University POLITEHNICA of Bucharest

Automatic Control and Computers Faculty,
Computer Science and Engineering Department

BACHELOR THESIS

vmchecker - enhancement and scalability

Scientific Adviser:
As. Dr. Ing. Răzvan Deaconescu

Author:
Valentin Gosu

Bucharest, 2012

Abstract

The testing and grading of students' programming assignments is a process that requires a lot of time, but also a similar testing environment and an objective individual to do the testing. Vmchecker is a system that allows the automated testing and grading of programming assignments with little human involvement.

This project is an attempt to improve the system, by expanding the types of testing environments and making the process scalable by distributing tasks between several workstations.

Keywords: vmchecker, LXC, KVM, VMware, scalability, python, student assignment

Contents

List of Figures

Notations and Abbreviations

procfs – proc filesystem

Chapter 1

Introduction

1.1 Brief History of vmchecker

1.1.1 Motivation

For students in Computer Science classes, programming assignments are the main way in which they gain practical experience regarding certain concepts and technologies. These assignments are also an important opportunity to receive feedback and might also count for the student's final grade.

Student assignments are generally easy to assess. The process consists of downloading the student's submission, running a number of tests to check that it compiles and gives the right output and finally reviewing the source code to make sure that no obvious mistakes were made.

While the whole process is not a difficult one, it quickly becomes a tedious task when you need to assess more than 20 student assignments. Also, when a large number of assignments need to be tested, they are generally divided between teaching assistants who do perform the tasks individually. Considering each of them might test the assignments in a different environment, grading fairness and consistency is eventually lost.

Another issue with the classical aproach to grading programming assignment is that while providing the student with feedback upon the work he has done is one of the main objectives, that feedback often arrives too late. Homework is usually graded days or weeks after completion, so what would have been valuable advice during the development phase, is now too little too late.

Apart from the code-review stage of assesing a student submission, the process is fairly easy to automate. Also, while an automated system may not provide feedback on the quality of the submitted code, it could provide the student with immediate information regarding the behaviour of his homework given certain input data.

In order to automate the testing process as much as possible, the **vmchecker** project was initiated by the Computer Science and Engineering department at the University POLITEHNICA of

Bucharest.

This project aims to improve the performance of the vmchecker automated grading system by adding support for multiple virtualization types, simplifying its design and by insuring its scalability.

1.1.2 Features

vmchecker was built with regard to the requirements of active courses. This meant that the system has a number of mandatory features:

- A submission that potentially crashes the system during testing will not affect the overall functionality, as it's sandboxed in a virtual machine.
- The system can automatically detect if the student has sent his homework late and apply a penalty.
- Homeworks can be rechecked at any time, if the grader has concerns regarding the functionality of the submission.
- The system provides immediate feedback regarding the building and testing process to the students. Faulty or incomplete homeworks may be resubmitted.
- Access to the source files and the test output files is given to the grader.
- The web interface and submission repository are located on a different system from the one doing the actual testing. So even though the testing system might be unavailable, students can still submit their homework.
- The system is able to evaluate submissions in any programming language, and it may even receive an entire virtual machine for testing.

1.2 The Architecture of vmchecker

The vmchecker automated grading system consists of two separate subsystems:

1. The Storer
2. The Tester

The storer subsystem runs an webserver that allow the user to upload an archive containing the user's homework assignment. This archive is stored in a repository, along with the results of the homework's evaluation.

The tester runs a queue manager, which awaits tasks from the storer and upon receiving them proceeds to powering on a virtual machine, compiling the submitted source code, running certain tests and uploading those tests to the storer.

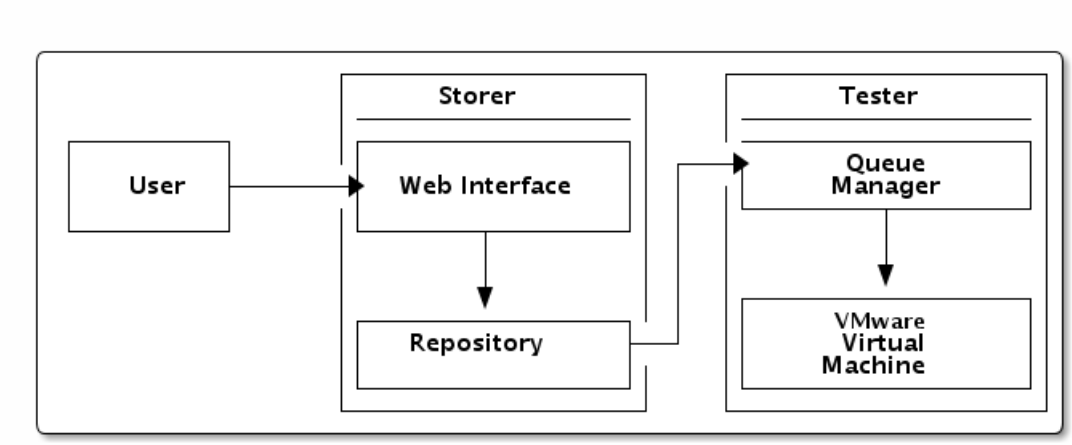

Figure 1.1: Simple Structure

1.2.1 The Storer

The storer is basically a Linux running machine that runs an Apache webserver in order to present the student with a web-interface so he can upload his assignment. As well as running the said webserver, the storer also hosts a repository of containing all of the student submissions, organized by course, assignment, student name and submission date. This is done in order to prevent the accidental overwriting of a submission.

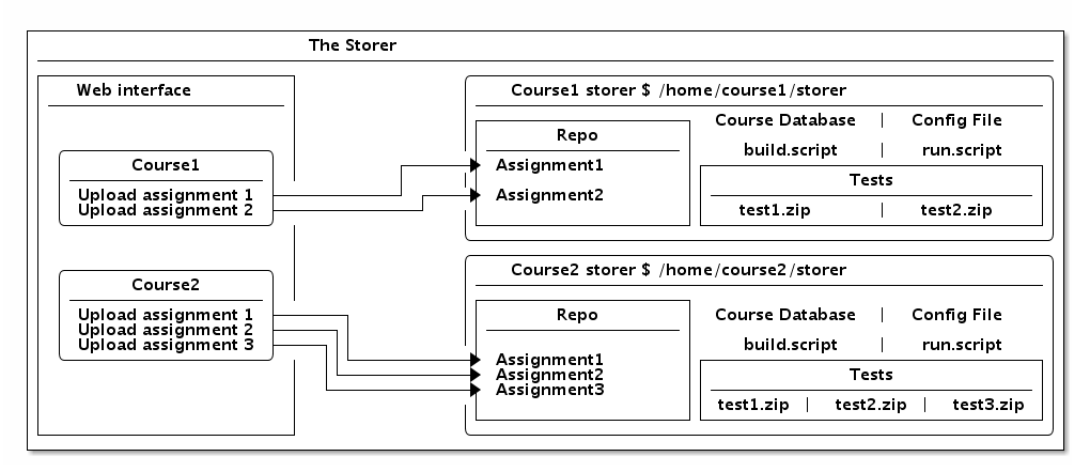

Figure 1.2: Storer Structure

The storer machine may host repositories for multiple courses. The storer must insure that only the authorized individuals have access to each repository. This is done by creating a separate linux user account to each course, and placing the repository inside the user's home.

Access to the machine is provided by SSH and authorization is done by placing the public keys of the account's users in the authorized_keys file.

The course's storer folder contains a config file, which lists the parameters for each assignment (such as deadline, title, virtual machine name), the configuration for each available tester (ip or url, username, queue path) and the configuration for each virtual machine.

Also in course's storer you may find a *tests/* folder, which contains an archive with each assignment's tests, several scripts to build and run each assignment, a database file, and a *repo/* folder.

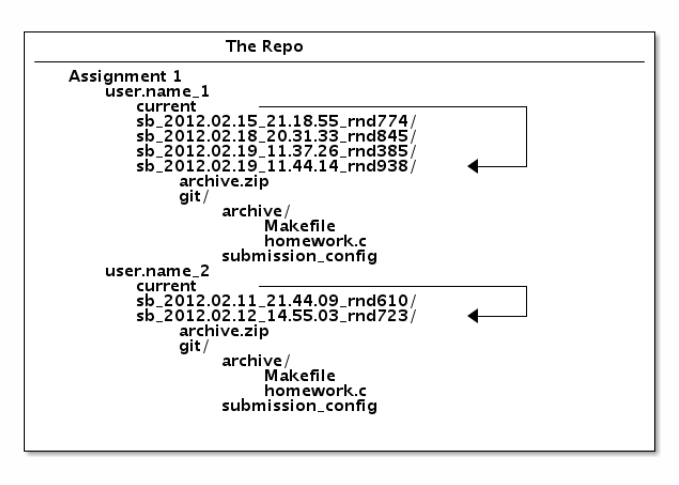

Figure 1.3: Repo Structure

The *repo/* folder contains every submission made to this course. It is organized by assignment, student name and subission date. A symbolic link is made to the most recent submission. The submission folder holds the original archive uploaded by the student, a *git/* folder with the unzipped contents of the archive and a file named *submission-config* which lists the submission's description, and a results folder which holds the output from the testing process.

1.2.2 The Tester

The tester is a separate Linux running machine which hass, just like the storer, a separate user account for each course. A process called *queue-manager* waits for file system events in the *queue/* folder. The storer uploads a new archive to the folder which consists of the student's submission, the tests, and the necessary scripts to build and run the files. When a new archive is uploaded, the queue-manager process extracts the contents to a folder and runs the executor.

The executor is a script that handles the virtual machine specific interactions. It powers on the virtual machine, it uploads the files, runs the build script and the necessary tests. The test results are contained in *.vmr* files. The executor then copies the result files to the bundle folder, shuts down the virtual machine, then uploads the results to the storer, in the folder designated by the *submission-config* file.

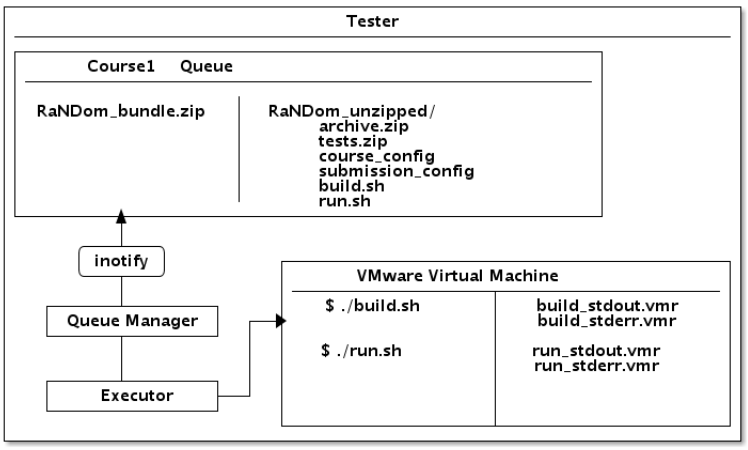

Figure 1.4: Tester Structure

1.2.3 Storer - Tester Interaction

Communication and file transfer between the storer and the tester is done through the SSH protocol.

Since the archived bundle is uploaded from the storer by the www-data user (which runs the apache server) it needs to have permission to read every course's private key. Permission is given by using the Posix ACL file permission system.

After the testing is completed, the queue-manager uploads the results to the storer using the path listed in the submission-config file. queue-manager is run by the course's tester account so it uses by default the account's private key to upload the files.

1.2.4 Installation

Installing the vmchecker system is a pretty complicated process. On the storer system, one simply needs to run the *setup.py* script, and then the *vmchecker-init-course* script, with the parameter *storer*. The last script initializes a new storer folder, with a default config file and an empty database.

The next step is generating the public and private keys and set the appropriate permissions so that the *www-data* user account can read them. Then add the appropriate settings for the assignments, tester and virtual machines to the config file. Keep in mind that the tester and virtual machines have not been set up yet.

The web interface is set up by creating a new user accound called vmchecker, setting up a *public_html* folder in its home and placing all the necessary files to this location. The next step is creating a new website configuration at */etc/apache2/sites-available/*

Finally, you need to edit the global configuration file at */etc/vmchecker/config.list* so the course will be listed in the web interface.

To install the tester a few extra steps are needed. vmchecker uses the VMware-VIX API to interact with the virtual machines. So you first need to install VMware server or WMware wokstation and the VMware-VIX package, then install the pyVix wrapper created by Lucian Grijincu.

The next step is to run the *setup.py* script, then the *vmchecker-init-course* script, with the parameter tester. The script will initialize the folder with a config file, and *queue/* and *tmpunzip/* folders.

The queue-manager script is then run.

1.3 Analysis of the Shortcomings of the Former Architecture

While it is a very useful and powerful tool, vmchecker has its share of drawbacks. These are caused by the way it was developed - in an incremental way, tackling each issue as it came up.

While rebooting the entire project might be the right course of action, backwards compatibility would be lost and the workload would proove monumental. We arrive to the conclusion that for the time being it's best to improve upon the existing model, while focusing on the major areas of concern.

1.3.1 The Virtualization Environments

From its inception the only virtualization environment supported by vmchecker has been VMware. This aspect leads to several flaws within the architecture.

First of all VMware-VIX API is needed to interact with the virtual machines. While there is only official support for the C language, a python wrapper is used to allow its integration with vmchecker's modules. It's needless to say that any modification to the API might break existing functionality and maintaining the python wrapper becomes an additional concern.

Beside the fact that it's very difficult to install on a Linux operating system using VMware-VIX API also has some requirements for the hosted virtual machines: VMware tools must be installed and running inside the guest in order to communicate with the virtual machine.

Another issue with VMware is the long startup time and the fact that it isn't immediately responsive following the restauration of a snapshot.

1.3.2 SSH Keys and Permissions

An important issue with vmchecker is the ease of setting up the tester and storer for a new course. At the moment the process is impeded by the fact that communication is done through the SSH protocol using public key authentication.

A new pair of keys needs to be generated, for both the storer and the tester account, then the appropiate permissions need to be set so that www-data has access to the keys.

1.3.3 Scalability

The existing support for a distribuited model in vmchecker is limited. While you may have multiple testers associated with the same storer, an assignment can only use one of those testers. While this is useful, since many courses may share the same storer, it also means that at times you'll have machines that are sitting idle, and machines that have a queue of over 20 submissions to be tested.

Also, in cases where the tester is a multicore machine, multiple virtual machines may be run at the same time. However, because of the current architecture of vmchecker, the extra performance of the host goes to waste.

In these cases, a dynamical way of distributing the workload between testers is needed.

1.4 Improving the Existing Model

This project aims to improve the performance of vmchecker.

All of the concerns listed in the previous section will be tackled as follows:

- Support for new virtualization platforms will be added, while keeping support for VMware. This will ensure a smooth transition and the new version of vmchecker will be backwards compatible.
- In order to eliminate the need for SSH keys for each account the communication model will be modified, such that a daemon process running on the storer will deal with the communication with the tester machines. The daemon process will also be the one to fetch the results from the tester machine, thus eliminating the need for the tester to know the storer's public keys.
- The queue-manager service will be able to provide status updates to the daemon, so it will be aware of the number and capabilities of all the available virtual machines. A load balancing service will be provided by the storer daemon, which will be able to properly place the submitted assignment in the least used queue.

Chapter 2

Extending the types of virtualization

The assignments that can be tested by vmchecker vary from simple C or Java programming assignments to entire virtual machines or kernel modules. The previous structure of vmchecker did not differentiate between these types of assignments in any way as it used a VMware virtual machine for everything.

In the next few sections we will look at other virtualization types and see if they hold any advantages to VMware.

2.1 LXC

While VMware offers an agreeable virtualization environment and a good isolation between the host and guest machine, simple user space assignments simply don't require full machine virtualization. While testing the submissions on the physical machine is not an option, because we need to provide the exact same environment for each test and because it would be vulnerable to fork-bombs and other types of attacks. However, an operating system-level virtualization environment would prove satisfactory for submissions that only require user space access.

Between many operating system-level virtualization methods, we have concluded that LXC is an acceptable choice because of the small effort required to deploy, run, and integrate with vmchecker.

2.1.1 Setting Up the Environment

In order to enable the usage of LXC as the virtualization type for testing an assignment, several steps must be completed. These steps are described in the documentation file found in the repository: *lxc-resources/INSTALL*

First of all, the required packages must be downloaded. To do this, the sysadmin must execute the *deps.sh* with root permission. The sysadmin should setup the interfaces needed to communicate with the container, by copying the contents of *lxc-resources/interfaces* to */etc/network/interfaces*.

To avoid using the effective IPs of each container, associations need to be added to */etc/hosts* as following the example in hosts file 2.1

```
10.3.1.1      deb0 deb0.virt deb0.virt.local # Host alias
10.3.1.11     deb1 deb1.virt deb1.virt.local # First container alias
```

Listing 2.1: Host Aliases

Networking should be restarted following this step.

The sysadmin should proceed to set up the LXC container. First of all, he should run the commands in listing 2.3 or add the following line to */etc/fstab* so the command will be run at startup:

```
none /cgroup cgroup defaults 0 0
```

Listing 2.2: Permanent mounting of cgroup

```
mkdir -p /cgroup
mount none -t cgroup /cgroup
```

Listing 2.3: Manually Mounting cgroup

The *lxc-resources/easylxc* script can now be used in order to create a new LXC container in which assignments can be tested. The script receives the name of the container as its argument. In the first stage of the project, only the first container name **deb1** was created and used.

Because the implementation of LXC was not yet finalized hence the **lxc-snapshot** command was unavailable, another way of restoring the container to a previous state was needed. So, after creating the container, the sysadmin needs to create a backup of the newly generated file system.

```
sudo lxc-backup deb1 1
```

Listing 2.4: Creating a Backup for the File System

Because some operations with containers require root access, in order to run the executor as a regular user the sysadmin should add the lines in listing 2.5 to the */etc/sudoers* file using the **visudo** command.

```
ALL     ALL=NOPASSWD: /usr/bin/lxc-start *
ALL     ALL=NOPASSWD: /usr/bin/lxc-stop *
ALL     ALL=NOPASSWD: /usr/bin/lxc-info *
ALL     ALL=NOPASSWD: /usr/bin/lxc-restore *
```

Listing 2.5: Uninteractive Sudo Permission

2.1.2 Implementing the LXC Executor

The component of vmchecker which handles interaction with the virtual machine is the executor, implemented in *bin/vmchecker-vm-executor*. This script is called by the queue-manager, on the

tester machine, and receives the bundle's path as its argument. The bundle represents a directory to which the contents of the received archive have been extracted.

Upon analysing the source code of the executor, it was fairly obvious that only certain sections needed to be modified in order to provide support for LXC vitualization. While this fact wasn't necessarily useful at this stage of the project, it became very important when trying to provide an common and extensible interface for all the virtualization types.

The original implementation was a collection of functions that passed a virtual machine object and a host object, and applied various actions to them. While certain functions, such as **connect_to_vm(vmwarecfg, vmx_path)** called certain VMware VIX functions and were specific to the VMware machine, others such as **test_submission(bundle_dir, vmcfg, assignment)**, **copy_files_and_run_script(vm, bundle_dir, machinecfg, test)** and **def start_host_commands(jobs_path, host_command)** were independent from any external API and even from the actual implementation of the methods they called.

My approach was to modify only the methods that were VMware specific, in order to provide a similar structure to both the LXC executor and the VMware executor. Control over the container was implemented by using the shell commands provided by LXC. Running commands inside the container was done by using a SSH connection to the container. The python source code is provided in listing 2.6

```
1  def guestRun(cmd):
2      """ runs the command inside the lxc container """
3      _logger.debug("Running␣in␣guest:␣%s" % cmd)
4      p = Popen(["ssh␣root@deb1␣"+cmd], stdout=PIPE,shell=True)
5      output = p.stdout.read()
6      _logger.debug("Guest␣output:␣%s" % output)
7      return output
8
9  def hostRun(cmd):
10     """ runs the command on the host """
11     _logger.debug("Running␣on␣host:␣%s" % cmd)
12     p = Popen([cmd],stdout=PIPE,shell=True)
13     output = p.stdout.read()
14     _logger.debug("Host␣output:␣%s" % output)
15     return output
```

Listing 2.6: Running Commands

Using the methods to run commands on the host machine or inside the LXC container, one could start the container by calling the powerOn() method, or run a command inside the container, that would return after a given timeout: runWithTimeout(cmd, timeout).

```
1  def powerOn():
2      hostRun("sudo␣lxc-start␣-n␣deb1␣-d")
3      while True:
```

```
4         o = hostRun("sudo␣lxc-info␣-n␣deb1")
5         if not o.contains("-1"):
6             return
```

Listing 2.7: Method Called to Power on the Container

```
1 def runWithTimeout(cmd, timeout):
2     try:
3         thd = Thread(target = guestRun, args = (cmd,))
4         thd.start()
5         thd.join(timeout)
6         return thd.isAlive()
7     except Exception:
8         return False
```

Listing 2.8: Method That Runs a Command Inside the Container

During the implementation of this project, the implementation of**LXC** was not yet finalized, so the **lxc-checkpoint** command wasn't available. The immediate solution was to create a backup of the container's filesystem to a backup location (*/lxc/rootfs*), and copy it back whenever restoring the state of the container was required. This was done by using the copy command of Linux, with the preserve ownership and reccursive options.

```
1 hostRun("rm␣-rf␣/var/lib/lxc/deb1/rootfs")
2 hostRun("cp␣-pr␣/lxc/rootfs␣/var/lib/lxc/deb1")
```

Listing 2.9: Restoring the Contaier to a Previous State

However, this method proved to be unsatisfactory since it required superuser access to the file system, and the executor would generally be run by another user. In order to bypass this limitation, two scripts provided by the developers of LXC would come into play. The **lxc-backup** and the **lxc-restore** scripts essentially create a copy of the file system and later restore it.

```
1     def revert(self, number = None):
2         if number==None:
3             number = 1
4         hostRun("sudo␣lxc-stop␣-n␣"+self.hostname)
5         hostRun("sudo␣lxc-restore␣"+self.hostname+"␣"+number)
```

Listing 2.10: Restoring the Container to a Previous State

2.1.3 Observations

Implementing an LXC executor has made **vmchecker** a much more flexible tool. While running a VMware machine inside another VMware machine is not recommended, mostly because of performance issues, running LXC inside the VMware machine does not noticeably increase the load of

the machine. This means that a single physical machine can now be used to host both the storer and tester, with the tester being sandboxed inside a VMware machine and the evaluation being done inside an LXC container.

2.2 KVM

As we described in the previous chapters, OS-level virtualization can only handle user space assignments. As for assignments such as kernel modules, we need full system virtualization. This is of course provided by VMware, however, as we mentioned, the environment is difficult to deploy on Linux operating systems and also the virtual machines require VMware Tools to be installed for a proper communication with the host.

One technology that seems to fit our needs is KVM. The fact that it is closely integrated with the Linux kernel and that it is able to support basically any guest operating system through QEMU make KVM a good addition to vmchecker's executor types.

2.2.1 Setting Up the Environment

Setting up **KVM** is an incredibly easy process. Only two packages need to be installed.

```
1 sudo apt-get install virtinst kvm
```

Listing 2.11: Install the Needed Packages

The next step is to create or use an existing kvm virtual machine. To achieve this task, we used the */var/lib/kvm/* folder, to which we copied an existing virtual machine, *saisp-vm.qcow2*.

The virtual machine needs to have the ssh daemon installed and the host's public key inserted into the authorized_keys file of the guest. To do this, the sysadmin should run the command in listing 2.12. This powers on and opens a window to interact with the virtual machine.

```
1 sudo virt-install --connect qemu:///system --name kvm --hvm --ram
    512 --disk path=/var/lib/kvm/saisp-vm.qcow2,format=qcow2 --
    network network=default --import --vnc
```

Listing 2.12: Initial run of the virtual machine

The **qcow2** format is very convenient to our needs. It is created having another image file as a base. Any modifications the user might make to the file system, will only be listed in the newly created file. In order to revert to the original file system, one only needs to create a new copy-on-write image file and use that to run the virtual machine.

The next step in the installation process is to create a copy-on-write file with the virtual machine image file as its base.

```
1 qemu-img create -f qcow2 -b saisp-vm.qcow2 base.qcow2
2 cp base.qcow2 run.qcow2
```

Listing 2.13: Creating the Target File

The virtual machine will then be installed started, so the domain name is available from the executor. Installation will be done without a graphical interface, as it's unneeded.

```
1 sudo virt-install --connect qemu:///system --name kvm2 --hvm --ram
    512 --disk path=/var/lib/kvm/run.qcow2,format=qcow2 --network
    network=default --import --graphics none
2 virsh destroy kvm2 # stop the VM
```

Listing 2.14: Install the KVM Domain

At this point, the KVM domain is installed and available.

2.2.2 Implementing the KVM Executor

Implementing the KVM executor posed different issues from LXC. While the overall aproach was the same, the general architecture of KVM created other challenges.

Managing the virtual machines was done in a similar way, by using the available shell commands provided by **virsh** while calling the hostRun(cmd) method.

One of the major challenges was getting the virtual machine IP. While the LXC container had a static and well defined IP, the KVM virtual machine gets its IP through DHCP. The initial approach was to use the serial console of the virtual machine an run the **ifconfig** command on the guest in order to get its IP. However, this proved unreliable and difficult to automate. The solution we settled on was to run the **arp** command on the host, in order to get the MAC-IP associations. The virtual machine's MAC address can be recovered from the guests XML description.

```
1  def getMac():
2      mac = hostRun("virsh␣dumpxml␣kvm2")
3      mac = mac[mac.find("<mac␣address=")+14:]
4      mac = mac[:mac.find("'/>")]
5      return mac.strip()
6
7  def getIP():
8      mac = getMac()
9      while True:
10         arps = hostRun("arp␣-a").split("\n")
11         time.sleep(1)
12         for arp in arps:
13             if mac in arp:
```

```
14                  IP = arp[arp.find("(")+1:arp.find(")")]
15                  _logger.info("IP: %s" % IP)
16                  return IP
```

Listing 2.15: Getting the KVM Guest's IP

2.3 Implementing a Generic Executor

Having three different scripts to act as executors is unacceptable considering that the queue-manager is totally unaware of both the assignment's settings and the executor's implementation. This leads to the conclusion that solutions need to be integrated in the same executor. As one might also notice, the three implementations of the executor also share a large portion of code and would benefit from a refactoring that uses class inheritance.

From the executor's source code we can identify three major method categories: Host related methods, guest related methods and main method, that parse the bundle's parameters and call the apropriate functions in order to run the tests.

Two host class was defined as shown in listing 2.16. Classes that extend Host need to override the getVM method in order to return a VMware, LXC or KVM virtual machine object. The other methods, such as **executeCommand** and **start_host_commands** are already implemented and can be called without modifications in classes that extend Host, or in other classes or methods.

```
1  class Host():
2      def __init__(self): pass
3      def executeCommand(self, cmd): pass
4      def getVM(self, bundle_dir, vmcfg, assignment): pass
5      def start_host_commands(self, jobs_path, host_command): pass
6      def stop_host_commands(self, host_command_data): pass
```

Listing 2.16: Generic Host Implementation

The VM class is a collection of methods that define a way to interact with the given virtual machine. While some methods are implemented, others that are virtual machine dependent are not. These methods, such as **start**, **stop** and **revert**, need to be overriden in the class that extends VM and implements the object that defines the interaction with a certain virtualization environment.

```
1  class VM():
2      host         = None
3      path         = None
4      username     = None
5      password     = None
6      IP   = None
7      def __init__(self, host, bundle_dir, vmcfg, assignment):    pass
8      def executeCommand(self, cmd):          pass
```

```
9     def executeNativeCommand(self, cmd):    pass
10    def hasStarted(self):   pass
11    def start(self):        pass
12    def stop(self):         pass
13    def revert(self, number = None):                    pass
14    def copyTo(self, targetDir, sourceDir, files):      pass
15    def copyFrom(self, targetDir, sourceDir, files):    pass
16    def run(self, shell, executable_file, timeout):     pass
17    def runTest(self, bundle_dir, machinecfg, test):    pass
18    def try_power_on_vm_and_login(self):                pass
19    def test_submission(self, buildcfg = None):         pass
```

Listing 2.17: Generic VM Implementation

The new step was to modify the naive scripts implemented for each virtualization method into new vmchecker modules. Creating a module for the VMware executor was very simple, and only required translation of the existing methods into a new class.

As one can see in Diagram 2.1, there are now three classes that extend Host: VMwareHost, lxcHost, kvmHost. When calling the **getVM** method of each of these classes, the coresponding virtual machine object is returned: VMwareVM, lxcVM, kvmVM.

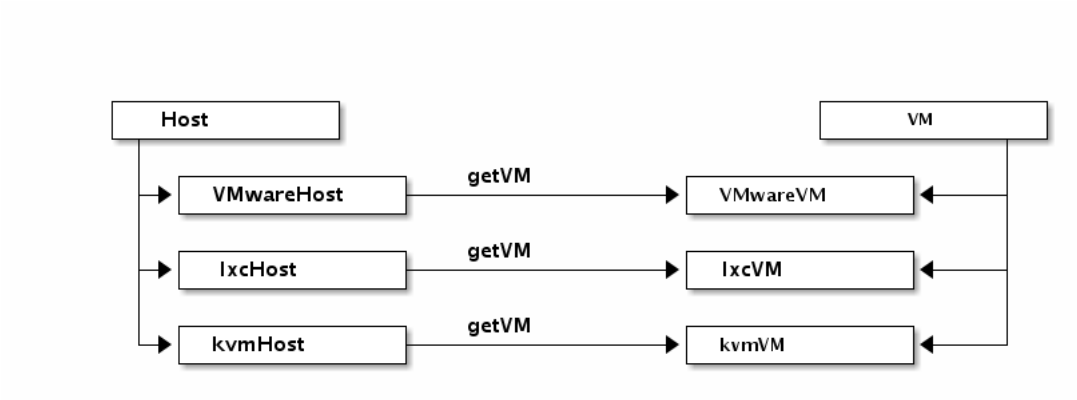

Figure 2.1: The class diagram

Chapter 3

Redesigning the Communication Model

3.1 Overview

The communication between the tester and the storer is currently done through SSH. Authentication is done through public-private key pairs. For example, let's take **course1**. When setting up the storer and tester for the course, the user course1 is created on both the storer and the tester. Keys are generated for each of them, and the public keys are added to each other's authorized_keys file. As one can see in Figure 3.1 there are two steps to the communication process: Step (1) is triggered by the student submitting an assigment. The files are saved in the repository, and a bundle is created containing the tests and scripts to build and run the tests. The bundle is transferred by the www-data user to the tester using SSH key 1, to the designated location in the course's *config* file.

Step (2) is triggered by the evaluation process being completed. The results are returned to the storer, by the queue-manager process running with course1's permissions, using SSH key 2. The *.vmr* files are saved on the storer to the location set in the *course-config* file.

3.2 Authentication and Security

Even though the communication model between storer and tester is reliable, adding a new course to the system involves a great amount of work, mostly because authorization requires generating and deploying SSH keys and setting the appropriate permissions.

Another important issue is that by the current model, the tester is the one that copies the results to the appropriate location on the storer. This is a serious security issue. Even though the tester doesn't really host important information, an attacker that has access to the tester can potentially access and destroy information located on the storer machine.

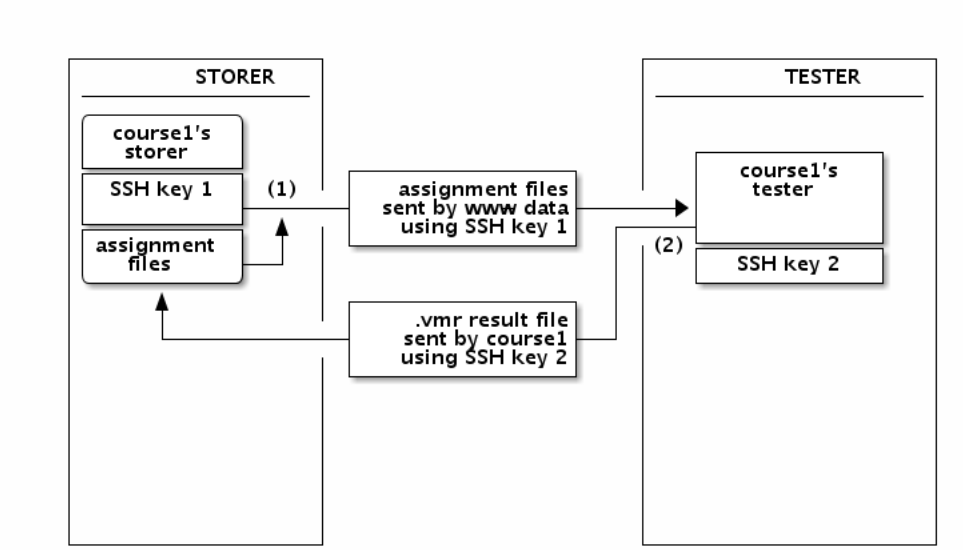

Figure 3.1: Communication Model

In order to eliminate this flaw, ideally the tester would have no access to the storer. Instead of the tester pushing the results to the storer, the storer would pull the results when notified. Even though this notification will be done through a less secure channel, the storer's integrity cannot be compromized.

Our approach was to introduce another element to the storer, that takes care of communicating with the tester or testers. This element will run as a separate process having permissions to access any of the files in any course's repository. We'll call this process the **daemon**.

Although it introduces some additional complexity to the diagram having the daemon in the system actually simplifies a large part of the communication. First of all, course1's storer no longer needs to hold a SSH key, as the only needed key held by the daemon. Secondly, the tester doesn't require each course to have it's own user account anymore. Since there is no possibilty to get access to the storer from the tester, any administrator that needs to set up a testing environment can simply access it directly.

The newly defined communication model works in the followin way:

The submission of an assignment in the web interface copies the files to the repo (1) and calls the queue_task() method of the daemon (2) which in turn creates and sends the bundle to the tester (3). Upon completion of the evaluation process, the tester calls the notify() method of the daemon (4) which copies the results from the tester (5) and places them in the repository (6).

For a secure daemon some restrictions have to be imposed. First of all, the queue_task() method must only be called from the storer and the notify() method must only be called from the tester. Calls from any other machines should fail. Secondly, when submitting an assignment to the tester, the daemon must retain the location in the course's repo, where the results will be saved, and issue a random key. This way, the location on the storer can't be modified by an attacker with access to

the tester.

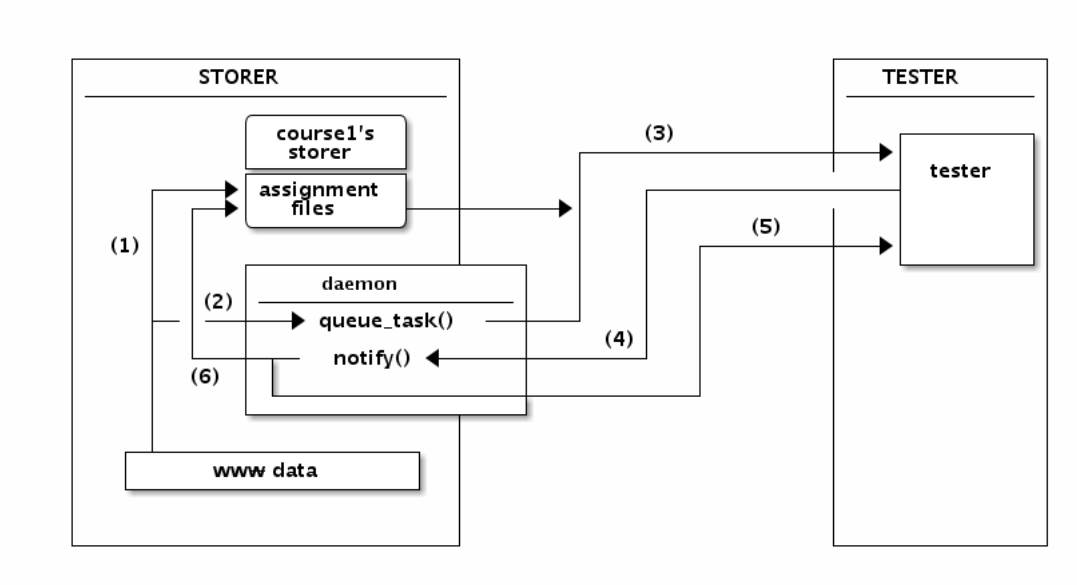

Figure 3.2: New Communication Model

3.3 Daemon's Implementation

Implementing the daemon was a rather simple task. The methods located in *submit.py* that were originally called from the web interface when a user submitted the assignment, were moved to *storer_daemon.py*. In order to provide access to these methods, python's XMLRPC was used. The daemon acts as a server and listens on a designated port for requests. The callers, which may be the storer itself or the tester machine will simply connect to the storer's port and issue a request.

Providing selective access to the methods was done by extending the **SimpleXMLRPCServer** and **SimpleXMLRPCRequestHandler** classes. When a method is called, the dispatcher checks if the function's name is in the restricted_functions dictionary. If it is, it then checks if the client IP matches any of the IPs defined in the list. A client that isn't in the list is not allowed to call that function, so the request fails.

```
1 class SelectiveHandler(SimpleXMLRPCRequestHandler):
2     def _dispatch(self, method, params):
3         server = self.server
4         clientIP, clientPORT = self.client_address
5         if method in server.restricted_functions:
6             if not clientIP in server.restricted_functions[method]:
7                 raise Exception('Method not supported for this IP')
8         func = None
9         try:
```

```
10              func = server.funcs[method]
11          except KeyError:
12              raise Exception('method␣"%s"␣is␣not␣supported' % method)
13          if func is not None:
14                  return func(*params)
15          else:
16              raise Exception('method␣"%s"␣is␣not␣supported' % method)
17
18  class SelectiveServer(SimpleXMLRPCServer):
19      restricted_functions = {}
20      def register_function(self, function, name = None, IP = None):
21          if not IP is None:
22              if name is None:
23                  name = function.__name__
24              if name in self.restricted_functions:
25                  self.restricted_functions[name].append(IP)
26              else:
27                  self.restricted_functions[name] = [ IP ]
28          SimpleXMLRPCServer.register_function(self,function, name)
29
30  srv = SelectiveServer(("", RPC_PORT), SelectiveHandler)
31  srv.register_function(queue_for_testing, IP='127.0.0.1')
```

Listing 3.1: SelectiveServer implementation

Since the daemon needs access to the repositories in order to create the bundle, it needs to be run with www-data's permissions or under another dedicated account. This account needs to be given full read-write permission to the repository by the *vmchecker-init-course* script.

Chapter 4

Load Balancing and Scalability

One of the major drawbacks of the current vmchecker is that at times an entire tester may be sitting idle, while another has over a dozen submissions to evaluate. The reason for this is that each assignment's tester is statically set in the course's *config* file.

The answer to this drawback would be to implement a dynamic way of selecting the tester with the shortest queue for testing a bundle. There are however a few issues to take into consideration.

The first problem to this approach is that testing a certain submission might require certain packages to be installed on the host operating system, that certain virtual machines might not exist, or more generally that evaluating a submission might fail, even though the submission should succeed.

Another problem is the way you assign submissions to each tester queue. While the web interface is able to check how many submissions are in a queue, it doesn't have any way of estimating the time an evaluation might take.

In order to tackle these issues, when notifing the storer that it has finished evaluating an assignment, the tester will also provide the running time. Failing to contact the tester, to start the virtual machine or no receival of the notification will be counted as taking a long time. This means that submissions for that assignment will not be directed to that virtual machine.

4.1 Estimating the Evaluation Time

The purpose of creating a scalable system is to reduce the load on the tester machine when evaluating a homework assignment and to reduce the time necessary to test each submission.

The time necessary to evaluate a submission consists of two parts: the running time of the actual program and the time waiting for other submissions to be tested.

The running time can be recorded by noting the start and end time of the executor script. The actual running time of the application inside the virtual machine can also be noted, by recording the start and end time of the executor commands.

In order to estimate the total waiting time for the submissions, more factors need to be taken into consideration:

- How many submissions are in a queue
- The estimated time for each of the submissions
- The course and assignment number of each submission

We define the estimated time to evaluate a submission for a certain assignment in a certain queue as follows:

$Et_{q_x}(assignment\_id)$ = $\frac{1}{n}\sum_{i=0}^{n} Time(i)$, where $Time(i)$ is the time required to evaluate the submission and $assignment\_id$ is the actual assignment_id as listed in the course's config file.

So the total time to evaluate all the submissions in a queue (q_x) would be defined as:

$T_{q_x} = \sum_{i=0}^{n} E_t(Assignment(i))$, where n = number of submissions in queue x.

In conclusion, the time required to evaluate a new submission would be the minimum time to test all of the submissions in a queue plus the estimated time to test the given submission.

$T = MIN(T_{q_i} + Et_{q_i}, i \in [0..n))$

4.2 Implementation

The load balancing system will be implemented by modifiying the **storer_daemon** and the **queue_manager**. First of all, the daemon will retain the estimated time needed to evaluate a submission, grouped by the tester's IP and by the assignment's id. Also, after evaluating each submission, the queue_manager will notify the daemon with the time it took to evaluate the last submission. This way, the daemon will hold the latest information regarding the estimated evaluation time.

Chapter 5

Testing and evaluation

In order to gain a good understanding on how the virtualization methods compare to each other, a few tests have been done.

The results are listed below:

Virtualization method: LXC

Total time	Running time
4.292s	0.4249s
4.311s	0.4363s
4.250s	0.4277s
4.311s	0.4406s

Virtualization method: VMware

Total time	Running time
30.404s	2.1983s
30.056s	2.1971s
30.190s	2.1984s
30.173s	2.1973s

Virtualization method: KVM

Total time	Running time
26.320s	0.2821s
25.453s	0.2298s
28.629s	0.2450s
26.482s	0.2831s

There are some interesting observations that can be drawn from the data. First of all, it is noticeable that LXC is by far the best virtualization method in terms of evaluation time. It is almost 10 times as fast as VMware. As profiling the python scripts revealed, a noticeable part of the time was occupied by the SSH connection, used for transferring the files in and out of the container. However, when using the Linux native copy command, **cp**, that time was reduced by up to a second. However, SSH is a much more consistent method to achieve this task.

Another interesting obeservation was that KVM actually has a better running time for the actual program than any other virtualization method, although the total time is close to that of VMware. However, considering the drawbacks of VMware, that makes it a worthy candidate.

The testing was performed on an Intel Core i7-2630QM machine at 2.0 Ghz, running Ubuntu 11.10. The testing bundle was the same in all cases, consisting in a computationally intensive homework for the Operating Systems course.

Chapter 6

Conclusion

This project represents a big improvement to the overall performance of **vmchecker** . While the system's ability to automatically evaluate and grade an assignment has not been affected, its performance has been greatly improved, especially at times of high workload.

- Extending the virtualization methods to LXC and KVM has reduced the project's dependance of VMware and Vix-API. These are proprietary technologies and the pyvix wrapper is quite difficult to maintain. Also, having all the executors extend a generic class makes adding a new virtualization type a very easy task.
- Redesigning the communication model has facilitated the installation process and eased the administrator's job when adding a new course to the system. Keys no longer need to be generated for each course, the permission settings that need to be set are in lesser number, and the storer's security has been greatly improved.
- The changes we have made to the **queue-manager** and **storer-daemon** modules cut the time required to evaluate a large set of submissions.

Chapter 7

Further development

Although **vmchecker** is a perfectly usable system, there is always room for improvement. We have identified some of the areas where **vmchecker** lacks support and intend to develop them in the future.

- **Web interface for grading the submissions**. Currently this is done by remotely connecting to the storer, reviewing the source code and the test results and writing the score in a text file. A web interface for grading and administration would improve both the usability and security of the system.
- **Storer access for students**. Accidents where a student uploads a homework in the wrong section happen with increased regularity. This is usually fixed by a course administrator. A student's ability to manage his own submissions would be a great improvement.
- **CLI utility for submitting assignments**. A command line tool for submitting the homework might prove very useful to many students, especially those who prefer to work on systems without a graphical user interface.
- **Cache for LDAP authentication**. Presently students use their LDAP user account and password to authenticate in the web inteface. However, if the LDAP server ever fails that means that vmchecker, although running, cannot be accessed by students.
- **MPI and OpenMP support**. In order to integrate these testing platforms into **vmchecker** , one only needs to extend the generic executor and implement the methods needed to interact with the system used for testing the MPI or OpenMP assignment.
- **Debian installation package**. Installing **vmchecker** on a new system is a difficult task, especially for those who have never used it before. A package that would take care of the installation process would be a great step forward regarding the popularity of **vmchecker** as an automatic grading system.

Bibliography

[1] Easy lxc installation script. Online. http://www.v13.gr/blog/?p=192.

[2] Kvm tutorial. Online. http://elf.cs.pub.ro/saisp/wiki/laboratoare/laborator-04.

[3] Logging xmlrpc server. Online. http://code.activestate.com/recipes/496700-logging-simplexmlrpcserver/.

[4] Lxc - how to. Online. http://lxc.teegra.net/.

[5] Official kvm website. Online. http://www.linux-kvm.org/page/Main_Page.

[6] Official lxc website. Online. http://lxc.sourceforge.net/.

[7] Python xmlrpc. Online. http://docs.python.org/library/xmlrpclib.html.

[8] Vix api documentation. Online. http://www.vmware.com/support/developer/vix-api/.

[9] vmchecker. Online. http://soc.rosedu.org/uploads/SCS_2008-2009_vmchecker_paper.pdf.