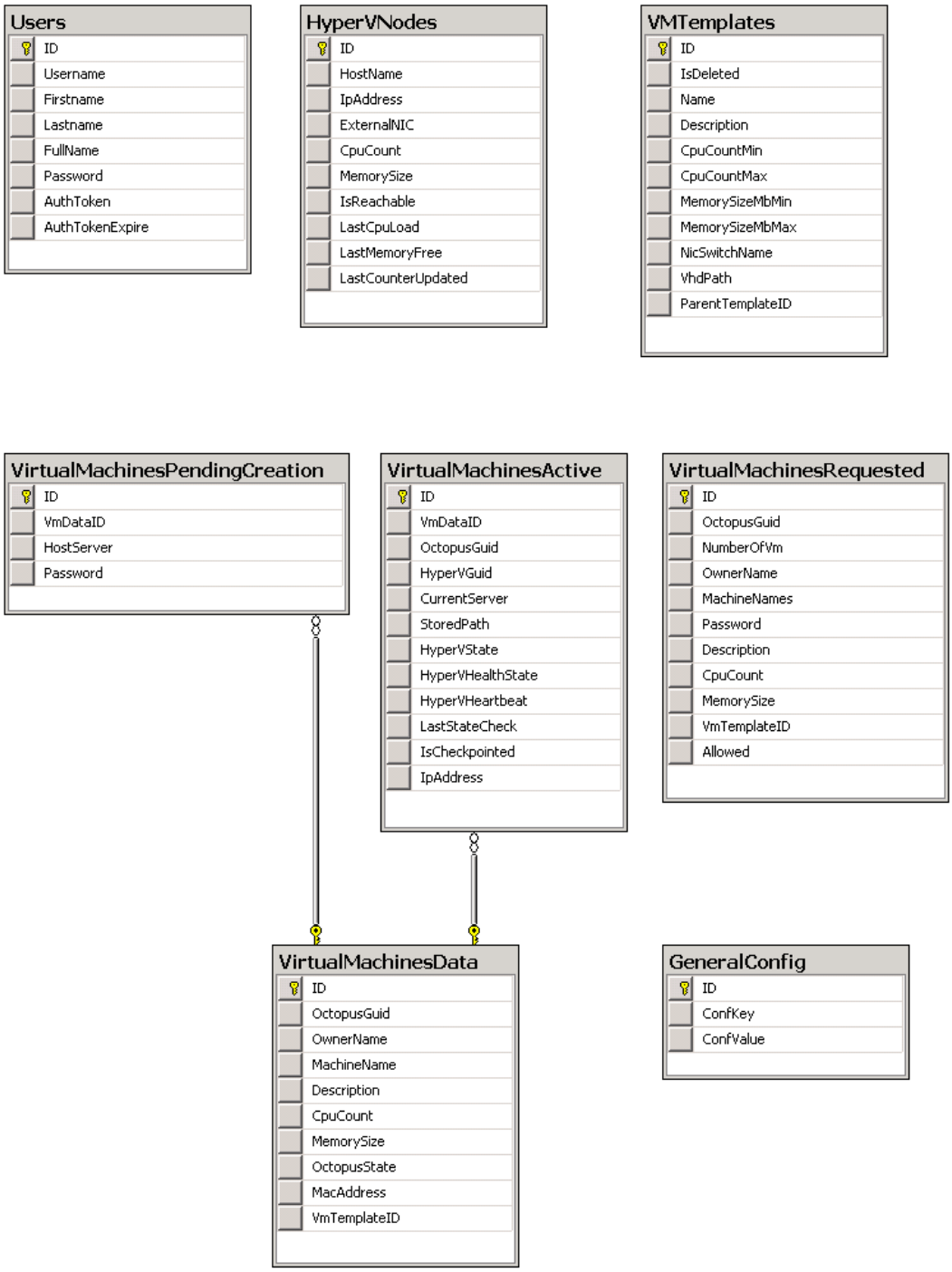
Column Name	Data Type	Allow Nulls
ID	int	No
ConfKey	nvarchar(50)	No
ConfValue	ntext	No

Table 8 - database table GeneralConfig

This table is used as a blob of data, or better as a Dictionary. It stores generic pairs key/value as text to ensure its persistence during the live of the system.

At this time, it is used for general configuration, like to set the Domain Administrator credentials and the storage path used to store the Virtual Hard Disks (VHD).

Concluding, the general schema for the database tables is the following one:


Users
ID
Username
Firstname
Lastname
FullName
Password
AuthToken
AuthTokenExpire
HyperVNodes
ID
HostName
IpAddress
ExternalNIC
CpuCount
MemorySize
IsReachable
LastCpuLoad
LastMemoryFree
LastCounterUpdated
VMTemplates
ID
IsDeleted
Name
Description
CpuCountMin
CpuCountMax
MemorySizeMbMin
MemorySizeMbMax
NicSwitchName
VhdPath
ParentTemplateID
VirtualMachinesPendingCreation
ID
VmDataID
HostServer
Password
VirtualMachinesActive
ID
VmDataID
OctopusGuid
HyperVGuid
CurrentServer
StoredPath
HyperVState
HyperVHealthState
HyperVHeartbeat
LastStateCheck
IsCheckpointed
IpAddress
VirtualMachinesRequested
ID
OctopusGuid
NumberOfVm
OwnerName
MachineNames
Password
Description
CpuCount
MemorySize
VmTemplateID
Allowed
VirtualMachinesData
ID
OctopusGuid
OwnerName
MachineName
Description
CpuCount
MemorySize
OctopusState
MacAddress
VmTemplateID
GeneralConfig
ID
ConfKey
ConfValue

3.5 The Octopus Deployer

The Octopus Deployer service is a start-up service, implemented for both Windows and linux, which is designed to run at the operating system startup to check if the local virtual machine needs to be configured (specialized), and eventually do that.

This service is used mainly to change the computer hostname and the Administrator/root password, with the choices made by the virtual machine's owner.

This service can be also used to perform specific operations in specific operating system instances; for example, in windows where the Hyper-V Integration Services are installed, it notifies the Octopus Service that the Virtual Machine's Network Card can be upgraded from the legacy one to the non-legacy, in order to increase its performance.

The implementation we provided for Windows and Linux are just examples. We implemented the Deployer service for Linux as a script in Python language, and for Windows as a Windows Service, but the implementation can be performed in different ways. In fact, we could change the Windows implementation with a javascript script, or a PowerShell script, in order to let administrators to change and personalize it to perform additional operations like joining a domain, applying security policies, etc.

3.5.1 Octopus deployer for Windows

The Windows version is implemented as a windows service. At the windows startup, it checks for the floppy disk if present, and looks for a file called “octopusDeployer.xml”. If the floppy disk is not present, or the file is not present on the floppy, the service stops itself.

If that file is found, it copies it into its local directory (to avoid using the floppy again) and starts the configuration process.

The virtual machine configuration, in windows, is performed with the following tasks:

1. searches for the configuration file “octopusDeployer.xml” in a floppy disk. If not present, it halts.
2. reads the configuration file (see Appendix A)
3. changes the Administrator password
4. changes the computer hostname
5. invokes the Octopus Service function to upgrade the NIC, from the legacy one to non-legacy.

   Note that the Deployer is designed to run only in the Windows version which supports the Hyper-V integration services, and they needs to be installed in the virtual machine template.
6. shuts down the local computer

   This is because the Octopus Service waits for the VM to be turned off before upgrading the NIC. In Hyper-V you can add or remove Network Cards from virtual machines only if they are turned off. Anyway, the Octopus Service has a timeout of 60 seconds, then forces the turn off of the virtual machine which has requested the NIC upgrade.

7. Now the virtual machine is started again, and the Deployer service starts at the windows startup. It remembers it's in a configuration process, so it resumes the previously configuration.
   The configuration is actually done, so now it invokes the Octopus Service function to notify the end of the virtual machine configuration.

With the last step, the Octopus Service knows that the configuration is done, so it changes the virtual machine state from "Configuring" to "Normal" and the virtual machine is ready to be used by the owner user.

The tasks 1, 2 and 5, as usual, are performed through the Windows Management Instrumentation interface. In detail, they calls the following WMI functions:

- *MIIS_CSObject.SetPassword* in namespace *root\MicrosoftIdentityIntegrationServer*
- *Win32_ComputerSystem.Rename* in namespace *root\cimv2*
- *Win32_OperatingSystem.Win32Shutdown* in namespace *root\cimv2*

3.5.2 Octopus Deployer for Linux

The linux version of the Octopus Deployer is implemented as script, written in Python language. It is executed every time the machine starts to perform the following operations:

1. It checks for the configuration file in a floppy disk. If present, jump to the step 3, else jump to the step 2.
2. It changes the machine name, then it halts.
3. reads the configuration file (see Appendix A)
4. changes the Administrator password
5. changes the computer hostname

The linux version is very simpler than the windows one. It doesn't need to reboot after changing the machine name (windows needs it) and, since there is no universal support for Linux for the Hyper-V Integration Services, it doesn't upgrade the Network Interface which requires the Virtual Machine to be shut down.

The script is written to be as far as possible portable between Linux distributions, so instead of using specific routines to change permanently the hostname, we chose to change it at every startup using the standard shell command "hostname" (steps 2 and 5).

Same point for changing the root password. Since it doesn't exist a portable command to do so unattended (without user interaction), we had to script a routing that uses the standard shell command "passwd" and interact with the command. The routine is described as follows:

```
def set_root_password(passwd):
    debugPrint("Setting password")
    child = spawn('passwd root')
    debugPrint("waiting for prompt 1")
    child.expect('.*[Nn]ew .*[Pp]assword')
    child.sendline(passwd)
    debugPrint("waiting for prompt 2")
    try:
        child.expect(['.*[Nn]ew .*[Pp]assword',
         '[Rr]etype', '[Rr]e-enter'], timeout=10)
    except Exception, e:
        child.send (chr(3)) # Ctrl-C
        child.sendline('') # tell passwd to quit
        raise Exception('New root password was not
         accepted')
    child.sendline(passwd)
    debugPrint("waiting for EOF")
    child.expect(EOF)
    debugPrint("closing...")
    child.close()
```

In a Ubuntu distribution, we put the script in a dedicated location /etc/octopus and we added to the file /etc/rc.local the following line:

```
/etc/octopus/startup-config.py
```

to have the script executed at every computer start-up, after all the other system scripts.

Again, this file is distribution dependent. In other distributions, for example in SuSe, there isn't any rc.local file. This can be addressed creating a symlink into /etc/init.d/rc5.d which points to a small script /etc/init.d/octopusDeployer which simply contains following lines:

```
#!/bin/sh
/etc/octopus/startup-config.py
```

4 Conclusions

This thesis describes the implementation of a Virtual Machine Manager which several features:

- managing virtual machines in a cluster of hypervisor servers
- allowing users to manage their virtual machines independently, without interacting with the system administrator
- allowing users to instantiate virtual machine from a pre-configured template, which allows a very quick instantiation of the virtual machine (it's ready to be used in few minutes)
- scripting in the system exprert CLIPS environment the scheduling policy and the quotas Octopus must apply to the system, in order to cover a broad spectrum of scenario.

Initially, the thesis' goal was only to create a virtual machine manager with scheduling features, which aggregates several hypervirsor servers to create a large unique resource to allocate virtual machines; with a graphical interface (or web interface) dedicated to users, and not to administrators, which shows only the virtual machines owned by the user who is using it, and allows him to fully manage his virtual machines and to create new ones from preconfigured templates.

The policies were initially implemented static, with some parameters to set simple quotas (max virtual machines per user, max ram used per user, etc) and an algorithm that migrates virtual machines between hosts to defragment its allocation and save energy turning off the empty hosts.

At later stage in the study we decided to investigate how to improve the policy manager to be more dynamic. We first decided to develop the policy

manager as an internal script, with a simple language defined by ourselves. But we realized that we were trying to design something that already exists in many form in bibliography. We then decided to try the CLIPS language with a very efficient open source implementation.

We added to the Octopus Schedule the CLIPS environment and we adapted the layers to interact with the CLIPS environment. In other words, action that Octopus do (by itself or by users request) has become actions invoked by CLIPS (for instance, creating a virtual machine, migrating them, deleting them, etc) and the environment where Octopus looks to take decisions (hosts performance counters, virtual machines, virtual machines performance counters) has become the environment (or, in CLIPS language, the facts) of CLIPS.

During all the development time, the Octopus Virtual Machine Manager received positive feedbacks. Letting the users creating and managing their own virtual machines, with the ability to create them in just few minutes, and the script language which allows administrator to implement quotas and policies are the most appreciated features of Octopus, which indicates that these features are a real need for the today's virtualization software.

Octopus was first presented at the Cloud Futures 2010 [25], a workshop organized by Microsoft in Redmond, WA, USA. The public was very interested in the concept of "template-based virtual machine creation" because it allows to have a ready-to-use virtual machine in a couple of minutes in a distributed way where IT managers are not a bottleneck for the resource allocation process. Also, the Octopus policies allow to manage the virtualization servers that keeps unused nodes turned off received a very good response; green computing is always welcome nowadays.

Octopus was then presented at the International Supercomputing Conference (IIS) 2010 in Hamburg, Germany. We tuned Octopus for the occasion to deploy a complete High Performance Computing cluster composed by only virtual machines, with a template for the head node and a template for the computing node. We showed how Octopus can be used to deploy a virtualized HPC cluster in less than 10 minutes, have it ready for the computation, and calculate an intensive calculation with Ansys Fluent, a flow modelling simulation software. It received a very good reception by the public, and many of them declared that they would like to have a virtual machine manager like Octopus is their organization. Of course, the green computing advantages was another vantage point.

Regarding High Performance Computing computation, we found that in some times using virtual machines can speed up the computation in a cluster of many-cores nodes.

The hardware we have experimented was a single Gateway prototype of 4 CPU 8-cores (for a total of 32 cores) and 32Gb of RAM, configured with Windows Server 2008 R2 and Microsoft Hyper-V.

We tried executing a complex simulation in Ansys Fluent 12.1 with 16 processes with and without shared memory, first as processes in the host operating system (without virtualization), then as processes sparse in a virtual cluster composed by 16 nodes (virtual machines with 1 core and 1,5Gb of RAM) and a head node (vm with 2 cores and 2Gb of RAM) with Microsoft HPC v3. Tried executing in both single and double precision.

All the executions, single precision in shared memory, single precision in non-shared memory, double precision in shared memory, double precision in non-shared memory, was always faster when executed in 16 virtual machines then executed as 16 local processes.

We also tried the non-shared memory executions with Microsoft MPI (MPMPI, based on MPICH2), and MPI for HP (HPMPI).

Virtual speedup (hw/virtual)

	HPMPI shared	HPMPI	MSMPI	Average
single precision	115.17%	109.16%	118.33%	114.22%
double precision	131.82%	137.42%	139.45%	136.23%

In all the cases, the virtualized cluster has improved the calculation time with more speedup in the double precision numbers. The result was hard to explain, but we deducted that because some software (like Ansys Fluent) are not able to scale linearly increasing arbitrarily the number of core, using single core virtual machines we are able to exploit better the powerful of every cores obtain a better scalability.

This result is amazing, and it deserves more investigation in exploiting the hardware to improve calculation time for all computations. This is definitely an interesting topic to investigate about, as a future work.

More recently, Octopus was presented at the GARR workshop called “Calcolo e storage distribuito” in 2012 in Rome, Italy. In this workshop, we focused the attention to the using of an expert system (in this case CLIPS) to implement a policy manager for a virtual machine scheduler. Attendees was very interested in the idea because of the flexibility a scripted expert system can give for scheduling large amounts of virtual machines.

4.1 Future works

Now that we have developed the software, it's ready to be deployed in a real environment to investigate better what the CLIPS script need to be improved. During the development of Octopus, we just assumed generic requests of system administrators using both our experience and external feedbacks. But only a real deployment can allows us to collect a large amount of feedbacks, diversified by different scenario, that allows us to improve the policy manager adding or improving the Actions set and to enrich the CLIPS Facts with more usable information.

Also, as future works, there are some development improvement that it's possible to apply to Octopus that we preferred to not implement in this first release.

First of all, since every currently-used hypervisors support an SDK or provide and API to interact with them, it would make sense to implement similar libraries like Hyper-F but for the other hypervisors, and make them as plug-in modules. Then, implementing a new layer under the Octopus scheduler which generalizes the communication with hypervisors, it would be possible to plug new modules to communicates with new hypervisors, and make Octopus a scheduler of heterogeneous hypervisors servers.

Of course, in this scenario, migration between nodes that run different software virtualization could be impossible; even the simplest migration, which is migrating a turned off virtual machine. A turned off virtual machine is basically stored only in its virtual hard disk; the virtual machine hardware specification is easy to recreate without any noticeable change. In this case, it's sufficient to

migrate the virtual hard disk between nodes, or moving nothing in case of a shared storage, to destroy the virtual machine metadata from the source server and recreate the same virtual machine metadata in the destination server, then attach to it the old virtual hard disk. But it's just enough that two different software virtualization doesn't support the same virtual hard disk format to make the migration between them impossible.

Otherwise, using different virtualization software could allow to deploy a specific configuration of a virtual machine in a better hypervisor. Deploying a Windows virtual machine is surely better with Microsoft Hyper-V, while it doesn't virtualize perfectly a Linux one (primarily for the lack of the Hyper-V integration services). And a Mac OS X virtual machine can be deployed (just for license issues) only on Apple-based hardware. Octopus could organize the virtualization servers into groups, based on the virtualization software, and schedule virtual machine into them separately.

Another optimization that Octopus needs is about the Hypervisor Nodes and Virtual Machines monitoring, that is the routine that monitors the performance counter (memory, cpu, disk) of nodes and virtual machines, and checks the virtual machines' status (running, stopped, paused…).

The monitoring at now is performed actively by Octopus, which in the main loop of the Octopus Scheduler it checks node by node, and virtual machine by virtual machines, all those values. This approach is not scalable, and with a large number of nodes which can host a very large number of virtual machine, it could become a serious bottle neck.

A new approach could be implementing the monitoring routines decentralized, in services installed in the Hypervisors Nodes which read and pushes on the Octopus Services all the values read in their local machines. In this

way, Octopus doesn't have to poll the servers continuously and hang up in connection timeout if the server becomes unavailable.

During the thesis development, we have already developed the idea to snapshot and templatize virtual machines, but we didn't provide an implementation to focus only on the expert system policy manager.

The idea of snapshotting (or checkpointing) is already a consolidated concept in the virtual machine world [26] [27] [28]. It consists of take a snapshot (like a quick image) of the virtual machine in a particular moment, saving the current disk state and the memory state (including all hardware registers) to have the possibility to resume that exact virtual machine state in the future. This could be achieved using the hypervisor functionality, which in the current implementation is the Snapshot function of Hyper-V, or manually saving the virtual machines, copying the saved state in a safe location, differencing the virtual hard disk to establish a check-point, then restarting the virtual machine. We didn't investigate the advantages and disadvantages of the two approaches yet.

The idea of templatization is to make a current virtual machine, in a turned off state, a virtual machine template. In this way, users will be able to create templates for other users using the virtual machines they have instantiated and re-configured. It will be possible to have hierarchies of templates: a clean template, which is inherited from specialized templates with different software, which are inherited from further-specialized templates with different configurations.

At the end, as future development, we must consider improving the security implementation, a better authentication method and the creation of user's roles. At this moment it exists only two types of users, simple users and administrators. Adding role will make possible to assign different privileges to different users (also

easier to script in the policy manager) like, for instance, allowing virtual machine templatizing only to certain users.

5 Bibliography

[1] “Virtual machine,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Virtual_machine.

[2] “Hardware virtualization,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Hardware_virtualization.

[3] “Application virtualization,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Application_virtualization.

[4] “Java virtual machine,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Java_virtual_machine.

[5] Lindholm, Yellin, Bracha and Buckley, “The Java™ Virtual Machine Specification,” Oracle, 2012. [Online]. Available: http://docs.oracle.com/javase/specs/jvms/se7/html/index.html.

[6] “Common Language Runtime,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Common_Language_Runtime.

[7] E. Meijer and J. Gough, “Technical Overview of the Common Language Runtime,” 2001.

[8] “Intel® Virtualization Technology,” Intel, 2006. [Online]. Available: http://www.intel.com/technology/itj/2006/v10i3/1-hardware/6-vt-x-vt-i-solutions.htm.

[9] “AMD Virtualization,” AMD, [Online]. Available: http://www.amd.com/virtualization.

[10] “Migration with vMotion,” VMware, [Online]. Available: http://pubs.vmware.com/vsphere-

51/index.jsp?topic=%2Fcom.vmware.vsphere.vcenterhost.doc%2FGUID-3EE13ED8-172F-4560-B806-1E342AD7C486.html.

[11] “Virtual Machine Live Migration Overview,” Microsoft, 2012. [Online]. Available: http://technet.microsoft.com/en-us/library/hh831435.aspx.

[12] “Storage XenMotion,” XenSource, Inc., [Online]. Available: http://wiki.xen.org/wiki/Storage_XenMotion.

[13] “Virtual machines Teleporting,” Oracle VirtualBox, [Online]. Available: http://www.virtualbox.org/manual/ch07.html#teleporting.

[14] U.S. Environmental Protection Agency, Report to Congress on Server and Data Center Energy Efficiency Public Law 109-431, 2007.

[15] J. G. P. Koomey, "Growth in data center electricity use 2005 to 2010," 2011.

[16] “Energy Star,” U.S. Environmental Protection Agency, 1992. [Online]. Available: http://www.energystar.gov/.

[17] J. Harris, Green Computing and Green IT - Best Practices, 2008.

[18] A. Berl, E. Gelenbe, M. D. Girolamo, G. Giuliani, H. D. Meer, M. Q. Dang and K. Pentikousis, “Energy-Efficient Cloud Computing,” *Oxford Journals, The Computer Journal,* 2009.

[19] R. Harmon, “Sustainable IT services: Assessing the impact of green computing practices,” in *Management of Engineering & Technology, 2009*, Portland, 2009.

[20] Google, “Method and system for calculating risk in association with a security audit of a computer network”. Patent US 2002/0147803 A1, 2002.

[21] S. Northcutt, L. Zeltser, S. Winters, K. Kent and R. Ritchey, Inside network perimeter security, TechRepublic, 2005.

[22] H. Wu, “Network security for virtual machine in cloud computing,” in *Computer Sciences and Convergence Information Technology (ICCIT)*, Seoul, 2010.

[23] “WMI .NET Overview,” Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/library/ms257340(v=vs.80).aspx.

[24] “Hyper-V: Using Live Migration with Cluster Shared Volumes in Windows Server 2008 R2,” Microsoft, [Online]. Available: http://technet.microsoft.com/en-us/library/dd446679(v=ws.10).aspx.

[25] “Octopus presentation at Clous Futures 2010,” Microsoft Research, 2010. [Online]. Available: http://research.microsoft.com/apps/video/default.aspx?id=122454.

[26] Microsoft, “About Virtual Machine Checkpoints,” 2008. [Online]. Available: http://technet.microsoft.com/en-us/library/bb740891.aspx.

[27] Oracle, “Snapshots in VirtualBox,” [Online]. Available: http://www.virtualbox.org/manual/ch01.html#snapshots.

[28] VMware, Inc., “Understanding virtual machine snapshots in VMware ESXi and ESX,” [Online]. Available: http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1015180.

[29] “Adding Workstation Nodes in Windows HPC Server 2008 R2 Step-by-Step Guide,” Microsoft, 2011. [Online]. Available: http://technet.microsoft.com/en-us/library/ff601849.aspx.

Appendix A:
deployment configuration file sample

```
<?xml version="1.0" encoding="utf-8"?>
<VMConf>
  <GeneralConf>
    <VmGuid>
      68cc5f67-2c39-4d50-a960-d52736c85d38
    </VmGuid>
    <MachineName>TEST3</MachineName>
    <AdminPassword>PasswordTest</AdminPassword>
    <OwnerName>mura</OwnerName>
    <OctopusServiceAddresses>
      <Address>octopussched</Address>
      <Address>fe80::88ee:ed33:c1ba:7d04%17</Address>
      <Address>fe80::9d9:218c:e26e:7037%10</Address>
      <Address>2002:8372:2a1::8372:2a1</Address>
      <Address>192.168.1.1</Address>
      <Address>131.114.2.161</Address>
    </OctopusServiceAddresses>
  </GeneralConf>
</VMConf>
```