University “Politehnica” of Bucharest

Automatic Control and Computers Faculty,
Computer Science and Engineering Department

BACHELOR THESIS

vmgen - Automatic Virtual Machines Generation

Scientific Adviser:
As. Drd. Ing. Răzvan Deaconescu

Author:
Mircea Urse

Bucharest, 2011

Abstract

Virtualization is widely used today, and sometimes the process of creating and configuring virtual machines is time consuming. Sometimes, the need to configure multiple similar machines occurs, and it would be better to avoid repeating the same operations manually. The generation process consists of relatively simple operations, which can be automated. **vmgen** attempts to achieve this goal. It is a tool that allows the user to request a virtual machine with a specified, configured, operating system installed on it and with a list of applications to be installed. Then the virtual machine is generated and configured, and an archive containing the virtual machine is provided to the user. The user must provide the configuration file, written from scratch, or altered through the vmgCtl utility. In the future there might be a Web interface to ease the generation of the configuration file.

Contents

List of Figures

Notations and Abbreviations

API – Application Programming Interface
CLI – Command-Line Interface
CPU – Central Processing Unit
DHCP – Dynamic Host Configuration Protocol
DNS – Domain Name System
GUI – Graphical User Interface
HDD – Hard Disk Drive
IDE – Integrated Drive Electronics
IP – Internet Protocol
KVM – Kernel-based Virtual Machine
LXC – Linux Containers
MAC – Media Access Control
MBR – Master Boot Record
OpenVZ – Open VirtualiZation
OS – Operating System
PCI – Peripheral Component Interconnect
SCSI – Small Computer System Interface
SSH – Secure Shell
VM – Virtual Machine

Chapter 1

Introduction

1.1 Virtualization Technologies

1.1.1 History

The virtualization is not a new technology. It was developed in 1960's by IBM, for the mainframes. The problem with the mainframes back then was that they were very expensive, and they were not used at their full capacity. The virtualization was created to allow the partitioning of a large mainframe in multiple virtual machines, in order to use the resources in a more efficient way. When the x86 architecture came out, and servers on x86 started to be used, virtualization didn't seem necessary, because the computers were cheap, and for a while (about 10 years, between 1980's - 1990's) it was abandoned. Soon, the need for virtualization began to rise once again, because the computers became more and more powerful and they could do more than running a single OS. Unlike the mainframes, which had hardware support for virtualization, the x86 architecture didn't. VMware developed in 1999 the first product which was able to run multiple OS on the same x86 hardware. The problem with the x86 virtualization, was that there were a number of 17 CPU instructions that couldn't be virtualized (the OS would crash or malfunction). VMware solved the problem by trapping these instructions when they are generated, and replaces them with virtualization-safe instructions. Later, more applications that allowed virtualization of the x86 hardware, from other companies, appeared: **VirtualPC**, by Connectix (later acquired by Microsoft), **Xen** (an open-source hypervisor), **VirtualBox** by innoTek (later acquired by Sun Microsystems, and now by Oracle), **lxc** and **OpenVZ** (open-source OS-level virtualization solutions). For more details on the virtualization technology history, [6], [9], [7] and [10] can be consulted.

1.1.2 Virtualization Types

Virtual machines are used for a wide range of purposes today. Depending on what the virtual machine is needed for, a user may choose between various solutions.

OS Level Virtualization

OS level virtualization solutions allow for a more advanced chroot jail. The virtual machines are called containers, virtual environments, or virtual private servers. All the containers share the kernel of the host machine, so the host must be running a Linux version, with support for the used technology. The containers have the file system and processes isolated from the other containers. The processes from the containers are scheduled by the host's scheduler. The overhead of virtualization is minimum because the containers don't run a full system inside them. This solution is used, for example, to run multiple servers, isolated from each other, or to offer users separate running environments, with minimum overhead. However, a problem exists with the kernel being shared by all the containers: if one of the container runs some bad kernel code and produce a kernel bug, all of the other containers and the host machine are affected. Also, a different OS from the host's OS cannot be run in a container.

Of the applications that offer OS level virtualization, I can mention **OpenVZ** and **lxc**, which are currently supported in **vmgen** .

Full System Virtualization

The full system virtualization solutions allow running an entire OS (no need to modify the OS to be able to run in a guest machine) inside the virtual machine. The OS does not need to be the same as the host's (running, for example, a Windows VM on a Linux system, or vice versa). Each VM uses virtual hardware, for which there are special drivers. The overhead of running this type of VM is higher than the overhead for an OS-level container. The scheduling for the processes inside each VM is done by the VM's OS. The host's OS only schedules the virtualization application's process. This solution is used to test a new OS, without installing directly on the hardware, to run multiple OS at the same time. It is also appropriate for kernel development and drivers programming: when a programming error is done, the kernel bug affects only the VM, which can be then restarted, or, better, reverted to a saved snapshot.

Examples of full system virtualization solutions are **VMware Workstation**, **VirtualBox**, **VirtualPC**, **KVM** etc. Only **VMware Workstation** is currently supported in **vmgen** .

Hypervisors

Both the previous virtualization solutions type, need an underlying OS running on the physical machine. A hypervisor is a minimal piece of software that run directly on the hardware, and offers the possibility to create and run multiple virtual machines (guests) on top of it. The guests don't see the hardware directly, they see a virtualization of it. The advantage is that multiple OS can be run concurrently, without the overhead of an intermediate layer. However, the performances are not the same as if the OS would run directly on the hardware: the virtualized hardware needs special drivers, which cannot make full use of the devices features. The OS can be an unmodified version, or a modified version which is aware of the hypervisor used, and generates instructions to be run on it (the hypervisor is transparent to the applications, but not to the OS).

Some hypervisors are **Xen**, **VMware ESX**, **Hyper-V Server** etc. **vmgen** doesn't currently support any hypervisor.

1.2 Motivation

I will describe the motivation behind the implementation of **vmgen** , and which are its goals.

As described above, the virtualization solutions are widely used. Creating a virtual machine is a time consuming process, although it consists of relatively simple operations, that can be automated. For one machine, it might not be a big problem, but when a user wants to generate a couple of machines, some of them having similar properties, the process becomes repetitive and it takes more time. **vmgen** aims to automate the process of generating and configuring a VM. The user only needs to make the request for the desired configuration(s), and he will receive them when they are created, without any interaction during the process.

When a user wants to create a network topology using some virtual machines (for academic or other uses), he may need to install several machines with the same configuration, but the network card configurations and maybe some services. It would be easier to only change a few lines in a configuration file and request a new machine instead of installing the machines from scratch.

Another use for **vmgen** would be for non-technical users or who don't know how to configure a machine and the services on it, but they need the machine, in order to interact with its services (e.g. a Web server). The user is able to just specify the services to be installed, and it will be given the fully configured machine.

In the following chapters, I will present how the application is implemented and how the VM generation process works. More implementation details can be obtained from the project's website (wiki and repository), at [4].

Chapter 2

Application Architecture

2.1 Application Deployment And User Interaction

vmgen is designed to run as a service on a configured remote server. After it is started, it begins to listen for user requests. At the moment, it cannot process multiple requests simultaneously, they are processed sequentially. The reason why it has to run on a configured server is that, in the process of generating the requested machines, it needs to use some preinstalled virtual machines, and some installation kits for the applications to be installed. The preinstalled virtual machines cannot be accessed concurrently, and there needs to be implemented a synchronization mechanism to enforce mutual exclusion. At the moment, a request gains exclusive access during the whole process. In the future, it should be changed so that a requests needs to gain exclusive access only on the critical sections of the process.

The application receives a configuration file as input and generates an archive containing the generated virtual machine after it has finished the generating process. At the moment, the user can write himself the whole configuration file, or he can use our **vmgCtl** utility to alter an existing configuration file; an example of usage is shown in Listing 2.1. There is no Web interface for now, but it will be implemented in the future, as it will be presented in Section 6.1. No matter what frontend is added, it generates a configuration file based on the user input, and passes the generated file to the main application, which is not aware of the frontend.

```
1 ./vmgCtl.py config-debian.conf hardware.num_cpu=2
```

Listing 2.1: vmgCtl usage example

2.1.1 The Programming Language

The application code is written in **Python**. The reason we chose Python was because it is very easy to write code in it, and the code is more readable than C code, for example. Python has a large collection of features that are already implemented that we could use (command execution, file

manipulation, archive file manipulation, list comprehension etc.). We could focus on implementing the application features rather than re-implementing the small pieces of code for different operations. Also, the written code is cross-platform, so it could be run either on a Linux machine or on a Windows one. The application doesn't need critical performance to run, so the slower execution of Python code is not a problem.

2.1.2 Configuration File

The configuration file is a plain text file, with the extension **.conf**, in a modified **INI**[1] format. Besides the sections, the configuration file has also nested subsection, defined by using multiple square braces, according to the nesting level. An alternative format would be **XML**[2], but the file would get even larger than it is and would not be very easily readable and editable by the user. A detailed description of each section and examples follow. The currently supported options are available on the project wiki[3].

The hardware section

In the **hardware** section, the user describes the hardware characteristics of the machine. The available options are different between the used virtualization solutions, but most of them are common among all the solutions. The user can specify the ID of the virtual machine (name or number), the operating system, the number of processors, the available physical memory, the number of hard disks and their parameters, the partitioning details for each of them, the CD drives, the network adapters.

```
1  [hardware]
2          vm_id = TestMachine
3          os = winxppro
4          num_cpu = 2
5          ram = 512
6          [[hdds]]
7                  [[[hdd0]]]
8                  size = 8GB
9                  type = ide
10                 scsi_index = 0
11                 pos = 0:0
12                 name = hdd.vmdk
13                 [[[[partitions]]]]
14                         [[[[[partition0]]]]]
15                         type = primary
16                         fs = ntfs
```

[1] http://en.wikipedia.org/wiki/INI_file
[2] http://en.wikipedia.org/wiki/Xml
[3] http://ixlabs.cs.pub.ro/redmine/projects/vmgen/wiki/SupportedOptions

```
17                          size = 5000
18                          ...
19                  ...
20          [[cd_drives]]
21                  [[[cd_drive0]]]
22                  pos = 1:0
23                  path = /path/to/iso
24                  connected = 1
25                  ...
26          [[eths]]
27                  [[[eth0]]]
28                          type = nat
29                          connected = 1
30                          hw_addr = 01:00:de:ad:be:ef
31                  ...
```

Listing 2.2: sample hardware section

A very important field is **os**. It specifies the OS which will run in the virtual machine, along with its architecture (ubuntu-64, fedora, winxxppro, windows7-64 etc). The identifiers are the ones used in VMware machine description files. All the OS dependent selections (the OS installation, **Config** and **Installer** modules instantiation, system disk cloning) will be made based on this field. The application has dictionaries indexed by the OS identifier, whose keys are the various alternatives. For more details about the subject, see Section 4.2.

The config section

The **config** section is used to specify some system parameters, like the password for the root user (or Administrator, on Windows), the hostname, additional repositories (for **apt** or **yum**) etc. These are generic settings that do not fall in the other categories (network, users, services, applications).

```
1 [config]
2         root_passwd = pass
3         hostname = trudy
4         bash_completion = 1
5         [[repos]]
6                 repo0 = deb http://http.us.debian.org/debian stable
                      main contrib non-free
7         ...
```

Listing 2.3: sample config section (Linux guest)

The users section

The **config** contains the information about the groups and the users in the new system. All the groups from the **groups** subsection are created. The users from the **users** subsection are created with the specified name, password and home directory, and they are added to the specified groups afterwards.

```
1  [users]
2          [[groups]]
3                  group0 = julius
4                  group1 = vmg
5                  ...
6          [[users]]
7                  [[[user0]]]
8                          name = caesar
9                          passwd = alesia
10                         groups = julius
11                         home_dir = C:\Users\caesar
12                 [[[user1]]]
13                         name = vv
14                         passwd = pass
15                         groups = vmg
16                         home_dir = C:\Users2\vv
17                 ...
```

Listing 2.4: sample users section (Windows guest)

The network section

The **network** section is a larger one, and it groups all the configurations available for networks: network applications, network adapter configurations, firewall rules.

The network adapter configurations are a little different between Linux and Windows guests. There are some common properties, like the type of addresses (static or dynamic), the IP address, the network mask. These are specified for each adapter. The difference is at the DNS address and the gateway. On Linux, we have global DNS servers (in **`/etc/resolv.conf`**) and a single default gateway, with multiple additional routes. On the other hand, in Windows, each network adapter has its own gateway and DNS addresses. Therefore, for Linux guests, there are global properties for the DNS and gateway, and for Windows, the properties are for each adapter.

The firewall can also be configured. The user can specify a list of ports to be opened, along with the protocol and a short description for each port. If the user wants to specify more complex firewall rules, he can specify in the **firewall-rules** subsection the exact commands (**`iptables`** on Linux or **`netsh firewall`** on Windows) to be run on the machine. It is not a very good practice to run the exact commands provided by the user, because he can try to run some malicious code, but

these commands are executed only inside the virtual machine, and only this will be affected, not the physical machine running the service.

```
[network]
        [[eths]]
                [[[eth0]]]
                        type = static
                        address = 192.168.1.2
                        network = 255.255.255.0
                        gateway = 192.168.1.1
                        dns = 192.168.1.254
                [[[eth1]]]
                        type = dhcp
                ...
        [[open_ports]]
                [[[port0]]]
                        proto = tcp
                        port = 22
                        description = ssh
                [[[port1]]]
                        proto = all
                        port = 65000
                        description = myport
                ...
```

Listing 2.5: sample network section (Windows guest)

The devel section

The **devel** section specifies the development tools to be installed: development environments, compilers, profilers, debuggers, libraries etc.

```
[devel]
        vim = 1
        emacs = 1
        eclipse = 1
        build-utils = 1
        kernel-devel = 1
        valgrind = 1
        python = 1
        php = 1
        tcl = 1
        ...
```

Listing 2.6: sample devel section (Linux guest)

The services section

In the **services** section are specified the services to be installed on the machine. The main options here are servers, like Web, mail, DNS, DHCP, FTP etc. There aren't many parameters to configure for them at the moment, but it might be a good feature to work on in the future. One of the reasons why this feature is not implemented yet is that there has to be a way to specify these options in the config file and we didn't find a suitable way to do that.

```
1 [services]
2         httpd = 1
3         dns-server = 1
4         dhcp-server = 1
5         ftp-server = 1
6         sshd = 1
7         svn = 1
8         git = 1
9         ...
```

Listing 2.7: sample services section

The gui section

The **gui** section lists the applications to be installed, which need a graphical environment to run (non-CLI).

```
1 [gui]
2         mozilla-firefox = 1
3         google-chrome = 1
4         mozilla-thunderbird = 1
5         wireshark = 1
6         ...
```

Listing 2.8: sample gui section

2.2 Modules Description

vmgen consists of several modules. Figure 2.1 gives an overview over the general architecture of the application. The modules and the relationship between them will be described in detail below. The application is executed in command line, and needs to receive as arguments the virtualization solution (vmware, lxc, openvz so far) and the configuration file. A parser module reads the file and passes the retrieved data to the main generation process. The generation process consists of various stages. The main stages are **hardware generation**, **system configuration** and **application**

installation. The operations executed in these stages can be applied to more than one configuration. For example, in the **hardware generation** stage, the operations are almost OS independent, and are different only across the virtualization solutions. In the **system configuration** stage, the operations are distribution and virtualization solution independent, and are common for each OS family (Linux and Windows). The **application installation** stage is common for each installer type (**apt**, **yum**, **source installation** for Linux, or the installation kits on Windows). Also, a very important component, used in all the other one is the one responsible for the communication with the virtual machine. The operations needed to communicate with a given machine are dependent only on the machine type (virtualization solution used).

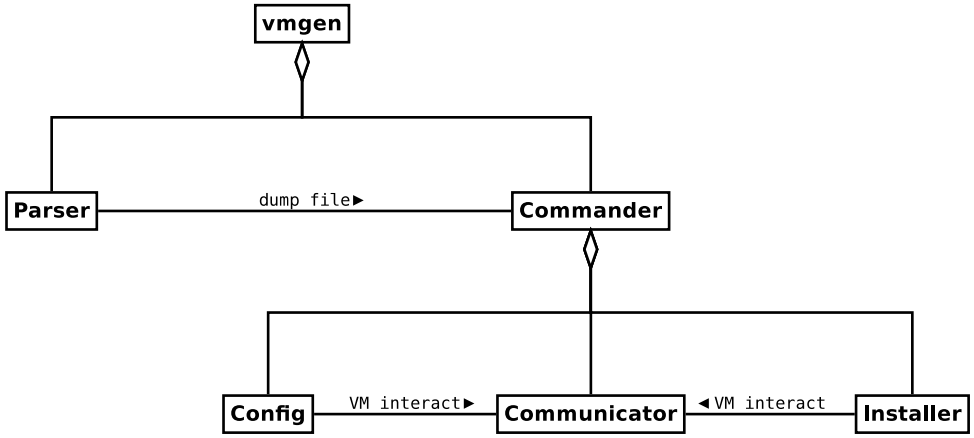

Figure 2.1: **vmgen** architecture

Given these observations, we tried to design the application to be modular. One of our goals was to not have duplicated code, so we grouped the operations that could be used in more than one place in a separate module and use that module instead. This provides easier code maintenance. The other goal was to design an extensible application. If later on someone wants to add support for a new feature (a new virtualization solution, new applications to be installed etc.), it is sufficient to create a new corresponding module, link it in the main application, and use the already implemented modules where needed.

In Figure 2.2 is presented the virtual machine generation process, from input to output. I will give a short overview over the functionality of the modules. Each module will be detailed in the next subsections. The user provides the configuration file for the machine he wishes to get. The **Parser** loads the file into memory and passes it to a **Commander**. The **Commander** generates the hardware of the machine and configures the OS, then it instantiates a **Config** and a **Installer** module. The **Config** makes the necessary system settings (users, network etc.). The **Installer** installs the requested applications and services. Then, an archive containing the virtual machine is created and made available to the user to download it.

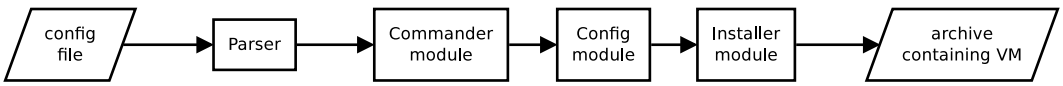

Figure 2.2: VM generation process

2.2.1 Configuration File Parser

The parser is a simple module, which uses the **ConfigObj**[1] module, from the Python library to parse the configuration file and store it in a **vmgStruct** structure. After the structure is created and populated into memory, it is serialized in a dump file, using the **pickle**[2] module (also from the Python library), in orderd to be de-serialized by the next component in the chain. The serialization offers a decoupling between the parser and the next stages of the application.

2.2.2 Commander Modules

The main component of **vmgen** is the **Commander**. The commander is instantiated at the beginning of the generation process and it basically controls the other modules. It receives the previously created dump file, and recreates the **vmgStruct** in memory. A commander uses the information from the **hardware** section in the config file (Subsection 2.1.2).

Because the steps needed to configure the virtual machine depend on what virtualization solution is used, there must be a commander for each supported virtualization solution. To keep the structure modular, an abstract class, **CommanderBase**, is used. This base class splits the generation process into smaller steps, and executes them in the correct order. For each step, there is an abstract method, which must be implemented by each of the concrete commanders. The abstract methods can be seen in Listing 2.9.

```
1  class CommanderBase:
2          ...
3          def startVM(self):
4          def shutdownVM(self):
5          def connectToVM(self):
6          def disconnectFromVM(self):
7          def setupHardware(self):
8          def setupPartitions(self):
9          def setupOperatingSystem(self):
10         def setupServices(self):
11         def setupDeveloperTools(self):
12         def setupGuiTools(self):
13         ...
```

Listing 2.9: CommanderBase methods

Only the **setupVM** method is called on a specific Commander instance. This method will, in turn, call the needed operations. The sequence of operations needed to configure a new virtual machine can be seen in Listing 2.10. The commander creates the hardware, partitions the disks, and setups the OS on the new machine, then it starts it and connects to it. The system configuration is made by instantiating the corresponding **Config** module (see Subsection 2.2.3) for the specified OS and

[1] http://wiki.python.org/moin/ConfigObj
[2] http://docs.python.org/library/pickle.html

calling its main method. The final step before turning the machine off is installing the applications, by category: services, developer tools, graphical applications. The installations are done by instantiating the corresponding **Installer** module (see Subsection 2.2.4). Each of these operations is implemented differently by the commanders.

```
1  def setupVM(self):
2          self.setupHardware()
3          self.setupPartitions()
4          self.setupOperatingSystem()
5
6          self.startVM()
7          self.connectToVM()
8
9          self.config = self.getConfigInstance()
10         self.config.setupConfig()
11         self.root_passwd = self.config.getNewRootPasswd()
12
13         self.installer = self.getInstallerInstance()
14         self.setupServices()
15         self.setupDeveloperTools()
16         self.setupGuiTools()
17
18         self.disconnectFromVM()
19
20         self.shutdownVM()
```

Listing 2.10: Commander sequence of steps

For each virtualization solution, a new commander is created by deriving the **CommanderBase** class and implementing the corresponding operations: **CommanderLxc**, **CommanderOpenvz**, **CommanderVmware** etc. An advantage of this organization is that a user who wants to add support for a different virtualization solution needs to define a new derived **Commander**, along with a corresponding **Communicator** (see Subsection 2.2.5) and add the particular implementations for its operations. The only modification needed to be made to the main application is at the top level, where the commanders are instantiated, to add the instantiation code for the new commanded.

The relationship between the classes can be seen in Figure 2.3.

For now, only 3 commanders are implemented, some of which will be presented in detail later: CommanderLxc (Chapter 3), CommanderVmware (Chapter 4) and CommanderOpenvz. In the future, more Commanders can easily be added to the application (more details can be found in Section 6.2).

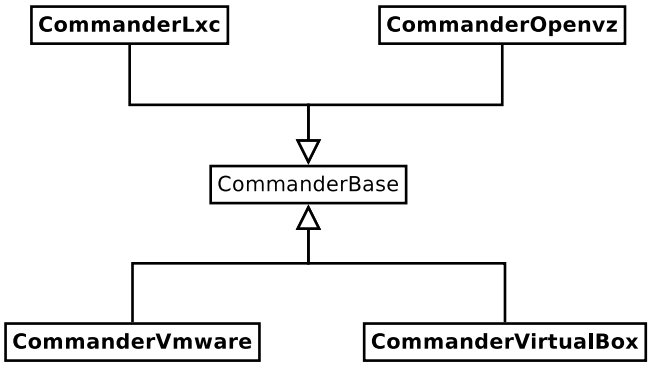

Figure 2.3: Commander modules

2.2.3 Configuration Modules

A **Config** module is responsible for configuring the OS on the virtual machine. The options used by the **Config** modules are placed in the **config** section in the config file (Subsection 2.1.2).

There is an abstract class, **ConfigBase**, which has abstract methods for each configuration group: **system configurations**, **users and groups**, **network**, **firewall** and a method for applying the settings generated by the previous ones. These methods are shown in Listing 2.11.

```
1 class ConfigBase:
2         def setupSystem(self):
3         def setupGroups(self):
4         def setupUsers(self):
5         def setupNetwork(self):
6         def setupFirewall(self):
7         def applySettings(self):
8         ...
```

Listing 2.11: ConfigBase methods

There is also a method, **setupConfig**, which calls each abstract method, to execute each configuration step. The sequence of steps is shown in Listing 2.12. The **getRootPasswd** method is used to return the stored password set for the privileged user during the configuration process, for later use.

```
1 def setupConfig(self):
2         self.setupSystem()
3         self.setupGroups()
4         self.setupUsers()
5         self.setupNetwork()
6         self.setupFirewall()
7
8         self.applySettings()
```

Listing 2.12: Config sequence of steps

A new **Config** module is created for each OS family: **ConfigLinux** and **ConfigWindows** by deriving **ConfigBase**. Each concrete **Config** class must implement the configuration abstract methods, according to the OS family it implements. The relationship between the classes can be seen in Figure 2.4.

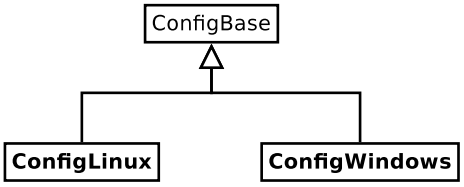

Figure 2.4: Config modules

A **Config** module is instantiated by the **Commander**, based on the operating system it read from the configuration file (in the hardware section). The **Commander** has a dictionary, whose keys are the possible values of the OS, and its values are the various concrete **Config** classes. After the **Commander** module instantiates the corresponding **Config** module, it needs to call only the **setupConfig** method to make all the necessary configurations. It then needs to retrieve the new password for the privileged user by calling the **getRootPasswd** method.

2.2.4 Installer Modules

The **Installer** module is used for installing applications inside the generated virtual machine. The main application does not need to know how the programs are installed, it needs to only send the install command for a specific program. The used tools and the needed operations are encapsulated inside an **Installer** module. The installing process of an application is not OS dependent, but rather depends on the installation tools provided by the OS. Some of these tools are **apt** for Debian based systems, **yum** for RedHat based systems, the installation from **sources**, for all the Linux distributions, individual **executable** kits, for Windows etc.

The **Installer** module is used to install the applications specified in various sections of the config file, like **devel** (Subsection 2.1.2), **services** (Subsection 2.1.2) and **gui** (Subsection 2.1.2).

The only method needed for an **Installer** module is one that receives a list of program names and installs them all. This method is provided by the abstract class **InstallerBase**. The definition of the class is shown in Listing 2.13.

```
1 class InstallerBase:
2         def install(self, programList):
```

Listing 2.13: Installer methods

For each installation tool supported, a derived class from **InstallerBase** is created, like **InstallerApt**, **InstallerYum**, **InstallerWindows**, **InstallerSource**. So far, only the first 3 installer types are supported. The relationship between the classes can be seen in Figure 2.5.

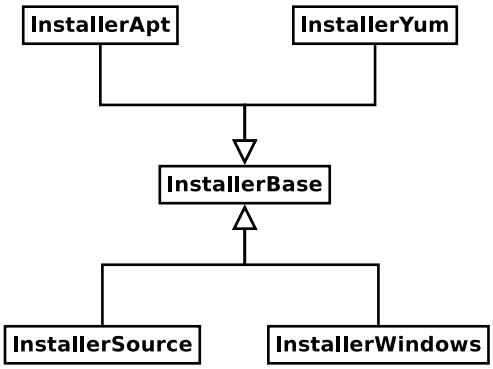

Figure 2.5: Installer modules

Internally an **Installer** module stores a dictionary, for associating a generic name for an application (OS and tool independent) with the real name and the parameters needed to install the application, using the module's specific installation tool. To add new applications, it is sufficient to add the corresponding entries in the dictionary, and they will be installable after that, without modifying anything else. The applications are installed with the default parameters, in the default path (usually in **C:\Program Files**). We could not found a suitable way to specify additional installation options for programs (like destination path, shortcuts created etc.), without making the configuration file too complex. A solution would be to specify the desired parameters for a program installation in a separate file, and in the main configuration file to provide the link to that file.

2.2.5 Communicator Modules

One of the most important modules is the **Communicator** module. It allows the rest of the components to communicate with the virtual machine (Figure 2.6). The rest of the modules don't need to know which machine are they sending commands to. They have a reference to a **Communicator** object, corresponding to the virtual machine, and they use the interface of the communicator to interact, indirectly, with the machine. Without the use of **Communicator** modules, the **Installer** and **Config** modules, which are independent of the virtualization solution used would have to decide by themselves which machine they are communicating with, and this code would be duplicated across their various implementations.

A communicator provides methods to run commands in the machine, to copy files from the local (physical machine) to the guest (VM), to remove files inside the guest. An abstract class, **CommunicatorBase** provides this interface. Its definition is presented in Listing 2.14.

```
1 class CommunicatorBase:
2         def runCommand(self, cmd):
3         def copyFileToVM(self, localPath, remotePath):
4         def deleteFileInGuest(self, remotePath):
```

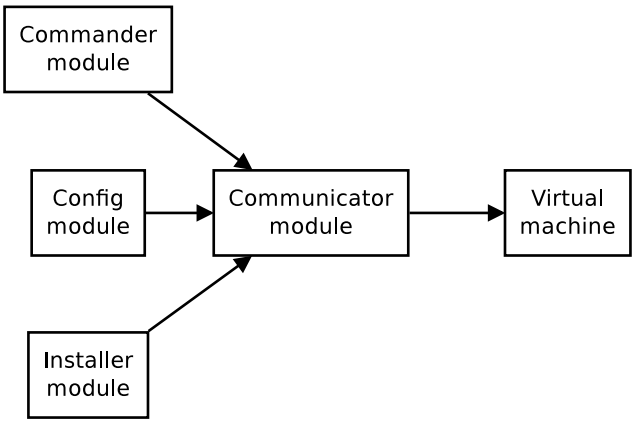

Figure 2.6: Communication with the VM

Listing 2.14: Communicator methods

A **Communicator** for each virtualization solution is created, by deriving **CommunicatorBase**. So far, we have created the derived classes for the supported virtualization solutions: **CommunicatorLxc**, **CommunicatorOpenvz**, **CommunicatorVmware**. The relationship between the classes can be seen in Figure 2.7.

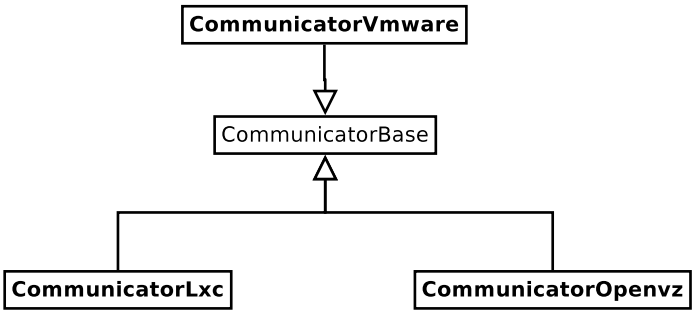

Figure 2.7: Communicator modules

The **CommunicatorVmware** module uses VMware's **vmrun** utility (at [8] can be found the official manual) to directly run applications, copy files inside the virtual machine, and delete files from it.

The **CommunicatorLxc** and **CommunicatorOpenvz** do not interact directly with the **lxc** and **OpenVz** containers. As will be presented in Chapter 3, the containers are not generated on the physical machine, but in a VMware virtual machine. So, these 2 communicators connect to the VMware machine through SSH, using public key authentication, and the execute the container specific commands to run programs inside the container. To copy files to the container, it is sufficient to simply copy through SSH (using **scp**) the files into the directory where the file system of the container resides. To delete files from the container, the steps are similar to the ones for copying, but instead of copying a file, the **rm** command is run through SSH, using the path of the container's file system.

Chapter 3

LXC

In this chapter, the process of generating an **LXC** container will be described in detail. The documentation I used can be found at [1] and [2].

3.1 Introduction to LXC Containers

LXC is an OS level virtualization solution, which has native support in the Linux kernel starting with version 2.6.29. It needs the **cgroup** kernel functionality to work. It is relatively recent (from 2008), and the documentation is pretty scarce. It took me a while to figure out the exact steps needed to generate a working container. The advantage **lxc** has over **OpenVZ** is the standard kernel integration. **OpenVZ** needs to use a modified version of the kernel (not available for all distributions) to be able to function.

A container consists of 3 parts: a configuration file, a mount points file (fstab) and a directory containing the root file system. These will be detailed in the next subsections.

3.1.1 Host Machine Configuration

To be able to run an **LXC** on a host machine, some configurations need to be made on the host. The **cgroup** functionality must be configured in the running kernel, then the **cgroup** file system must be mounted in /**cgroup**. For the network adapters in the container to work, at least one bridge interface must be configured on the host. A sample entry in the /**etc**/**network**/**interfaces** for a Debian system is shown in Listing 3.1. Finally, the **lxc** utilities must be installed: **lxc-create**, **lxc-start**, **lxc-execute**, **lxc-stop** etc. These can be installed either using the available package managers (**apt**, **yum**) or compiling the sources from the project site[1].

```
1 auto br0
2 iface br0 inet dhcp
```

[1] http://lxc.sourceforge.net/index.php/about/download/

```
3         bridge_ports eth0
4         bridge_stp off
5         bridge_maxwait 5
6         post-up /usr/sbin/brctl setfd br0 0
```

Listing 3.1: Bridge configuration

3.1.2 Configuration File

The hardware of the container is described in a configuration file. The configuration file is in plain text format, and consists of a list of key-value associations, for setting various properties of the container. There are some standard options present in the config file, which don't change across the different container configurations. The complete list of options and more details can be obtained from the manual page of lxc.conf (*man lxc.conf*). A sample config file is shown in Listing 3.2 (only the options that are relevant for the user). The **lxc.network.*** options are used to specify the network cards configurations. The location of the root file system and the **fstab** file (both relative to the config file's location) are specified by **lxc.rootfs** and **lxc.mount** respectively. The name of the machine is set by the **lxc.utsname** option.

```
1 lxc.utsname = TestMachine
2 lxc.rootfs = rootfs.TestMachine
3 lxc.mount = fstab.TestMachine
4 lxc.network.type = veth
5 lxc.network.link = br0
6 lxc.network.name = eth0
7 lxc.network.mtu = 1500
8 lxc.network.ipv4 = 172.16.30.160/24
9 lxc.network.flags = up
```

Listing 3.2: Sample config file

3.1.3 Mount Points File

The file containing the mount points (I will call it the **fstab** file) is a plain text file, which contains the mount points to be used inside the container. When the container is started, the will appear as the /**etc**/**fstab** file. It can be defined inside the root file system, as **$rootfs**/**etc**/**fstab**, or it can be defined outside the file system, and included in the container configuration file. A sample **fstab** file is shown in Listing 3.3. The file is stored outside the root file system, which is located in the **rootfs.TestMachine** subdirectory.

```
1 none rootfs.TestMachine/dev/pts devpts defaults 0 0
2 none rootfs.TestMachine/proc proc defaults 0 0
3 none rootfs.TestMachine/sys sysfs defaults 0 0
```

```
4 none rootfs.TestMachine/var/lock tmpfs defaults 0 0
5 none rootfs.TestMachine/var/run tmpfs defaults 0 0
6 /etc/resolv.conf rootfs.TestMachine/etc/resolv.conf none bind 0 0
```

Listing 3.3: Sample fstab file

3.1.4 Root File System

The file system directory contains the OS file structure and all the container's files. It is, in general, based on a minimal OS file structure, which is then modified according to the specific requests. There are some utilities for getting a minimum OS file system, like **debootstrap** for Debian systems, and **febootstrap** for Fedora systems. These utilities download from the Internet a specified version of the OS. To copy files inside the container, it is sufficient to copy them in the corresponding relative path to the root directory. The files of the container are also directly accessible, through their relative paths. This is useful for editing the system configuration files during the generation process.

I had some trouble using the **febootstrap** utility to download a Fedora minimal OS. I found usage examples, but none of them worked, because it said the arguments were invalid. I later found out that the latest version was 3 (which was also included in the system's repositories), which was completely different from the second one. All the examples I found were for the second version. I then downloaded the older version and the command worked as expected. The only place where I found out that there are multiple completely different versions of the program was the application's official page[1].

The container generation operation is distribution dependent. A Debian container can be generated on a Debian running host machine, and the same goes for a Fedora container. However, after a container is generated, it can be deployed and used on any distribution, regardless of its type. **vmgen** supports only 2 containers type: Debian and Fedora. A third option would be an Ubuntu container, but I couldn't generate a working container. I considered that Debian containers can be used instead of Ubuntu ones, because they have a similar file structure and configuration files.

3.2 The Setup

Because of the distribution dependent generation process, and to avoid cluttering the physical machine, the containers are created in a VMware virtual machine (support machine). There is a machine running Debian and one running Fedora. They are used to generate each container type. The machine that will be used is selected based on the **os** field in the application configuration file (more details in Section 3.3). The machines are configured as described in Subsection 3.1.1. Both machines run a SSH server, and they are configured to use public key authentication for the root user. The use of public key authentication allows the same set of keys (a private and a public key) to be used across the whole application, where needed, and eliminates the need to remember the

[1] http://people.redhat.com/~rjones/febootstrap/

password of the root user in each of the used machines. All the commands needed for the container creation are executed through a SSH connection (even if not mentioned explicitly).

3.3 Container Generation

The LXC archive, downloaded from the official site, contains some scripts for generating container: **lxc-debian**, **lxc-fedora**, **lxc-ubuntu**. As I said earlier, the **lxc-ubuntu** didn't work. The other 2 scripts seemed too complex, so I chose not to use them. Instead, I created my own set of scripts (BASH scripts): **my-lxc-debian.sh**[1] and **my-lxc-fedora.sh**[2].

A **CommanderLxc** and a **CommunicatorLxc** are used for the container generation process. The **Communicator** uses direct file manipulation on the container, and runs commands on the container using the **lxc-execute** utility. These operations are executed on the VMware virtual machine, through SSH.

The **CommanderLxc** module is the central module in the container generation process. It reads the OS from the input configuration file, and it calls the corresponding scripts, on the corresponding VMware machine to obtain the container. The selections are made using a stored dictionary, indexed by the OS identifier. The dictionary used to select the support machine and the script file is shown in Listing 3.4.

```
1  distro = {
2  "debian":{
3          "vm":"/home/vmgen/vmware/Debian␣(lxc)/Debian␣(lxc).vmx",
4          "hostname":"root@debian-lxc",
5          "script":"my-lxc-debian.sh",
6          "scripts-folder":"../scripts-lxc/debian/"},
7  "fedora":{
8          "vm":"/home/vmgen/vmware/Fedora␣64-bit/Fedora␣64-bit.vmx",
9          "hostname":"root@fedora-lxc",
10         "script":"my-lxc-fedora.sh",
11         "scripts-folder":"../scripts-lxc/fedora/"}
12 }
```

Listing 3.4: Dictionary for selecting OS specific elements

The support machine is powered on, then using the **Communicator**'s methods, the scripts are copied from a local folder onto it and executed there. The scripts for the 2 supported distributions are similar. First of all, the specified version of the file system is downloaded, using **debootstrap** or **febootstrap**. Because the download can be time consuming, an alternative would be to have the file systems downloaded in a local folder, and only copy them from there in the new location, instead of downloading them each time. After the file system is downloaded, various system configuration

[1] http://blog.bodhizazen.net/linux/lxc-configure-debian-lenny-containers/
[2] http://blog.bodhizazen.net/linux/lxc-configure-fedora-containers/

files are altered (using **sed**) or removed (some startup files). The root user password is set to a default value, to preserve the modularity of the application (it is the **Config** module's task to set the root password). The container config file and fstab are then generated. A fragment of the code used to generate the config file is shown in Listing 3.5. The used variables are initialized before, with the appropriate values. The network settings are specified inside the config file, by appending the needed options. After that, the container can be powered on, and the **ConfigLinux** and the corresponding **Installer** module (**InstallerApt** or **InstallerYum** are used to make the remaining configurations and to install the specified applications and services inside the container. An archive containing the container (config file, fstab file and root file system) is then created, and returned to the user.

```
1  cat << EOF > $config
2  lxc.utsname = $name
3  lxc.tty = 4
4  lxc.rootfs = rootfs.$name
5  lxc.mount = fstab.$name
6  lxc.cgroup.devices.deny = a
7  # /dev/null and zero
8  lxc.cgroup.devices.allow = c 1:3 rwm
9  lxc.cgroup.devices.allow = c 1:5 rwm
10 # consoles
11 lxc.cgroup.devices.allow = c 5:1 rwm
12 ...
13 EOF
```

Listing 3.5: Generating the config file

Chapter 4

VMware

4.1 Hardware Generation

A VMware virtual machine has 2 main components: a machine description file (**vmx**) and one or more virtual disks (**vmdk**). Each of these must be generated and configured.

4.1.1 The Machine Description File

The machine description file contains the virtual machine name, information about the OS and the hardware components of the machine (processors, memory, network adapters, hard-disks, cd drives etc.). It is a list of property-value associations, stored in plain text format. To generate the description file for the new machine, the information from the hardware section in the config file is used. The currently supported options in the config file are the OS type, the name of the machine, the number of processors, the amount of physical memory (RAM), the hard-disks, the cd drives, and the network adapters parameters. For the hard-disks and cd drives, the file path, the controller type (**ide**, **lsilogic** etc.) and the position on the controller (**ide0:1**, **scsi0:5**) can be specified. For a network adapter, the user can provide the type of connection to the host (**nat**, **bridge**, **host-only**), the hardware (MAC) address, and if the adapter is powered on when the machine boots up. Only some basic options must be specified in the **vmx** file. When the machine is powered on, the file is updated and some additional options are automatically added (like hardware addresses for the network cards if none specified, the PCI slot numbers etc.).

VMware does not provide an official list of options, but some extensive unofficial references can be found online at [5]. A nice tool I found when searching for the available options for the **vmx** file is an online application[1] which is able to generate customized **vmx** files. The user can select the desired options in a visual form, and then he is returned a generated **vmx** file.

A sample machine description (vmx) file, generated (by **vmgen**), is provided in Listing 4.1. For example, the **scsi0.*** options specify the HDD controller's and the drive's parameters. In this case,

[1] http://www.easyvmx.com/

the controller is **SCSI**. For an **IDE** controller, the syntax is similar to the one used for the cd-drives. The disks can be specified using **scsiX:Y** or **ideX:Y** fields (where X is the SCSI/IDE controller's number and Y is the controller's port index).

```
#!/usr/bin/vmware
config.version = "8"
virtualHW.version = "7"
guestOS = "debian5-64"

numvcpus = "2"
memsize = "256"
displayName = "First GenVM"

# hard-disk
scsi0.present = "TRUE"
scsi0.virtualDev = "lsilogic"
scsi0:0.present = "TRUE"
scsi0:0.fileName = "first-disk.vmdk"

# cd-rom
ide1:0.present = "TRUE"
ide1:0.deviceType = "cdrom-image"
ide1:0.fileName = "debian-6.0.0-amd64-CD-1.iso"

# ethernet
ethernet0.present = "TRUE"
ethernet0.startConnected = "TRUE"
ethernet0.connectionType = "nat"
```

Listing 4.1: vmx file sample

4.1.2 The Virtual Disks

A virtual disk is a file that can be created with an utility provided by VMware, **vmware-vdiskmanager**. It can be a single file, or split in multiple 2GB size files. It is basically, like a physical disk. The creation utility requires the size of the disk and the controller type (**ide**, **lsilogic** etc.). To create the desired virtual disks, the parameters of each disk is retrieved from the config file and passed to the **vmware-vdiskmanager** utility. Usage examples and the available options can be found out by running the **vmware-vdiskmanager** without any arguments. Example commands to create a SCSI and an IDE virtual disks are shown in Listing 4.2.

```
vmware-vdiskmanager.exe -c -s 800MB -a lsilogic -t 0 scsi-disk.vmdk
vmware-vdiskmanager.exe -c -s 7GB -a ide -t 0 ide-disk.vmdk
```

Listing 4.2: vmdk creation commands

4.2 Operating System Installation

One of the most important step in the virtual machine generation is the OS installation. VMware offers an API to interact with the virtual machine, but only after the OS is installed on it. The machine can be controlled using the VMware's **vmrun** command-line utility, or using the **pyvix** framework. The **pyvix** framework is a wrapper written in Python over **VIX**, the official API provided by VMware for controlling the virtual machines programmatically. The method used by the **CommunicatorVmware** is the **vmrun** one.

4.2.1 Unattended OS Installation Solutions

The first thing I tried to do was to find a way to install the OS unattended. For Linux[1], it is possible to provide a file containing the answers to the questions shown during the OS install process. However, I found out that there are different ways for different distributions: **preseed files** for Debian and Ubuntu, and **Kickstart** for RedHat and Fedora. I then found out about **fai**, an application which should be able to install any distribution unattended, but I couldn't get it to work.

For Windows XP[2], I found a way to create a simple text file containing the information needed during the setup process. This method didn't work in Windows 7[3], for which I found a large application which was able to generate an XML file containing the answers.

4.2.2 The Setup Used

Because there were different ways to install different systems, and because a system installation is a time consuming operation, which changes very little from one installation to another, I decided to take a different approach[4]. I manually installed some basic virtual machines with some of the OS supported by **vmgen** and I stored their disks in a folder (their **vmx** files are not necessary, because new ones will be created). The machines have only the OS installed, along with *VMware tools*, for easier interaction afterwards. When the user requests a new machine, the corresponding base disk is selected and the system partition is cloned from it onto the new disk.

An auxiliary machine is needed to partition the newly created disks and to clone a base installation onto it. I configured an additional virtual machine (**VMaster**), with a Debian system, with no GUI, and which was accessible through SSH using public key authentication.

[1] http://ixlabs.cs.pub.ro/redmine/projects/vmgen/wiki/AutoLinux
[2] http://ixlabs.cs.pub.ro/redmine/projects/vmgen/wiki/AutoXP
[3] http://ixlabs.cs.pub.ro/redmine/projects/vmgen/wiki/Auto7
[4] http://ixlabs.cs.pub.ro/redmine/projects/vmgen/wiki/UnattendedInstall

4.2.3 Setting Up The Partitions

After all the disks of the new machines are created, they are attached to the **VMaster**, along with the base disk corresponding to the desired OS. To attach the disks to **VMaster**, the **vmx** file of the **VMaster** is duplicated, and the necessary lines are added to the copy. The duplication is necessary, to avoid looking for the new options automatically added in the original file after the machine is started, in order to remove the disks after the job is finished. By cloning, the copy is simply discarded at the end of the process. The base disk is selected using a dictionary stored inside the **CommanderVmware** module. The dictionary is indexed by the OS identifier. The dictionary is shown in Listing 4.3 (only a limited number of systems is supported at the moment; more will be added after the application is finished and tested). Besides the name of the base disks (the path is a default one for all of the disks), the dictionary has entries for other parameters of the disks: the **MBR** type (**grub2**, **grub**, **win**), the file system type of the system partition, the disk controller type.

```
1  base_disks = {
2          "debian5-64":{
3                  "name":"debian-64.vmdk",
4                  "type":"lsilogic",
5                  "mbr":"grub2",
6                  "fs":"ext4"},
7          "ubuntu-64":{
8                  "name":"ubuntu-64.vmdk",
9                  "type":"lsilogic",
10                 "mbr":"grub2",
11                 "fs":"ext4"},
12         "windows7-64":{
13                 "name":"Win7-64.vmdk",
14                 "type":"lsisas1068",
15                 "mbr":"win",
16                 "fs":"ntfs"},
17         "winxppro":{
18                 "name":"WinXP.vmdk",
19                 "type":"ide",
20                 "mbr":"win",
21                 "fs":"ntfs"}
22         }
```

Listing 4.3: sample OS based dictionary

After the **VMaster** is started, the application needs to wait for the OS to boot up. This is accomplished using pyvix to wait for the VMware tools to load inside the guest. The methods that implement this functionality are shown in Listing 4.4. **VMaster** needs to have VMware tools installed, in order to use this feature. A thread waiting for the tools to be ready is created and started. The thread only exits if the tools are ready, or when it is killed by another one. To allow the application

to continue even if the thread did not return, a timeout is set. When the timeout expires, an error code is returned.

```
def connect_to_vm(vmx_path):
        try:
                # try defaults
                host = pyvix.vix.Host()
        except pyvix.vix.VIXException:
                print "error host"
        try:
                vm = host.openVM(vmx_path)
        except pyvix.vix.VIXException:
                print "error openVM"
        return (host, vm)

def _wait_for_tools(vm):
        try:
                vm.waitForToolsInGuest()
        except pyvix.vix.VIXException:
                pass

def wait_for_tools_with_timeout(vm, timeout):
        tools_thd = Thread(target = _wait_for_tools, args=(vm,))
        tools_thd.start()
        # normally the thread will end before the timeout expires,
           so a high timeout
        tools_thd.join(timeout)

        if not tools_thd.isAlive():
                return True
        return False
```

Listing 4.4: wait for VM to boot up

After the machine is started, the partition tables on each disk are created according to the provided setup in the config file (in the partitions sub-section). The **parted** utility is used to create the partition tables. A problem I encountered was that the type for the **Windows** partitions was reported correctly by **parted**, but it was reported as *Linux partition* in **fdisk** (which was the real type), and a Windows system running on that partition wasn't able to boot. The fix for this problem was to use the non-interactive version of **fdisk**, **sfdisk**, to change the partition type to **Windows type**. To create the file system on the partitions, the **mkfs*** utilities were used. The dictionary used to select the specific utility for a partition is displayed in Listing 4.5.

```
mkfs = {
        "ntfs":{
```

```
3                   "cmd":"mkfs.ntfs",
4                   "id":"7"},
5           "ext2":{
6                   "cmd":"mkfs.ext2",
7                   "id":"83"},
8           "ext3":{
9                   "cmd":"mkfs.ext3",
10                  "id":"83"},
11          "ext4":{
12                  "cmd":"mkfs.ext4",
13                  "id":"83"},
14          "swap":{
15                  "cmd":"mkswap",
16                  "id":"82"}
17  }
```

Listing 4.5: utilities to create file systems

4.2.4 Setting Up The OS

After the partitions are created, the system partition from the base disk is cloned onto the partition designed as the new primary (the first partition of the first new disk). For a **NTFS** system partition, the **ntfsclone**, along with the **ntfsresize** utilities were used, whereas for the **ext*** partitions, a simple **cp -ax** is used to copy the files from one partition to the other. An alternative for the **ext*** partitions is to use **dd**, but it takes more time to copy (because it must copy all the blocks of the partition, not only the used ones).

Then, the MBR of the new system disk is updated, to make it bootable. For the base disks having **grub** or a **Windows boot loader**, it is sufficient to copy the first 446 bytes (out of 512) of the original MBR, using **dd**. This includes only the boot loader part, without the partitions information (which is also stored in the MBR). For the **grub2** disks, the above method doesn't work, and the **grub-setup** utility needs to be executed, like in Listing 4.6 (the new system partition is located on the /**dev**/**sdc** disk, and it is mounted in /**mnt**/**new_hdd**/)

```
1  grub-setup -d /mnt/new_hdd/boot/grub /dev/sdc
```

Listing 4.6: configuring grub2

After the new disks are configured, the **VMaster** is shut down and the disks are detached from it. After the new disks are created and configured, the newly created machine can be powered on to continue the generation process.

I tested the partitioning and cloning for the various supported systems: Ubuntu, Debian, Windows XP, Windows 7.

4.3 System Configuration (On Windows Running Guests)

I was responsible for implementing the Windows configuration and installation modules. These modules are not VMware specific, but among the virtualization solutions we used so far, VMware is the only one to support a Windows running guest.

There is one problem with running commands on the guest system using **vmrun**: the command to be executed cannot be too complex (cannot have quoted arguments). Therefore, I chose to create a batch script, in which all the desired commands are listed, and then the file is copied onto the guest and executed there. Another advantage of the script is that there is no overhead for executing each command through **vmrun** (**vmrun** is called only one, to execute the script).

The **ConfigWindows** module has the methods described in Subsection 2.2.3. To avoid creating a script file for each configurations group (system, users, network, firewall), a string containing the whole script is created, and each method writes the necessary commands into it. In the end, the **applySettings** method is called, which writes the resulted string into a file on the disk (**`config.bat`**), and then copies the file onto the guest machine and executes it.

Although the documentation is not as detailed as the one for Linux, Windows has some powerful command line utilities which can be used to configure the system, like **net**, **netsh**, **wmic** etc. I gathered in a wiki page[1] the needed commands and arguments.

A sample generated **`config.bat`** file can be seen in Listing 4.7.

```
1  wmic COMPUTERSYSTEM where name="vmgen-pc" call rename "xp-gen"
2  net user Administrator pass2
3
4  net localgroup vmg /ADD
5  net localgroup julius /ADD
6  net user caesar alesia /ADD
7  net localgroup julius caesar /ADD
8  net user vv pass /ADD
9  net localgroup vmg vv /ADD
10
11 netsh interface ip set address name="Local␣Area␣Connection" static
       192.168.1.2 255.255.255.0 192.168.1.1 1
12 netsh interface ip set dns name="Local␣Area␣Connection" static
       192.168.1.254 primary
13
14 netsh firewall add portopening tcp 22 ssh
15 netsh firewall add portopening all 65000 myport
```

Listing 4.7: Sample generated config.bat

wmic is an utility used to display and change various system configurations. More usage information can be obtained by running **`wmic /?`**.

[1] http://ixlabs.cs.pub.ro/redmine/projects/vmgen/wiki/SystemConfig

The **net** command is used to configure the groups, the users and to control the system services. Only the users and groups features are used for now, to create groups, create users, and add users to the specified groups. Running **`net help [section]`** (where section can be blank and displays the sections, or one of the section displayed without the extra argument).

For the networking and firewall configurations, the **netsh** utility is used. It allows to see the current configuration and to change it. Usage information can be displayed by running **`netsh ...help`**, where the dots can be substituted by a chain of zero or more sections (e.g.: **`netsh help`**, **`netsh interface show help`** etc.). For the firewall, there can be defined both allowed ports and allowed applications (**vmgen** uses only the ports feature).

4.4 Application Installation (On Windows Running Guests)

On Windows, unlike on Linux, there is no central repository of applications. The supported applications are manually downloaded and stored in a folder on the server. There is a dictionary file, in which are stored the details for installing each application. If the administrator of **vmgen** wants to add support for additional applications, he can download the setup kits and add the corresponding entries to the dictionary.

Extensive documentation about installing application non-interactively on Windows can be found at [3]. There are 3 types of installers supported by **vmgen** : **msi** installers, **simple executable** installers and **archives**. There are different arguments to run the installer in quiet mode (non-interactive). The **msi** installers use the **msiexec** utility, written by Microsoft, and which has standardized arguments. The executable installers, in general, have the **"/S"** argument for quiet mode, but there are some exceptions. The archives are actually the application folder stored in a zip file. To install them, it is sufficient to extract them in a default path (usually in **`C:\Program Files`**). To be able to extract them, the machine needs to have an archiving application preinstalled (I preinstalled 7zip on the Windows machines, and added it to the system path, to be accessible from every folder). A fragment of the dictionary used by **InstallerWindows** is shown in Listing 4.8. There are sample entries for each type of installer, with or without additional arguments.

```
1  programs = {
2          "pidgin": {
3                  "type":"simple",
4                  "setup-file":"pidgin-setup.exe",
5                  "args":"/DS=1␣/SMS=0␣/L=1033␣/S"
6          },
7          "vim":{
8                  "type":"simple",
9                  "setup-file":"gvim-setup.exe"
10         },
11         "python":{
12                 "type":"msi",
13                 "setup-file":"python-setup.msi"
```

```
14          },
15          "emacs":{
16                  "type":"script",
17                  "setup-file":"emacs.zip"
18          }
19          ...
20  }
```

Listing 4.8: Program details dictionary

The installer module receives the list of applications to be installed and generates a **setup.bat** file, in which are written the commands for installing each application (see the previous section for why a script file is created instead of executing the individual commands). The needed install kits are added to an archive and copied, along with the **setup.bat** file to the guest machine. The batch script also contains the command for the extraction of the archive. The script is then executed on the guest machine, and the applications are installed. After the installation is completed, the script deletes all the kits and the initial archive from the guest machine. The script is then deleted from the guest and from the local folder too. A sample generated **setup.bat** file can be seen in Listing 4.9.

```
1  7z.exe x -oe:\test\ "e:\test\setup.zip"
2  del "e:\test\setup.zip"
3   e:\test\ThunderbirdSetup.exe -ms
4   msiexec /qb /i e:\test\python-setup.msi
5  del "e:\test\ThunderbirdSetup.exe"
6  del "e:\test\python-setup.msi"
```

Listing 4.9: Sample generated setup.bat

Chapter 5

Conclusion

So far, the application's core has been developed and tested. All the components were individually tested, and then they were integrated and the whole system was tested. I ran the full process for creating a VMware virtual machine, with Windows XP installed, and 4 applications, and it took about 5 minutes. For the lxc container generation, I tested the creation of Debian and Fedora containers, but without making the system configurations and installing any applications.

The application is usable, but it offers a limited number of features: only a small number of OS are available, a small number of applications are installable, the user interaction is not very friendly. We focused on designing the application and implementing it in a modular way, to make it easy to extend it later and we didn't have the necessary time to polish the application and make it fully usable. The project should be continued and the application should be released. It will be easy to extend it with new features, because of its structure. From my point of view, the application will be useful when it will be finished and it will be able to handle a wide range of configurations requests, allowing for a easier user interaction. Along with some caching mechanisms to speed up the generation process, the application will allow the users to obtain the needed virtual machines, spending much less time. They don't need to interact with the process, and **vmgen** configures the machines more quickly, because it can reuse some already generated pieces, and has no downtime between the operations (the next operation starts exactly after the current one is finished).

Chapter 6

Further Development

Now that the core application is implemented and tested, it can be extended, to allow more functionalities and better user interaction.

6.1 GUI

The application is somewhat difficult to be used as it is right now. The user needs to create from scratch a configuration file (or modify an existing one). This process is error prone, some typos might occur. Also, the user needs to know all the available options and possible values for them.

To ease the user interaction with the application, a GUI would help. The addition of a graphical frontend will allow more users to try the application and do not be discouraged by the complexity of the configuration file and avoids the typing errors. The user will see a Web frontend, where he can select the desired options. He is guided towards completing the whole file, and he is presented the available options at each step. The frontend module then generates a configuration file containing the options the user selected, and then passes it to the main application. This modular approach prevents the need to modify the existing application when adding a new frontend. In this way, multiple frontends may be added, according to the **vmgen** administrator's desire. The Web interface may look like the previously mentioned **EasyVMX**[1] online application.

6.2 Additional Virtualization Solutions

The supported virtualization solutions, so far, are **VMware**, **lxc** and **OpenVZ**. Support for other technologies can be added in the future. The **Commander** and **Communicator** modules for **VirtualBox** should be similar to the ones implemented for **VMware**. Some research needs to be done and see how **KVM** and **xen** virtual machines can be generated. After the needed operations are found, the corresponding modules for these virtualization solutions can be created. The modular architecture

[1] http://www.easyvmx.com/

of the application allows for easy addition of extra virtualization solutions, by implementing only the **Commander** and **Config** corresponding modules. In the **main** method, the selection of the module to be instantiated must be also modified, to include the new modules.

6.3 Additional Options In The Configuration File

Support for more installable applications can be added by adding the corresponding entries in the dictionaries of the **Installer** modules. In the future, there should be the possibility to provide additional parameters for the installers (like installation path, shortcuts created, additional settings after installation). This can be done by providing a script file containing the exact commands to be run by the installer process. This has the disadvantage that the user needs to know exactly the commands needed. It also presents a security risk, by allowing the user to enter code that will be executed directly, but fortunately, it is only executed on the guest virtual machine, and the potential damage is limited to only the guest. This feature is essential for configurable services, like a Web server, a DNS server, a DHCP server etc. These need additional settings to be made in order to be useful.

Also, the options available for specifying the hardware components are fairly basic, it should be possible to allow the user to specify more advanced settings, if he wishes to. Only a limited number of OS are supported at the moment. Support for more OS should be added in the future, by preparing the needed resources (preinstalled base installation disks for **VMware**, support machine for **lxc** and **OpenVZ** etc).

Bibliography

[1] Lxc howto. http://lxc.teegra.net/. [Online; accessed 03-July-2011].

[2] Lxc tutorials. http://blog.bodhizazen.net/tag/lxc/. [Online; accessed 03-July-2011].

[3] Unattended applications installation on Windows. http://unattended.sourceforge.net/installers.php. [Online; accessed 03-July-2011].

[4] vmgen wiki. http://ixlabs.cs.pub.ro/redmine/projects/vmgen/. [Online; accessed 03-July-2011].

[5] Vmx available options. http://sanbarrow.com/vmx.html. [Online; accessed 03-July-2011].

[6] Kenneth Hess and Amy Newman. *Practical Virtualization Solutions - Virtualization from the Trenches*. Prentice Hall PTR, 2009.

[7] VMware inc. History of virtualization. http://www.vmware.com/virtualization/history.html. [Online; accessed 03-July-2011].

[8] VMware Inc. Using vmrun to control virtual machines. http://www.vmware.com/pdf/vix180_vmrun_command.pdf. [Online; accessed 03-July-2011].

[9] David Marshall, Wade Reynolds, and Dave McCrory. *Advanced Server Virtualization*. Auerbach Publications, 2006.

[10] Wikipedia. Timeline of virtualization development. http://en.wikipedia.org/wiki/Timeline_of_virtualization_development. [Online; accessed 03-July-2011].