Masaryk University
Faculty of Informatics

Cloud Scheduler for OpenNebula Middleware

Master's Thesis

Bc. Gabriela Podolníková

Brno, Fall 2016

Replace this page with a copy of the official signed thesis assignment and the copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Gabriela Podolníková

Advisor: RNDr. Dalibor Klusáček Ph.D.

Acknowledgement

I would like to thank my supervisor RNDr. Dalibor Klusáček Ph.D. for all the help during those years we were working together. Also I would like to thank Mgr. Boris Parák for being my adviser when it comes to OpenNebula nuances. Lastly, I am very grateful to my family and my boyfriend David for always encouraging me to continue.

Abstract

This thesis focuses on the proposal of the new scheduler for OpenNebula. OpenNebula is a platform for building and managing cloud infrastructure. Scheduler is usually one of many components these cloud managers offer. In this work, we describe the current scheduler that OpenNebula provides, its features and its drawbacks. The practical part of the thesis presents the design and implementation of the new scheduler. The newly proposed scheduler is called ONEScheduler. ONEScheduler was designed to have modular architecture, and to be easily extensible and configurable. The scheduler is implemented as a maven application build in Java 8 using frameworks, like Spring, Jackson or XPath.

Keywords

Cloud, Cloud computing, OpenNebula, scheduling, scheduler, virtualization

Contents

1 Introduction

In the past few years, cloud has became an essential part of the everyday world. The trend is to move the data from desktops to large data centers [1] and provide scalable computing resources and easy accessibility. Hence, cloud computing is on the rise.

Peter Mell and Timothy Grance describe cloud computing [2] as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction". Consequently, cloud computing is bringing scalable computing capabilities on-demand. These capabilities are shared over the Internet and are independent of the consumer's location. The consumer can demand these capabilities in a given quantity and at any time. Resources are monitored and the customer is being charged for the quantity of utilized resources.

Therefore, the essential characteristics of the cloud model are on-demand scalable services, elasticity, network access, multi-tenancy and pay-per-use system. These features are enabled by the cloud infrastructure. The cloud infrastructure consists of a collection of hardware and software. The hardware represents the physical resources like storage, computing devices, servers and network. The software is an abstraction layer that enables essential characteristics of the cloud model.

Depending on the type of the provided service, cloud computing can be classified into three categories [3]. Provided services are either infrastructure (IaaS), platform (PaaS), or software (SaaS), listed by their decreasing flexibility and difficulty level [1]. *IaaS* model offers a scalable computing infrastructure to the consumer. *PaaS* provides the ability to use the cloud for deploying and managing consumer-owned applications. Finally, the *SaaS* model offers only the usage of provider-owned applications.

Cloud can have four possible deployment models. The *public cloud* is accessed over the Internet and is available to anyone who is paying for the service. The *private cloud* is owned, built, and then managed by a single organization. In the *community cloud* the cloud infrastructure

is shared among several communities sharing a common concern (security, mission). Finally, the *hybrid cloud* combines both the public and the private characteristics [1].

For the end user, cloud is one big, rather homogeneous, resource while in reality, the physical infrastructure of the cloud is often distributed and heterogeneous — a fact which is mostly hidden to a user. This level of abstraction is achieved by the process of virtualization. Virtualization simplifies management of the physical hardware. These physical machines are running an operating system (OS) and a *hypervisor*. The *hypervisor* enables the virtualization of resources (CPU of the physical machines, I/O devices), creating, running and managing *Virtual Machines* (VM). VMs represent a simulated computer environment equipped with a *guest OS*.

With the growth of the demand for cloud services, many management systems have been developed. They are used for managing the cloud environment. Some of the managers come from academic background — like OpenNebula [4] — and some of them were designed by companies — e.g., OpenStack [5] and CloudStack [6]. This work focuses on the cloud management platform OpenNebula. OpenNebula was released for the first time in 2008 and since then this open source project aims to ease the process of building an enterprise cloud. It serves as a resource manager (RM) that offers space for deploying VMs. RM also controls the life cycle of these VMs. However, only the deployment itself is not enough. RM should contain a module that decides when a VM will be deployed and on which resource. This component is called the *scheduler* and *scheduling* represents the allocation of VMs onto resources in time.

The *scheduling problem* [7] can be defined as a problem of finding a solution that fulfills certain *criteria*. These criteria are mathematically expressed using so called *metrics* (objective functions). Then the quality of the solution depends on the value of the given metric(s). These metrics can be either resource-centric (used to study the system performance), user-centric (includes fair-sharing of the resources among users, user's waiting time, reliability), or VM-centric (allowing to measure the VM performance). Thus, the goal of the scheduler is to optimize one or more of these metrics. A *scheduling policy* is a specific scheduling algorithm which goal is to optimize given metrics. A policy can determine which VM will be processed first, which user

has higher priority, or how the resources will be utilized, i.e., balanced, maximized or minimized. The solution produced by the scheduler is a product of combining several different policies. For example, we can use a fairness policy together with some resource selection policy, like First Fit or Best Fit.

Another resource-centric approach that is used in OpenNebula's default scheduler [8] is overcommitting the capacity of the resource. Overcommitment of resources is an approach that assigns a VM on a resource even if there is not enough free space. This technique is applied because VMs are usually requesting more computational space than what they actually need. Therefore the resource does not use its full potential.

A scheduler can be either static or dynamic. The former considers only requirements of the VM and with that knowledge places the VM on the resource. The latter adapts to the current situation in the infrastructure like failures, restarts, temporal unavailability of resources, and load balancing. A dynamic scheduler redistributes already running VMs on other resources in order to optimize the resource usage. This process is called migration [9].

The process of finding the optimal schedule comprises of developing many different approaches [10]. The efficiency of these new scheduling algorithms needs to be proven before they can be applied to the real infrastructure. Therefore, there is a need for a tool that can emulate the infrastructure and run a set of workloads in order to test the algorithm. These tests need to be iterated several times, thus it is necessary to have the ability to reproduce the environment. Hence, scheduling simulators were introduced.

As we can see the scheduling can have many approaches aiming to satisfy some criteria. Unfortunately OpenNebula's scheduler is not designed for easy extensions and is not very well documented. For these reason, and also that it is a stand-alone module, third parties tried to create a new scheduler, such as Haizea [11], that is based on VM lease requests [12], or Green Cloud Scheduler [13] that is oriented to energy-awareness [14]. These projects were not very successful, thus they are no longer active.

The goal of this thesis is to design a new scheduler component for OpenNebula that is easily extensible with new scheduling algorithms. The scheduler contains the usual modules for managing typical re-

quirements like prioritization of users, filtration of available resources, and scheduling VMs onto resources. Unlike the other attempts to design a new scheduler for OpenNebula, our architecture is modular, hence enabling easy extensions. The main part of this work is to implement an efficient way for representing the data and logical parts of the scheduler. Lastly, the scheduler is equipped with the possibility of switching the scheduler into simulation mode.

This work is organized as follows. First, the Chapter 2 introduces the infrastructure of OpenNebula. Subsequently, Chapter 3 presents the OpenNebula's existing scheduler. The following chapter proposes the design of our approach, the architecture and implementation details. The last two chapters conclude this work with the evaluation of our approach and suggestions for future work.

2 OpenNebula - Cloud Management Platform

Cloud management platforms are combining software and technologies like virtualization in order to deliver a solution for simpler management of the cloud environment. One of these platforms is an open-source project OpenNebula [15]. OpenNebula eases building and managing virtualized data centers and IaaS cloud models. It allows simple networking, storage, virtualization, monitoring, accounting and multi-tenancy [4]. The platform is build on a monolithic core written in C++ language. Above this core there is the web interface Sunstone together with a set of modules written in Ruby language and the provided stand-alone scheduler written in C++ language (more details follow in Chapter 3.

The cloud architecture in OpenNebula consists of storage, networking and virtualization components. The infrastructure is often composed of many clusters, such infrastructure is represented in the Figure 2.1. Clusters are containing a set of hosts where VMs are deployed. Datastore (DS) provides the necessary storage that serves as a repository for VM's running disks. The whole infrastructure is interconnected by a network. The following sections describe important parts of the OpenNebula's infrastructure in more detail.

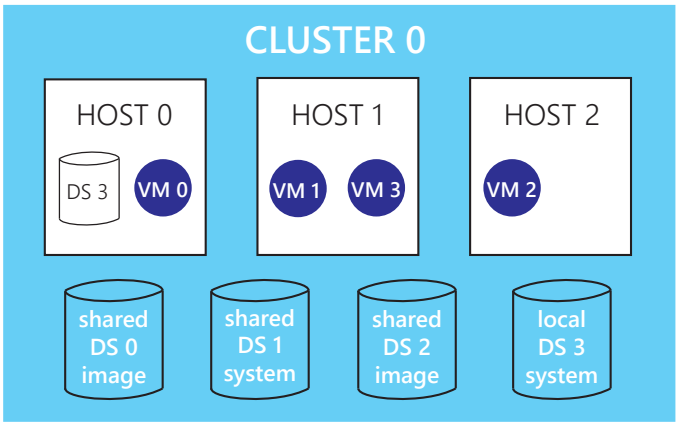

Figure 2.1: Cluster Infrastructure

2.1 Image

Image is necessary for defining VM's disks. The image can be either an *operating system* (OS), *CD-ROM* or *datablock*. Each VM can have only one *CD-ROM* disk and must have at least one *OS* type disk (bootable disk). Images are stored in a Datastore dedicated only to images. More about Image Datastore is explained in the Section 2.4.

Images are either persistent or non-persistent. If the image is persistent, it can be used only by one VM as no copy of that image is made. On the other hand the non-persistent image is copied so they can be used by many VMs.

2.2 Virtual Machine

Virtual Machine is an emulation of a computer system that provides functionalities of a physical computer. In OpenNebula, VM characteristics are defined in a template. VM is created once the template is instantiated. The following attributes define the VM:

- *State* (id) – defines the state of the VM's life-cycle, e.g., `pending` (1), `active` (3), `done` (6), etc.
- *Rescheduling flag* – set on 1 if the VM is considered for rescheduling.
- *CPU* – number of cores requested by the VM.
- *VCPU* – number of virtual cpus.
- *Memory* – amount of requested memory in megabytes.
- *Disks* – multiple disks with the desired storage expressed in megabytes.
- *Histories* – history records of the VM.
- *Scheduling requirements and policies* – defines the scheduling options and is further discussed in the Section 3.1.

The disk section can contain as many disks as the user needs. Following list presents the possible types of disks:

- *Persistent disk* – disk using a persistent image from the Image Datastore.
- *Clone disk* – disk using a non-persistent image from the Image Datastore.
- *Volatile disk* – file-system disk created on-the-fly.

2.3 Host

Host represents the hypervisor enabled resource where the VM can be deployed. Host is defined by the state, e.g., `enabled`, `disabled`, `offline`, `error`, and by the available and allocated *cpu*, *memory* and *storage capacity*. The enabled Host is monitoring the real usage of the resource.

Every Host can have a *reserved cpu* and *memory* capacity. The amount of these reservations is subtracted from the total capacity. The number of the requested reservation can be negative so that the Host can be *overcommited*.

A Host belongs to a Cluster and can have multiple local Datastores assigned. Host also memorizes the number of running VMs and can have PCI devices attached.

2.4 Datastore

The storage space in OpenNebula is represented by Datastores. OpenNebula recognizes three types of Datastores:

- *Image Datastore* – stores disk images. These images are moved, or cloned to the System Datastore whenever the VM is deployed.
- *System Datastore* – holds the images for running VMs. These images are moved, or cloned back to the Image Datastore whenever the VM is shutdown.
- *File and Kernels Datastore* – stores plain files.

Datastore can be either *shared* or *local*. The former means that the available and used capacity can be found directly in the Datastore attributes. The latter belongs locally to a Host, hence the information about used and free megabytes of the storage space is found directly on the Host. The capacity information is available only when the Datastore is in the monitoring state.

Datastore characteristics are determined by the driver that is assigned to each Datastore. The transfer driver configuration is defined in the `oned.conf` file. This file is used to configure OpenNebula and includes definitions of drivers. An example of a driver configuration is shown in the `oned.conf` snippet 2.1.

$$\begin{aligned} TM\_MAD\_CONF = [\\ name = "dummy", \\ ln\_target = "NONE", \\ clone\_target = "SYSTEM", \\ shared = "yes"] \end{aligned} \tag{2.1}$$

The *name* is the name of the Datastore's driver. The parameter *shared* defines whether the Datastore is shared or local. The *ln_target* specifies the target of the image but only when the image is persistent. The *clone_target* specifies the copy location for a non-persistent image. Both of these parameters can be set to one of these options:

- *NONE* – this option meas that the image is not copied, therefore no more space is used in the Datastore location.
- *SELF* – the image is copied to the Image Datastore.
- *SYSTEM* – the image is copied to the System Datastore.

2.5 Cluster

Generally a Cluster is a group of physical resources. In OpenNebula each Cluster in the system has assigned a set of Hosts and Datastores. In order to create a complete environment, every Cluster should have at least one System Datastore assigned. Similarly as the Host, Cluster can also have a *reserved cpu* and *memory* capacity. Thus each Host in this Cluster will subtract or add the given reservation.

2.6 User and Group

OpenNebula is provided with a multi-tenancy feature. A tenant is a group of users sharing resources. User is defined by a name and a password and can belong to more groups. To limit the access to resources, we can assign a quota to a user or a group. With the use of quotas we have a better overview of the resource usage. The following entities are those that can be monitored by using quotas:

- Datastore – limits the number of images and the maximum number of megabytes of Datastore storage that can be used.
- Network – this quota monitors the number of assigned addresses.
- Virtual Machine – defines the limit for cpu, memory, disk size and number of VMs that can be created.
- Image – limits the maximum number of images that the VM can use at the same time.

Each of these quotas can be set individually for each user and group or globally. The global quota is a default option and applies to all users and groups.

2.7 ACL and Permissions

Every infrastructure should have means for controlling the system. The supervision over the infrastructure is achieved by an authorization system. The authorization in OpenNebula is managed by ACL and by permissions.

2.7.1 Permissions

The permission authorization is very similar to the Unix rights management. One permission is defined by three triads. First triad defines what the *owner* can do, second triad defines what the *group* members can do and last triad is for *other* users. Each of these triads have the following rights: *use*, *manage* , or to be an *admin*. The resources with associated permissions are VMs and Images.

2.7.2 ACL Understanding

The system of ACL rules is a complex system that controls the access to resources. For example with an ACL rule it is possible to permit a designated user the right to manage and use some resources. Before an action (like deployment of a VM) is performed these rules are checked. Examples of such rules are given below.

```
#5  HOST/%100  MANAGE  *
```

This rule grants the user with ID 5 to manage the Host on the Cluster with ID 100.

```
@10  DATASTORE/#1  USE
```

This rule grants the group of users with ID 10 to use the Datastore with ID 1.
As we can see the rules split into four parts:

- *ID* – identifies either the user #5, the group – @10, or uses the asterisk sign to designate all users.
- *Resource* – defines one or more resources like Host or Datastore. When more resources are being defined, they are separated by a plus sign. After the forward slash, there can be an ID of the resource. The asterisk sign means that the rule applies to all resources of the specified type.
- *Rights* – determine operations like *manage* or *use* that the user is authorized to perform.
- *Zone*[1] – the ID of a zone where the rule is applied. This part is optional.

1. We can have several instances of OpenNebula. Each of these instances is called a zone. These instances can be configured as one master and several slaves. In other words, a zone is a group of Hosts under the control of a single OpenNebula instance.

3 The OpenNebula's Default Scheduler

The OpenNebula's scheduler is designed to find the best resource for a VM by filtrating and ranking resources by given policies. Policies that the scheduler shall use are configurable. The configuration is defined in the `sched.conf` file.

The scheduler works with data pools grouping elements of one type (Hosts, VMs, Users, Clusters, Datastores and ACL rules). Each scheduling cycle the scheduler takes those VMs that are in the pending state. The next step is to filter unsuitable resources out. Then one suitable Host and Datastore is chosen based on the given policy and they create a match with the VM. OpenNebula follows so called match making algorithm for finding the most suitable resource for the VM.

3.1 The Match Making Algorithm

The idea of the match making algorithm is to firstly filter the non suitable Hosts and Datastores out. The non-suitability means that the resource does not fulfill the VM's requirements. As the next step the scheduler picks the best Host and Datastore based on the given policy. This policy defines some ranking metric, thus resources are prioritized based upon the rank. Hosts are ranked by using one of the placement policies and Datastores have assigned rank based on one of the storage policies.

The match making algorithm follows these steps:

- removes VMs that requires more Image Datastore storage than there is available at the moment;
- filters those Hosts that do not have enough *memory* or *cpu* capacity to host the VM out;
- filters Hosts that do not fulfill the VM's requirements out;
- filters Datastores that do not fulfill the VM's requirements out;
- ranks Hosts based on the given placement policy and chooses the best one;

- ranks Datastores[1] and chooses the best one;
- the chosen Host and Datastore represent the best match for the VM.

3.2 Understanding the Requiremens and Ranks

VM can require a specific resource or a subset of resources. These requirements can be set in VM's template under the scheduling section. The `SCHED_REQUIREMENTS` attribute defines requirements for a Host. The `SCHED_DS_REQUIREMENTS` specifies characteristics of the required Datastore. These attributes are not obligatory. Requirements are boolean expressions and can contain any of resource's parameters. An example of the requirement with Host's `CPUSPEED` attribute:

$$\texttt{SCHED\_REQUIREMENTS} = \text{"}\texttt{CPUSPEED} > 1000\text{"}$$

VM can specify a ranking policy. These policies are also chosen in the scheduling section of the VM's template. The placement policy is set under the `SCHED_RANK` attribute. The `SCHED_DS_RANK` attribute defines the storage policy. Ranks are arithmetic expressions composed of resource's attributes and numbers in a form of an equation. An example of such equation is shown below. The rank for the Host will be calculated based on the result of the equation.

$$\texttt{SCHED\_RANK} = \text{"}\texttt{RUNNING\_VMS} * 50 + \texttt{FREE\_CPU}\text{"}$$

OpenNebula provides three following predefined placement policies.

- *Packing* – minimizes the number of Clusters in use by placing the VM on a Host with more VMs running first.
  `SCHED_RANK = RUNNING_VMS`
- *Striping* – maximizes the nodes available to a VM by using the Host with less VMs running first.
  `SCHED_RANK = -RUNNING_VMS`

1. only those Datastores that are in the same Cluster as the chosen Host

- *Load Aware* – maximizes the nodes available to a VM by choosing the Host with less load first.
  `SCHED_RANK = FREE_CPU`

The following list presents options for the storage policies:

- *Packing* – chooses the Datastore with less free space to optimize the storage usage.
  `SCHED_DS_RANK = -FREE_MB`

- *Striping* – chooses the Datastore with more free space in order to balance the I/O load.
  `SCHED_DS_RANK = FREE_MB`

The policies can be defined either by setting the mentioned attributes or by configuring them globally in the `sched.conf` file. With the latter option, defined policies applies to all VMs.

3.3 Summary

The match making algorithm has many strong characteristics like the fact that it is possible to use any attribute for configuring requirements and ranks. To support that many options, the scheduler uses a parser generated by Bison[2]. However the supported policies are only applied to resource prioritization, thus the scheduler does not support any prioritization of users or groups. Therefore VMs are being processed in a FIFO fashion. This solution is not appropriate in regards of fair-sharing of resources among users. Furthermore, the scheduler does not support dynamic migrations. Hence some of the resources might be overloaded whilst some of them underloaded. Migrations would dynamically balance the infrastructure by a defined criteria.

Another shortcoming is that it fixes the best Host during the Host ranking process. The problem with this approach is that the optimal solution might not be found. When the Host is fixed, we are limiting ourself to one Host and Datastores in the according Cluster. We might want to prefer to chose the best storage rather than the best Host. The OpenNebula's default scheduler is lacking that possibility.

2. general-purpose parser generator: `https://www.gnu.org/software/bison/`

4 Our Approach – ONEScheduler

This chapter proposes the design and implementation details of our scheduler — ONEScheduler. In the previous chapter, the OpenNebula's scheduler was described. We also presented its shortcomings. The mentioned drawbacks can be easily fixed by implementing and incorporating new policies. Hence, we were motivated to implement a new scheduler, that would be possible to extend with new features. ONEScheduler is also configurable via a separated configuration file.

First, we propose the design of our approach, the package structure and technical solutions. Then we describe the configuration mechanism. Finally, each of the provided features are explained in details.

4.1 Design of ONEScheduler

This section focuses on the architecture of the newly proposed scheduler. ONEScheduler is designed to be easily extensible and to have its features separated into several modules. The architecture also has to consider that the scheduler will be used in the testing mode. This mode is provided for testing scheduler's features with a set of workloads. Thus, the origin of the data needs to be hidden behind an interface.

When designing a new piece of software, design patterns [17] are usually applied. The strategy pattern is used for incorporating new algorithms. Each policy is implemented as a plugin with defined interfaces. The scheduler has the data origin hidden behind interfaces, thus the abstract factory pattern applied. The scheduler sees only the interface, not the specific implementation. The singleton pattern was applied to auxiliary classes, as they do not need to be instantiated.

The overall idea of ONEScheduler is inspired by the match making algorithm. We took advantage of this algorithm in order to preserve the OpenNebula features, like resource filtration and criteria based resource ordering. The source code of the scheduler provided by OpenNebula is available on GitHub. Hence, the work on our scheduler started by studying the code of the OpenNebula's scheduler. The design of ONEScheduler was drawn once it was clear which parts of the scheduler will be separated to create a more logical structure, hence more readable code.

4.1.1 Top Level Structure

Figure 4.1 outlines the top level structure of ONEScheduler. Scheduler is configurable via separated configuration files. The configuration is located in the `configFiles` package. The testing mode has necessary XML documents located in the `pools` package. The `configFiles` package also contains `tmMadConf.xml` file, where the transfer manager drivers configuration is defined. Why is the drivers configuration needed is explained in the Section 2.4. The `jar` file represents the OpenNebula provided OCA. It contains necessary methods for communication with OpenNebula.

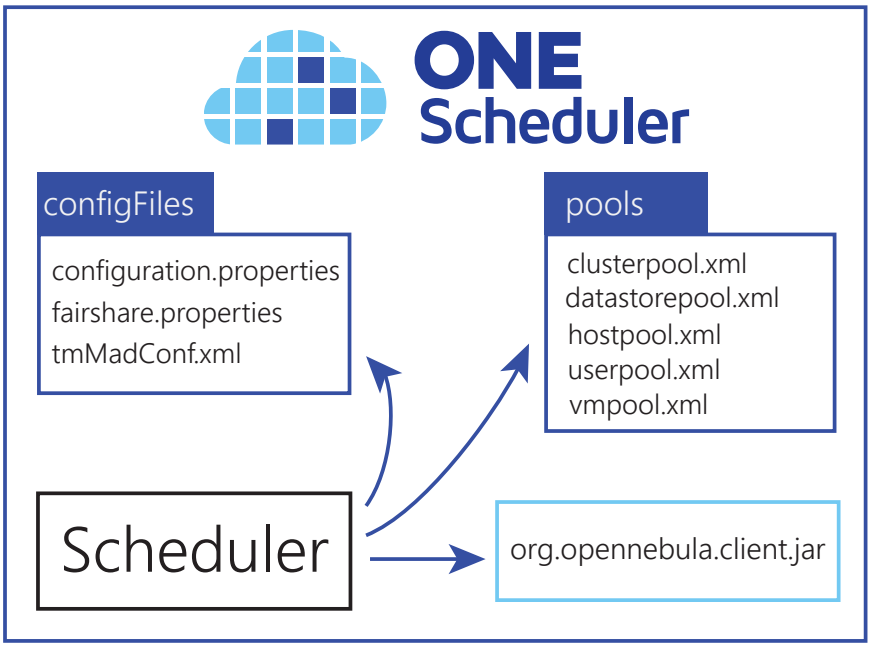

Figure 4.1: Top Level Structure

4.1.2 Package Structure

The package structure is represented in Figure 4.2. The main package `scheduler` groups scheduler's features into several packages. Each package represents a module handling one part of the scheduler's logic. How these modules have been designed is discussed in the Section 4.6.

The `xml` package contains the mechanism for loading necessary data for the testing mode. The implementation is further explained in the Section 4.4. The `one` package is used for loading the data pools from OpenNebula. This part is further discussed in the Section 4.3.

Classes in the `config` package instantiates pools, filters and policies. The checked `LoadingFailedException` exception is thrown if the instantiation of pools failed.

The `result` package contains classes managing the deployment of VMs. This part is further explained in the Section 4.7.

Finally the `extensions` package comprises of classes that has all their methods static. These methods are used very often and it was appropriate to exclude them into a separate package.

4.2 Technical Solutions

At the beginning of the implementation process, we were discussing two options for the language use — Ruby and Java. Developers of OpenNebula would prefer the Ruby language. However we chose the Java language because it is known to be faster performing language than Ruby. Thus we decided to build the scheduler in Java 8. Besides using the standard Java platform, we used Spring [18] framework for handling dependency injection[1], Jackson [19] library and XPath for parsing XML documents, and SLF4J API[2] for logging.

The development of the new scheduler was eased by the fact that OpenNebula provides API for the communication and data retrieval. We used the Java OpenNebula Cloud API (OCA). This API consists of methods designed as wrappers for XML-RPC [16] methods[3]. For using the API, it is necessary to be familiar with the XML representation of entities in OpenNebula. In this work we are using the version 5.2 of OpenNebula [8].

First of all, we had to decide how we will be parsing incoming XML documents. The scheduler is designed to have the option for

1. The dependency is some object (service) and injection is the process of passing of the dependency into a dependent object (client).
2. Simple Logging Facade for Java, `http://www.slf4j.org/`
3. XML-RPC is a protocol that is used for remote procedure call. Data are in a form of an XML and are transported by the HTTP protocol.

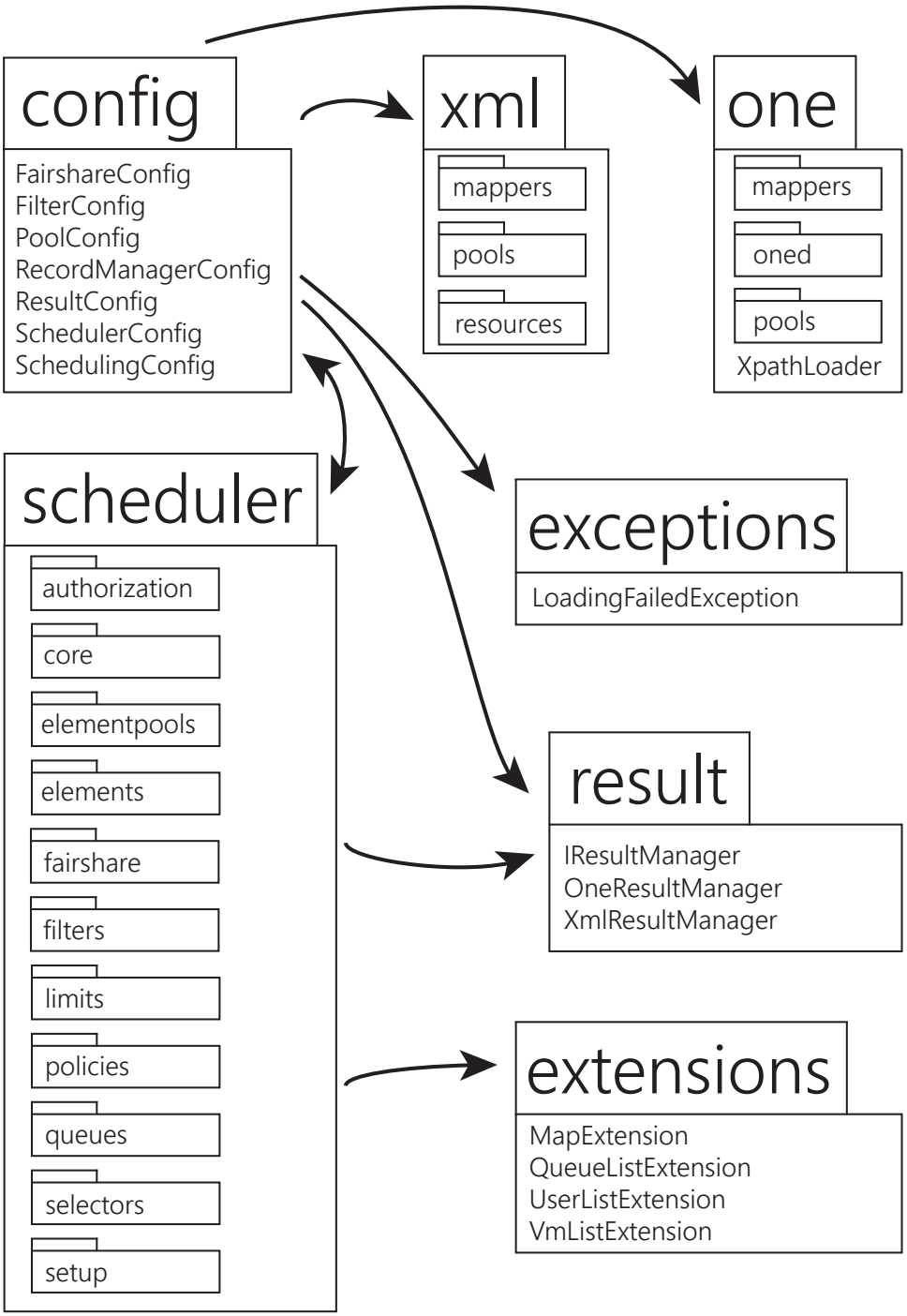

Figure 4.2: Package Structure

switching into a testing mode. In this mode, we parse data pools with the help of Jackson library. The package `xml` handles parsing of provided data sets. In the mode, when the scheduler communicates with OpenNebula, we use OCA's methods for parsing data pools coming from OpenNebula. These methods are based on the XPath technology. When using XPath, it is necessary to specify the path to the desired attribute in the XML document. Each XML document represents one pool grouping elements of one type. These elements have a fixed structure defined by XSD (XML Schema Definition). With the knowledge of the schema, we are able to specify the path to the element's attribute and retrieve the value. The package `one` handles parsing of incoming data pools from OpenNebula.

With the two possible mode options, it was necessary to provide interfaces that would enable switching between those options. Data pools are presented to the scheduler through the following interfaces — `IAclPool`, `IClusterPool`, `IDatastorePool`, `IGroupPool`, `IHostPool`, `IUserPool`, `IVmPool`. Thus, the scheduler is unaware of where data have their origin. Both modules contain the necessary mappers that map each object from the XML document to the following elements — ClusterElement, DatastoreElement, HostElement, UserElement and VmElement, further referenced only as *Element* or *Elements*. The scheduler then works with these Elements.

4.3 OpenNebula Client

This section explains details of the data retrieval from OpenNebula. The package `one` contains the mechanism for fetching and storing incoming data pools. In order to fetch pools, like ClusterPool, DatastorePool, HostPool, UserPool and VirtualMachinePool, it is necessary to create an instance of the `Client` class. This class is provided by OCA. The instance of the `Client` class is passed to the constructor of pools.

Each object in the pool is mapped to an according Element. OCA offers the `xpath()` method for each pool element. An attribute or a node is retrieved by specifying the path in the XML document. A node is a complex object, that an element in the XML document can contain. An example of a node is the disk node in the VM element. As

we know, one VM can have multiple disks, so each disk is represented as one node.

Elements have some of their attributes obligatory and some optional. In order to ease the mapping, the `XPathLoader` class was created. The static methods in this class help with the retrieval of an attribute or a node, if it exists.

4.4 Testing Mode

In order to test the scheduler's features, we have provided a testing mode. This testing mode can be further developed into a simulator. Running OpenNebula means an unnecessary overhead. Therefore we have developed a way to emulate the communication with OpenNebula. The emulation is reached by creating XML documents representing necessary entities. Each of those elements is defined by a template that has a given structure defined by XSD.

This section describes details of the data retrieval for the testing mode. The testing mode uses Jackson library mechanisms for parsing XML documents. Jackson is a Java library for processing data formats like JSON and XML. We take the advantage of the library and used it for deserializing POJOs from XML.

The `xml` package contains the mechanism for the deserialization. The `XmlMapper`'s `readValue()` method implements that mechanism by using Jackson. In order to read the document, the method needs the XML representation of one pool in a form of a `String` and the class to which the document will be parsed. The XML file is then processed and the list of elements is created. The procedure of creating POJOs by using Jackson is annotation driven.

In order to create the list of elements defined in the XML file, it needs to have the annotation with the name of the root tag. Hence each of these lists should be annotated as follows.The `localName` would be, in our case, `CLUSTERPOOL`.

```
@JacksonXmlRootElement(localName = "XML_ROOT_TAG")
```

Each attribute of the element needs to have `JacksonXmlProperty` annotation in order to be recognized by the parsing mechanism in Jackson library. An example of such annotation is given below. The `local-`

`Name` corresponds to the XML tag of the attribute. A tag `<ID>` would have the corresponding `localName=ID`. Once the process of deserialization is ended, obtained POJOs needs to be mapped to Elements.

4.5 Configuration

ONEScheduler has two possible modes, the mode that communicates with OpenNebula, and the testing mode. The scheduler is also designed to have the possibility to incorporate new algorithms. Hence, it was necessary to introduce a configuration mechanism.

4.5.1 Configuration System

The configuration was separated from the source code into a file. This solution enhances the comfort of setting parameters, like used policies, for the scheduling process. The configuration file is a simple property file with keys and values. The `configuration.properties` file is located in the `configFiles` package. The following example represents a snippet, setting the filters, from the configuration.

`hostFilters` = `FilterHostByCpu`, `FilterHostByMemory`

Loading of the file handles the JRE[4]-provided `Properties` class. The class `PropertiesConfig` parses the configuration file. This class contains basic methods that are used for getting the property value. The value can be one of the following types: `String`, `Array` of `Strings`, `Integer` and `Float`. In order to obtain the value of the parameter, it is necessary to call the according getter method on an instance of the `PropertiesConfig` class. The getter needs the name of the configuration parameter key.

If ONEScheduler is in the testing mode, the path to data sets needs to be set in the configuration file. The following list represent necessary configurable parameters:

- `hostpoolpath`
- `clusterpoolpath`

4. Java Runtime Environment

- `userpoolpath`
- `vmpoolpath`
- `datastorepoolpath`

The connection to OpenNebula needs these parameters:

- `secret` – should contain the OpenNebula authentication tuple, user:password, e.g., oneadmin:opennebula
- `endpoint` – should contain the url where the RPC server is listening, e.g., `http://localhost:2633/RPC2`

4.5.2 Dependency Injection

The `config` package handles the instantiation of pools, filters and policies that are specified in the configuration. There are many dependencies among classes, therefore it was appropriate to use some framework that helps with dependency injection. We are using the Spring IoC container that is responsible for instantiating classes and injecting dependencies into them [20].

The Spring container is driven by annotations. Every class is annotated by `@Configuration` , indicating that these classes contain definitions of one or more `@Bean` methods. These beans can be then injected into classes that are dependent on the particular bean. To inject some instance, as an attribute into another object, the `@Autowired` annotation is used. The following classes are provided `@Configuration` classes.

- `FairshareConfig` – instantiation depends on the selected policy in `fairshare.properties`.
- `FilterConfig` – creates instances of filters. Filters are defined in the `configuration.properties` and we distinguishe filters for fairshare, Hosts and Datastores.
- `PoolConfig` – configures data pools.
- `RecordManagerConfig` – creates files fair-share-related files.
- `ResultConfig` – instantiates the result manager.

- `SchedulerConfig` – creates the instance of the `Scheduler` class and instances of the authorization manager and chosen placement and storage policies.

- `SchedulingConfig` – instantiates the corresponding implementation to `IQueueMapper`, `IVmSelector` and `ILimitChecker` .

In order to use the defined beans, it is necessary to create an instance of the Spring's `AnnotationConfigApplicationContext`. This class is one of the implementation of the `ApplicationContext` interface. The `getBean()` method returns the bean instance of the given type.

4.6 Modules

A module represents one separated package that groups classes with same goals, like fair-sharing, filtration or authorization. The package `scheduler` consists of these modules.

Figure 4.3 shows the package organization of provided modules, the blue ones represent the data. The gray ones are features that OpenNebula's match making algorithm includes. Packages `fair-share`, `queues`, `selectors` and `limits` represent new features that ONEScheduler have introduced.

The `core` package contains the essential `Scheduler` class where all the necessary steps to create a plan are taken. The `SetUp` class in the `setup` package contains the application main method. The instance of the `Scheduler` is created in the main method. The `schedule()` method is called on the created instance of the `Scheduler` class and the scheduling process starts.

4.6.1 Core

The `core` package contains essentials classes, like `Scheduler`, `Match` or `RankPair`. This section describes these classes.

The `Match` class has been created to represent the Host and Datastore that were matched to VMs. Every time a match is found, the VM is either added to the list of VMs to an already created Match, or the Match is created and the VM is added as the first one.

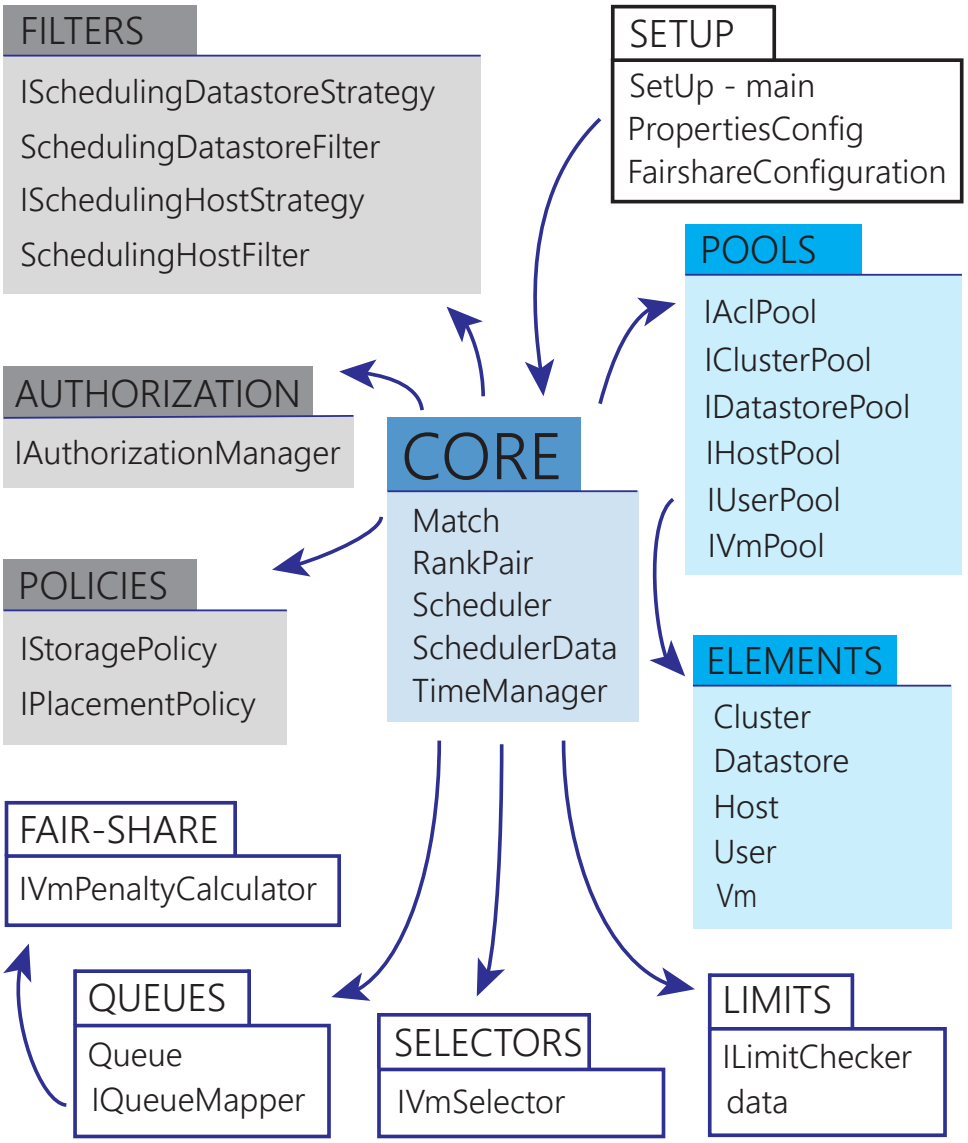

Figure 4.3: Modules

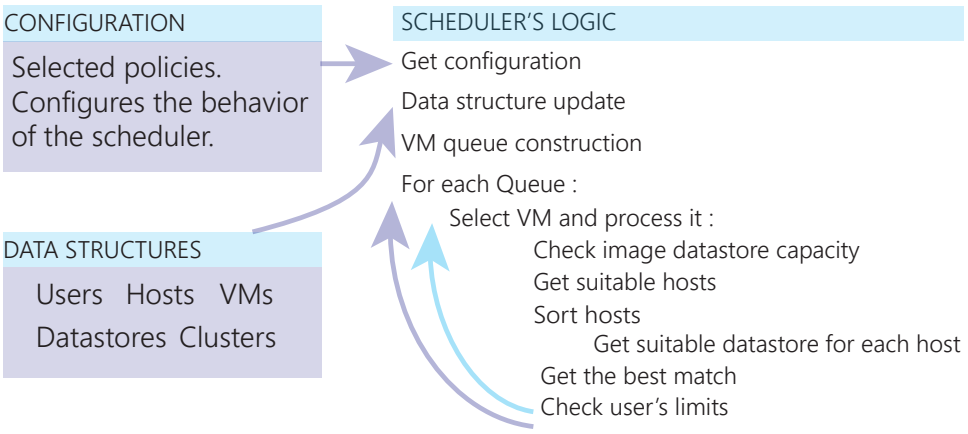

Figure 4.4: Scheduler Loop Logic

The `RankPair` class is used when Datastores are ranked, based on the selected storage policy. Therefore the `RankPair`'s attributes are Datastore and an Integer representing the assigned rank. The `RankPair` have been introduced to have clearly defined assigned ranks for Datastores.

The core class of ONEScheduler is the `Scheduler` class. It does all the necessary steps in order to produce a plan. The final plan fulfills certain policies that were given in the configuration file. Figure 4.4 represents the logic of the scheduler's steps. First, the scheduler needs to read the configuration, then it updates data structures. Subsequently, pending VMs are put into queues. The order in which these VMs are selected from the queue is defined by the specified policy. Resources are filtered and the VM is matched with the most suitable Host and Datastore. The suitability depends on the chosen scheduling policy.

4.6.2 Authorization

The authorization manager is needed to determine whether a user is authorized to use the resource. If the user is authorized, there is an ACL rule permitting him to access the resource.

Resources that are checked before scheduling are Hosts and Datastores. How ACL rules looks like in OpenNebula is described in the Section 2.7. Rules that influence the scheduling are those that

have the *manage* right for Host or a subset of Hosts and *use* right for Datastore or more Datastores.

We created the interface `IAuthorizationManager` in order to determine authorized resources. The manager parses ACL rules and assigns to each user those resources that the user is authorized to use. The following methods are provided by the interface.

- `authorize()` – sets the subset of Hosts and Datastores that the user can access.
- `getAuthorizedHosts()` – returns the list of authorized Hosts for the VM.
- `getAuthorizedDs()` – returns the list of authorized System Datastores for the VM.

The ACL authorization applies only when using OpenNebula. If the scheduler works in testing mode, no ACL rules are created. However this feature can be easily added by implementing the logic into `AuthorizationManagerXml`.

4.6.3 Filters

The filtering is one of the core ideas of the OpenNebula's match making algorithm. ONEScheduler implements all the necessary filtration introduced in OpenNebula. On top of that, our approach proposes an extensible version of the original filtration mechanism.

ONEScheduler distinguishes between filtering Hosts and Datastores. These filters have many strategies and they are separated in the package `filters`. Each of the strategies has to implement given interface. The `ISchedulingHostFilterStrategy` is an interface for Host policies. The Datastore policy needs to implement the `ISchedulingDatastoreFilterStrategy`. Both interfaces contains a single method `test()` which returns true, if the test passed, and false otherwise. These methods needs to know which VM is being currently scheduled, the candidate Host and also the Datastore (if we are implementing a Datastore strategy).

`SchedulerData` class

When testing the VM against some criteria defined by the filter, it is necessary to pass current data. However ONEScheduler does not deploy every time a Match for a VM is found in order to reduce the number of I/O actions. Thus, the `SchedulerData` class have been created to store the reserved cpu, memory, and running virtual machines for each Host and free storage space for each Datastore. This class works as a reservation system. It is necessary to update these attributes every time a Match is created.

Host Filtration

Before choosing the Host for VM, it is necessary to filter the unsuitable Hosts out. They are ruled out based on the memory and cpu capacity. Each VM demands certain amount of Host's memory and cpu. If the Host does not have enough capacity to host the VM, it is not considered as suitable.

The implementation of memory and cpu filter is in separated classes called `FilterHostByMemory` and `FilterHostByCpu`. These filters are essential for the correct deployment, so they should be configured in the `configuration.properties`.

We also provide other two optional filters. The `FilterHostByPci` that handles correct matching of Peripheral Component Interconnect (PCI) nodes required by VM and provided by the Host. This filter should not be used if the system is not configured to be using PCIs.

The `FilterHostBySchedulingRequirements` handles requirements of the VM. In the Section 3.2 we described how a VM can have defined requirements. The filter parses the VM's scheduling requirements. If the Host, that is currently being tested, matches the VM requirement, the test returns true.

The strategies are set in the configuration file. The `test()` method of these strategies is called in the `SchedulingHostFilter` class. This class takes the list of strategies configured in the configuration file and applies the test. If all the tests passed for the Host, then the Host is suitable for the VM.

By Host filtration we are obtaining a subset of Hosts where the VM can be deployed. The scheduling continues with Datastore filtration.

Datastore Filtration

Each suitable Host has assigned some storage capacity, either locally or has some shared Datastores. Disks of the VM are stored in these Datastores. Datastores are filtered based on their free storage. The total size of VM's disks is calculated and is checked against the System Datastore free capacity.

The filtration of Datastores based on their free capacity is implemented in the `FilterDatastoreByStorage` policy. As it was explained in the Section 2.4, the Datastore storage capacity value depends on whether the Datastore is shared or local. We need to know if the Host and the Datastore belong to the same Cluster. If the Datastore is in the same Cluster as the Host, the Datastore might be suitable. Then if the Datastore is shared, the capacity can be checked directly on the Datastore. If the Datastore is local, the capacity is checked on the Host's local Datastore nodes.

The Section 2.4 mentions, that for determining, whether the Datastore is shared or not, we need to retrieve the transfer manager configuration in the `oned.conf`. This part of the configuration needs to be copied to the `tmMadCong.xml` located under `configFiles` directory.

The next provided filter handles the VM's Datastore scheduling requirement. Hence, the `FilterDatastoreBySchedulingRequirements` tests whether the VM requests a specific Datastore. If there is such requirement, the test checks if the current Datastore is the one.

Similarly to the Host filtration, the `SchedulingDatastoreFilter` takes the list of configured storage policies. Each filter tests the Datastore. All tests passing means that the Datastore is suitable for the VM. At the end of the Datastore filtration, we obtain a list of suitable Datastores for each Host.

4.6.4 Policies

Policies are used for criteria-based Host ordering. The criteria is expressed by the policy. Policies available in ONEScheduler are inspired by the resources ranking in OpenNebula. We described these policies in the Section 3.2.

Usually, the policy depends on the current situation of resources, thus to each method that the interface defines, it is necessary to provide the `SchedulerData` instance.

Host (Placement) Policies

The `IPlacementPolicy` is an interface that each placement policy should implement. The `IPlacementPolicy` defines one method `sortHosts()` that takes already filtered Hosts and returns a list of sorted Hosts based on the selected policy. We provide `LoadAware`, `Packing` , and `Striping` strategies.

Datastore (Storage) Policies

The storage policy should implement the `IStoragePolicy`. The interace declares `selectDatastore()` method that selects the best ranked Datastore for each Host and returns the `RankPair`. This object encapsulates the best ranked Datastore along with its assigned rank. The `RankPairs` that are obtained for each Host can be then compared based on their rank. The `getBestRankedDatastore()` method gets the Datastore that fits the most the selected criteria. We provide `StorageStriping` and `StoragePacking` strategies.

4.6.5 Filtration and Policies Workflow

Figure 4.5 shows the workflow of getting *candidates* . First steps like, filtration and resource sorting based on the given policy were described in previous sections. Generally, a candidate for a VM is a suitable Host with a suitable Datastore. Candidates are represented as a `Map` containing Hosts and RankPairs. Hosts are keys in this `Map` and are ordered by the placement policy. Thus on the first place in this map, there is the most suitable Host for the VM. Every key has assigned value, RankPair, defining the best Datastore and its rank.

The candidates map was created in order to give the option to choose either the best Host or the best Datastore. The suitability depends on placement and storage policies. For example, if the placement policy was set on *Packing*, it would mean that the first Host in the map would be the one with the most VMs running. If the storage policy

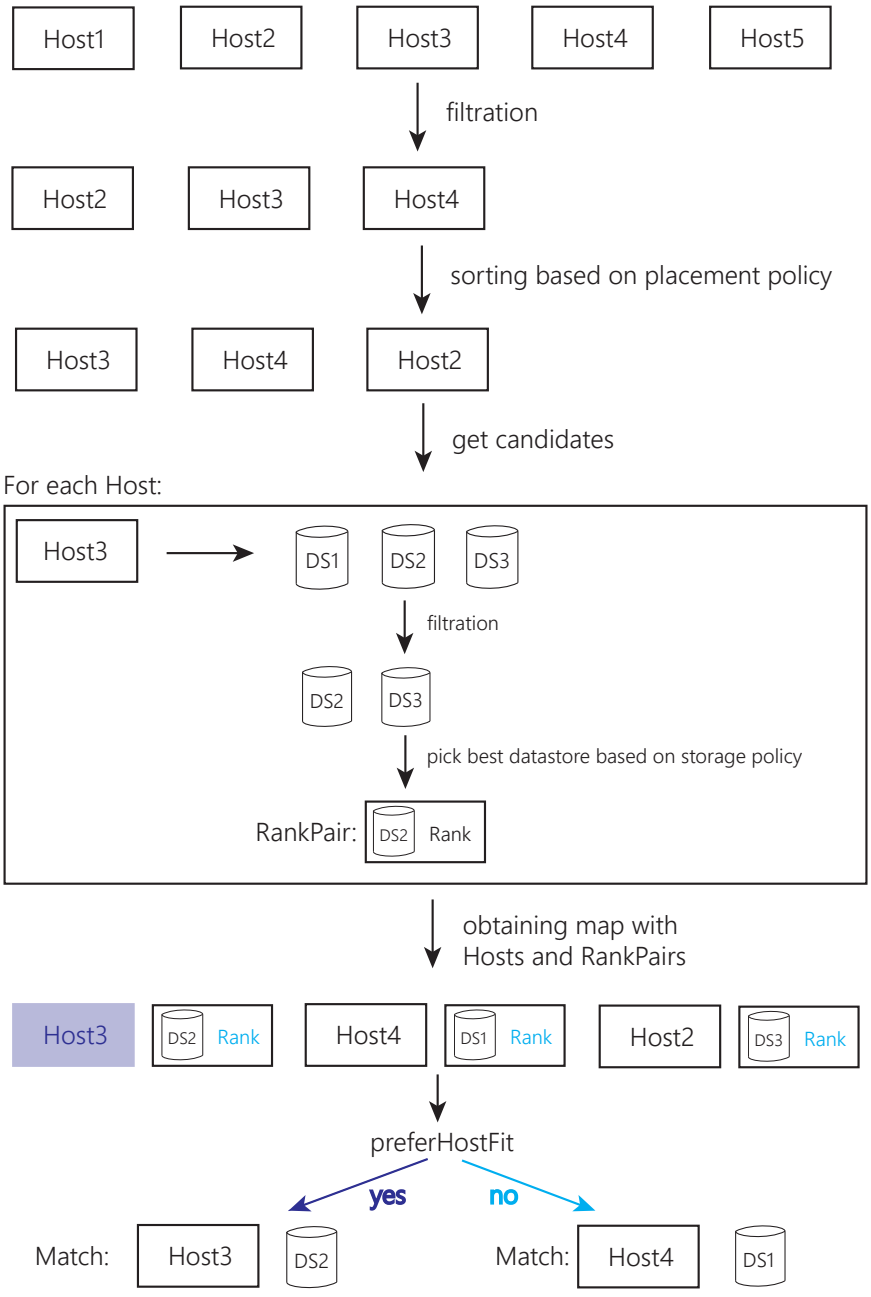

Figure 4.5: Getting candidates

would be set also on the *Packing* policy, it would choose for each Host the Datastore with less free space available.

Then there are two options for choosing the Match from these candidates. One is to prefer the best Host fit. The second is to prefer the best Datastore fit. These options are represented in the last step in Figure 4.5.

4.6.6 Fair-share

The goal of fair-sharing is to assign VMs in a fair way. The fairness depends on the fair-sharing technique we use. Each technique implements an algorithm that assigns a priority number to a user or a group. The fair-sharing is applied in the `Queue` mapping implementations discussed in the next section.

4.6.7 Queues

Queue based models are introduced when VMs can be grouped by some criteria. We designed our `Queue` class as a list of VMs, assigned priority, and a name. The list of VMs is represented as a `ConcurrentLinkedQueue` that contains VMs. Queue priorities are the same, if the fair-sharing policy is not set. Hence, the assigned priority is based on the chosen fair-sharing policy.

The `Queue` class contains `dequeue()` method represented on the Figure 4.7. This method is returning and at the same time removing the first VM in the Queue. Our `Queue` implementation also provides the `queue` method shown on figure 4.6. This method adds VM at the end of the Queue.

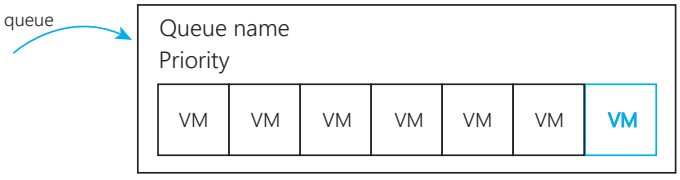

Figure 4.6: Queue

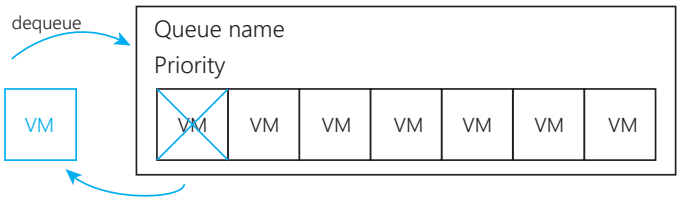

Figure 4.7: Dequeue

Queues are created based on the chosen `IQueueMapper` implementation in the configuration file. The interface has one method `mapQueues()` which takes the list of VMs and maps them into queues. The following list presents proposed implementations:

- `OneQueueMapper` – this implementation is the easiest one. It creates one queue that is filled by VMs in the FIFO fashion.
- `FixedNumOfQueuesMapper` – this class creates the number of queues given in the configuration file and puts all VMs into those queues in FIFO fashion.
- `QueueByUserMapper` – creates queue for every user and puts their VMs into according queue. The priority in this mapper is not set.
- `UserFairshareMapper` – this mapper adds to the previous implementation the priority of the user. The priority is assigned to the queue and the queues are sorted by that number.
- `UserGroupFairshareMapper` – handles VMs in a same way as the previous implementation but it creates queues for groups. The queue belonging to one group has the VMs in it ordered by the priority of the user.

4.6.8 VM Selection

The order in which VMs in queues are iterated defines the implementation of the `IVmSelector` interface. The one method declared, the `selectVm()` method, chooses the VM from queues. Which VM is

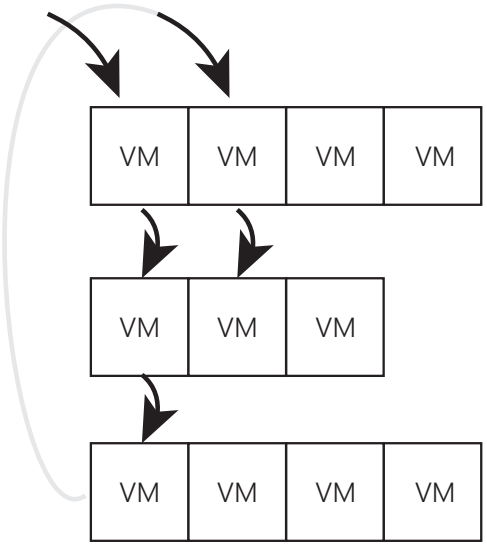

Figure 4.8: Round Robin-like algorithm

selected depends on the implementation. We provide `RoundRobin` and `QueueByQueue` implementations.

Round Robin-like algorithm

Figure 4.8 demonstrates how the Round Robin algorithm goes through queues. The algorithm remembers the index of the current queue and changes it for the next iteration. The algorithm starts at the first queue, calls dequeue to get the VM and increases the index of the current queue by one. The next iteration, the VM is dequeued from the second queue. When the index reaches the last queue, it resets its value to zero to point to the first queue again. The VM selections goes until all the queues are empty.

Queue by Queue algorithm

The `QueueByQueue` implementation is selecting VM by VM in one queue. When the current queue is empty, it continues with the next queue. Figure 4.9 demonstrates how this algorithm is cycling through

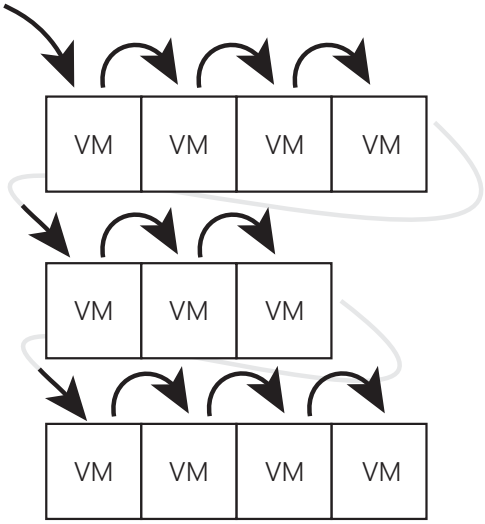

Figure 4.9: Queue by Queue algorithm

all queues. It always dequeues the first VM in the first Queue until all queues are empty.

4.6.9 Limit Checking

Users or groups can have assigned limits. These limits can control, for example, the resource usage or number of assigned VMs. These limits needs to be checked before the VM is matched with the chosen Host and Datastore.

This feature provides the `ILimitChecker` interface. It contains two methods: `checkLimit()` and `getDataInstance()`. The former takes the VM and its Match and returns whether the VM's user does not exceed his limits. If true is returned, the VM can be deployed on the Host. The latter method returns the data instance. This instance is needed for storing the current limit-related data, that the implemented limit checker controls.

4.7 Writing the Results

Once the solution — plan — is found, the results needs to be written. ONEScheduler is handling the result writing by offering the `IResultManager` interface with two defined methods. The first is the `deployPlan()` method that deploys the VM. It takes the passed plan, deploys VMs and returns the list of VMs where the deployment failed. The failure can happen, especially when using the scheduler with OpenNebula, the deploy does not need to be successful. So when an exception in OpenNebula occurs, the method remembers the failed VM. The second provided method is `migrate()`. This method migrates those VMs that have the rescheduling flag.

`OneResultManager` deploys the VM using the method `deploy()` and migrates the VM by `migrate()` , these methods are provided by OCA. The `XmlResultManager` is ready to have the simulation module attached in the future.

5 Evaluation

The evaluation of ONEScheduler was accomplished by running several experiments. The purpose of experiments is to see how the scheduler is performing with the increasing workload (number of VMs) and/or size of the infrastructure. The performance of ONEScheduler was analyzed by tracking the runtime and memory usage. The goal was to see if the runtime grows with the increasing number of VMs and Hosts and if the memory consumptions is peaking in any parts of the scheduling process.

For the purpose of experiments, we defined what is a *small*, *medium* and *big* Host.

- *small Host* – 2 cpu, 1 GB of maximum RAM and 512 GB of available storage
- *medium Host* – 8 cpu, 4 GB of maximum RAM and 1024 GB of available storage
- *big Host* – 64 cpu, 128 GB of maximum RAM and 4096 GB of available storage

Also we define what is a *small*, *medium* and *big* VM.

- *small VM* – 0.25 cpu, 512 MB RAM and 512 MB of requested storage
- *medium VM* – 1 cpu, 2 GB RAM and 4 GB of requested storage
- *big VM* – 4 cpu, 4 GB RAM and 16 GB of requested storage

Each runtime experiment is composed of several tests. Each test was repeated 10 times in order to obtain the average required runtime[1] and the corresponding standard deviation.

1. Value of the first repetition was omitted to prevent including the overhead of starting the application into results.

5.1 Experiments

All of the tests were measured using the same configuration file. The used policy for Host and Datastore was *Striping*. The filtration includes only the necessary memory, cpu and storage filters. The Queue mapping was omitted and we used only one FIFO Queue. For experiments we also did not use any fair-sharing algorithms. We wanted to incorporate only those features that the standard OpenNebula's scheduler is providing. All of the measurements were run on a PC equipped with Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, 8 GB RAM, running Windows 10.

5.1.1 Heterogeneous Experiment

The infrastructure for the heterogeneous experiment consists of real data[2], with the actual capacity availability, from the MetaCloud[3]. For the purpose of this experiment, we defined possible values for VM's cpu, memory and storage capacity. The workload for this experiment was generated by creating VMs combining the possibilities below. The following list presents the attribute with according set of possible values.

- cpu = $\{0.25; 0.5; 1\}$
- memory = $\{512MB; 1024MB; 2048MB\}$
- disk size = $\{512MB; 2048096MB; 4096MB\}$

Figure 5.1 shows results for the heterogeneous experiment. Corresponding standard deviations are shown in Table B.1. The biggest growth is registered with 1024 VMs and 512 VMs. However the growth is not increasing that quickly. The reason is that the heterogeneous infrastructure has Hosts with real available storage space. Hosts have enough cpu and memory capacity to host the VM, however the storage might not have enough free space to store VM's disks. Thus, VMs are only limited to several Hosts with enough storage space.

2. http://metavo.metacentrum.cz/pbsmon2/cloud/
3. MetaCloud is a service provided by the Czech National Grid Infrastructure MetaCentrum, `https://www.metacentrum.cz/cs/`

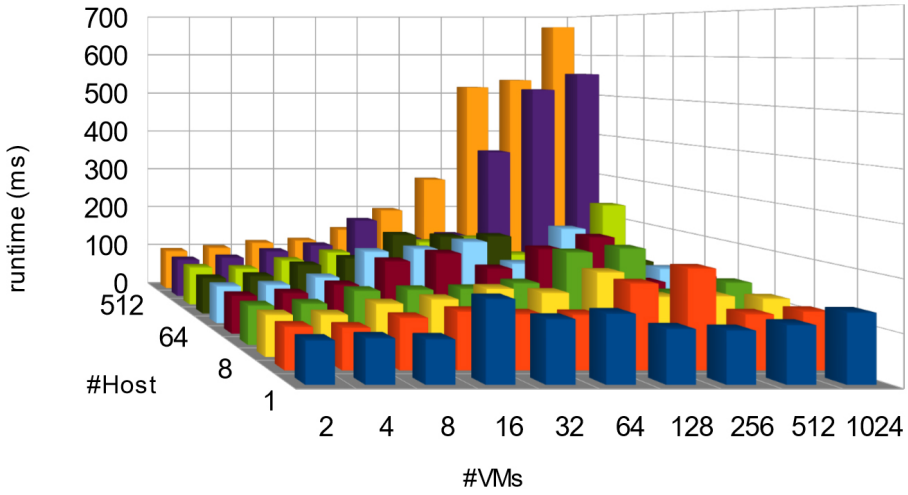

Figure 5.1: Heterogeneous Data

5.1.2 Homogeneous Experiment

We divide this experiment[4] into three experiments.

- The first was measured with small VMs and medium Hosts.
- The second was measured with medium VMs and medium Hosts.
- The third was measured with big VMs and medium Hosts.

The first experiment is represented in Figure 5.2 with the corresponding standard deviations in the Table B.2. In this case, VMs were matched in most times. Thus, apart from the growth with 1024 and 512 VMs, runtime values stays almost the same.

The second experiment can be seen in Figure 5.3 with the corresponding standard deviations in the Table B.3. This graph shows how the value of the runtime rapidly increased with 512 or 1024 VMs on 256 or 512 number of Hosts.

4. The data sets used in this experiment can be found on `https://github.com/GabiP/ONEScheduler/tree/experiments`

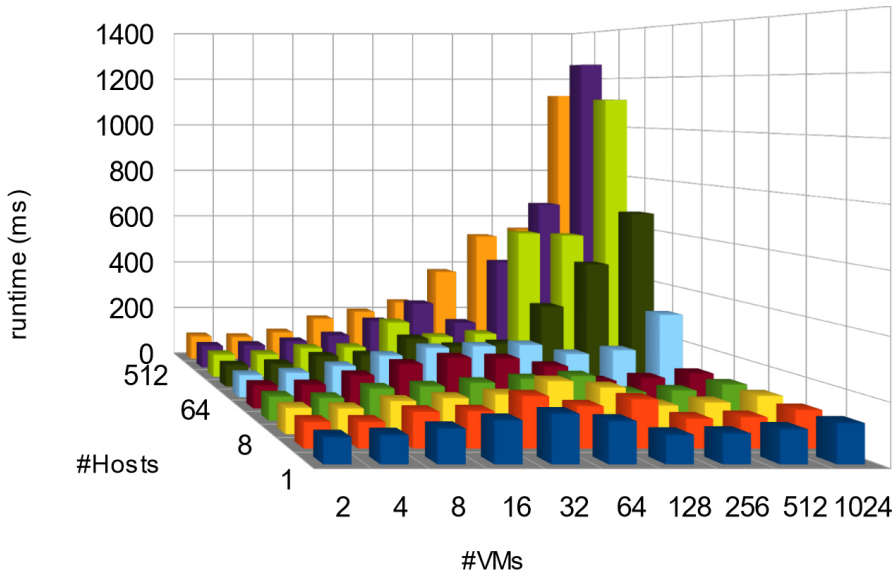

Figure 5.2: Homogeneous Data (Small VMs and Medium Hosts)

Figure 5.4 represents results of the third experiment. The corresponding standard deviations are in the Table B.4. The values with big workload and small infrastructure has only few of the VMs matched. In most cases VMs were not assigned to Hosts. Therefore the values of runtime stayed with the big workload almost the same as with the smaller workload.

5.1.3 Memory Usage Experiment

The memory usage was tracked using the built-in profiler in Netbeans IDE. We used workload with 256 and 1024 VMs with 8 cpu, 16 GB RAM and 32 GB of the required disk size. For 256 VMs we had 128 available Hosts. For 1024 VMs we had 512 Hosts. Each Host had 64 cpu, 128 GB of RAM and 32 TB of available storage.

Figure 5.5 shows results with 256 VMs with 128 Hosts with logging turned on. The highest allocated memory was 120 MB.

Figure 5.6 shows results for 1024 VMs with 512 Hosts with logging turned on are more interesting. The highest allocated memory was

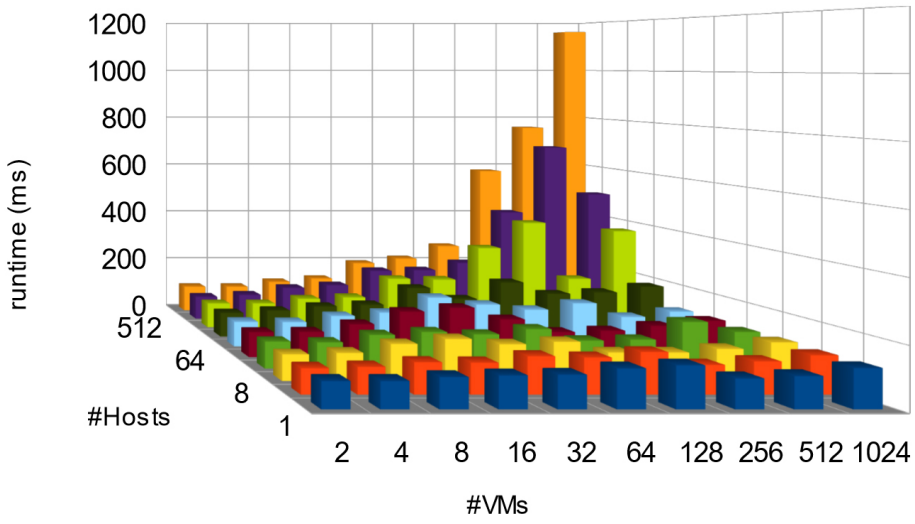

Figure 5.3: Homogeneous Data (Medium VMs and Medium Hosts)

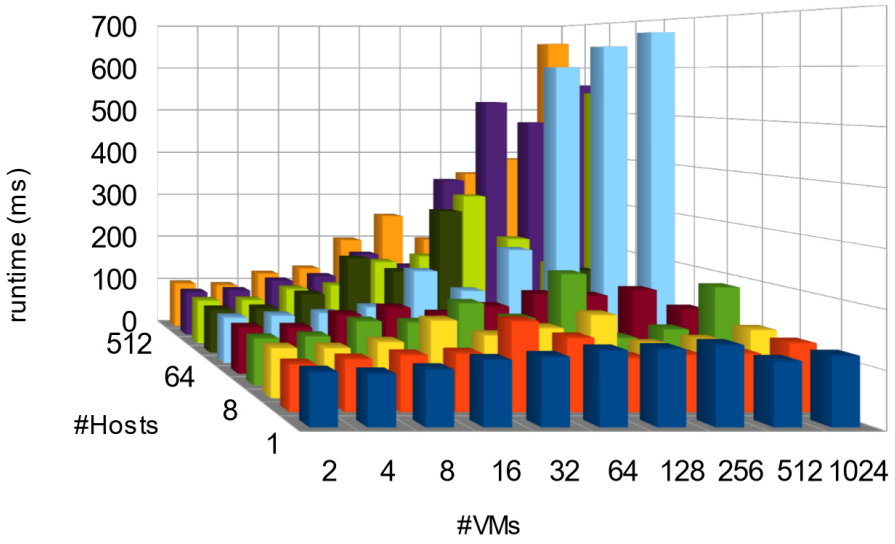

Figure 5.4: Homogeneous Data (Big VMs and Medium Hosts)

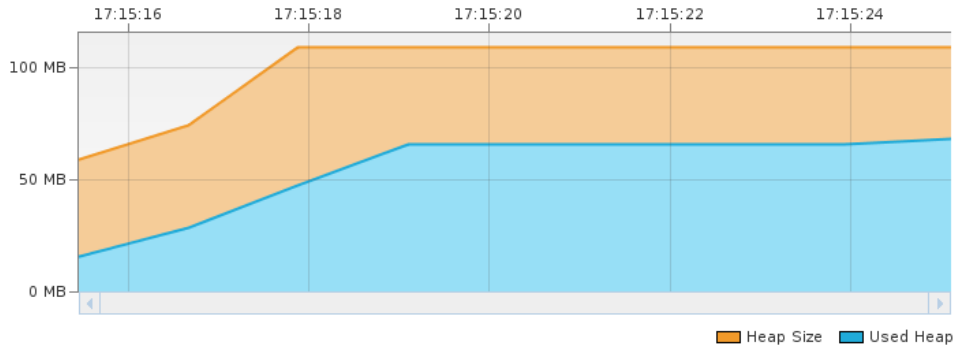

Figure 5.5: Used memory with 256 VMs, 128 Hosts (with logging on)

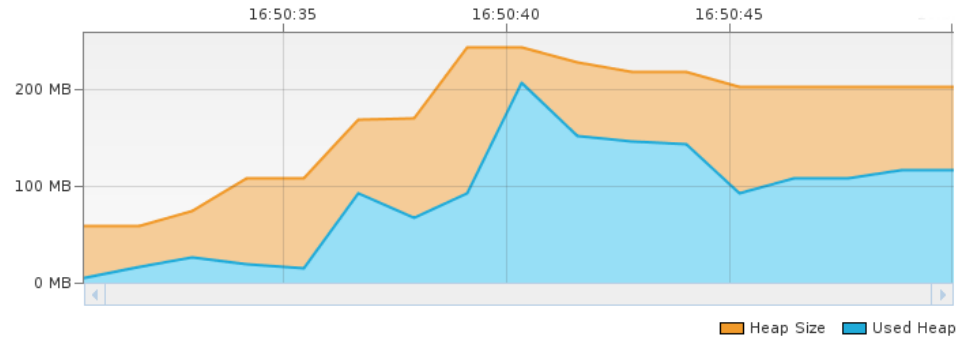

Figure 5.6: Used memory with 1024 VMs, 512 Hosts (with logging on)

220 MB. The memory usage is peaking the most where the scheduler is logging the most, like in the filtration process. The increasing infrastructure ads more possibilities where the VM can be deployed, thus the logging in filters adds some overhead.

Figure 5.7 represents results with the logging turned off with 256 VMs and 128 Hosts. In this case the allocated memory was never higher than 75 MB.

Results for 1024 VMs and 512 Hosts with logging turned off are represented in Figure 5.8. In this case the allocated memory was never higher than 120 MB.

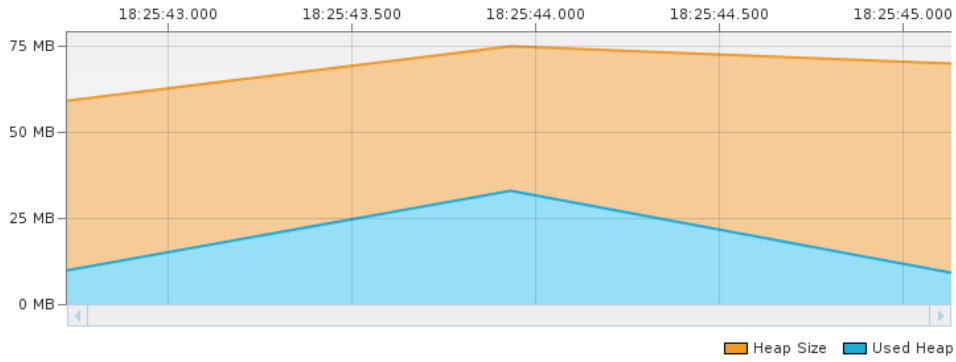

Figure 5.7: Used memory with 256 VMs, 128 Hosts (with logging off)

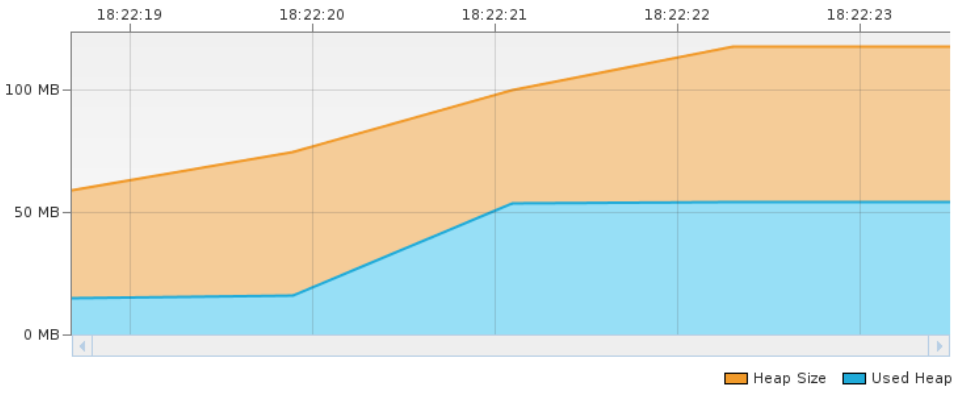

Figure 5.8: Used memory with 1024 VMs, 512 Hosts (with logging off)

With the given workload and infrastructure, the memory never steps over the 250 MB limit. We have run the memory usage experiment to see also how the scheduler performs with and without logging. As we expected, the logging adds an overhead.

6 Conclusion

The goal of this thesis was to design new extensible scheduler for cloud management platform OpenNebula. The first part of this work consisted of defining the concept of cloud computing and why a scheduler is an important component of a cloud management system. Next chapter described the OpenNebula's infrastructure. Subsequently, the OpenNebula's default scheduler was presented. Following chapters focused on the design and implementation of the proposed scheduler.

Our approach, ONEScheduler, was inspired by the scheduling component in OpenNebula. We successfully implemented the filtration of resources and criteria based resource ordering features that the OpenNebula's scheduler provides. ONEScheduler is composed of many modules, each handling one part of the scheduling process like authorization, fair-sharing, queue mapping, filtrating and ranking. Some of these modules contains new features, like queue based scheduling and fair-sharing of resources. The goal was to provide interface to each module. These interfaces are used for incorporating a new scheduling policies. Furthermore, ONEScheduler can be switched into testing mode in order to test new policies.

For added convenience, we also made ONEScheduler configurable. The configuration is in a separated file and is used for defining scheduling policies.

The last part of this work was dedicated to experiments, testing the ONEScheduler's performance. We were tracking the scheduler's runtime and memory usage. Tests were executed in the testing mode with heterogeneous and homogeneous data sets. Results were as we expected, with 1024 VMs, the runtime was quickly growing. When testing the memory consumption of the scheduler, we discovered that logging adds an overhead to the memory usage.

To sum up, ONEScheduler is capable of producing a schedule that fulfills given criteria. The schedule consists of pending VMs mapped onto resources. If the scheduler has the connection to OpenNebula, the scheduler can deploy these VMs onto planned resources in OpenNebula.

For future work, the development can further go towards automatic migrations. The scheduler will have the ability to migrate the already running VM if there is a more suitable resource available. ONEScheduler is also currently being extended by the development of a simulator called ONESimulator. This simulator will emulate the environment of OpenNebula. ONESimulator can be attached to ONEScheduler through a provided interface.

Bibliography

[1] Kai Hwang, Jack Dongarra, and Geoffrey C. Fox. *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[2] Peter M. Mell and Timothy Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, United States, 2011.

[3] Wenhong Tian and Yong Zhao, eds. *Optimized Cloud Resource Management and Scheduling*. Boston: Morgan Kaufmann, 2015.

[4] *OpenNebula, industry standard open source cloud computing tool.* 2016. URL: http://opennebula.org/ (visited on 12/01/2016).

[5] *OpenStack*. 2016. URL: https://www.openstack.org (visited on 12/01/2016).

[6] *Apache CloudStack*. 2016. URL: https://cloudstack.apache.org/ (visited on 12/01/2016).

[7] Maciej Drozdowski. *Scheduling for Parallel Processing*. 1st. Springer Publishing Company, Incorporated, 2009.

[8] *OpenNebula Documentation, version 5.2*. 2016. URL: http://docs.opennebula.org/5.2/ (visited on 12/01/2016).

[9] Flavien Quesnel. *Scheduling of Large-scale Virtualized Infrastructures: Toward Cooperative Management*. 1st. Wiley-IEEE Press, 2014.

[10] Joel J. P. C. Rodrigues Chun-Wei Tsai. "Metaheuristic Scheduling for Cloud: A Survey". In: 8.1 (2014), pp. 279–290.

[11] Borja Sotomayor. *Haizea, The Haizea Manual*. 2009. URL: http://haizea.cs.uchicago.edu/ (visited on 12/01/2016).

[12] Borja Sotomayor et al. "Virtual Infrastructure Management in Private and Hybrid Clouds". In: *IEEE Internet Computing* 13.5 (Sept. 2009), pp. 14–22.

[13] *Green Cloud Scheduler*. 2012. URL: http://coned.utcluj.ro/GreenCloudScheduler/ (visited on 12/01/2016).

[14] T. Cioara et al. "Energy Aware Dynamic Resource Consolidation Algorithm for Virtualized Service Centers Based on Reinforcement Learning". In: *2011 10th International Symposium on Parallel and Distributed Computing*. 2011, pp. 163–169.

[15] Ruben S. Montero, Rafael Moreno-Vozmediano, and Ignacio M. Llorente. "IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures". In: *Computer* 45.undefined (2012), pp. 65–72.

[16] Frederic Magoules, Thi-Mai-Huong Nguyen, and Lei Yu. *Grid Resource Management: Towards Virtual and Services Compliant Grid Computing*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 2008.

[17] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[18] *Spring framework*. 2016. URL: `https://spring.io/l` (visited on 12/01/2016).

[19] *Jackson GitHub repository*. 2016. URL: `https://github.com/FasterXML/jackson-dataformat-xml` (visited on 12/01/2016).

[20] *Spring IoC container*. 2016. URL: `https://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html` (visited on 12/01/2016).

A ONEScheduler Tutorial

This tutorial explains how to use the ONEScheduler[1] either in testing mode or with OpenNebula. When trying the testing mode, the parameter `testingMode` in `configuration.properties` needs to be set on true. Next, it is necessary to set paths to the directory, where your testing files are. These paths are set on default `pools` package with an example set of pool XML files. We also provided more extended files in `onePoolXml` package. These files represent real host, cluster and datastore pools from Metacloud. The VM pool and user pool is generated in order to test large amount of pending VMs.

In order to use ONEScheduler with OpenNebula the secret and endpoint parameters must be configured to match the OpenNebula authentication credentials. If you are using VirtualBox with an OpenNebula appliance, the default secret should be oneadmin:opennebula and the endpoint http://localhost:2633/RPC2. Then you should kill the OpenNebula scheduling daemon and run the ONEScheduler application.

A.1 Extending ONEScheduler

Let's assume that you want to extend the policies to sort Hosts by some class introducing a new strategy, let's call it `NewPlacementPolicy`. Then this class needs to implement the `IPlacementPolicy`, therefore the `sortHosts()` method will apply the new strategy.

In order to use this policy, the bean definition of the placement policy in the `SchedulerConfig` class needs to be extended as well. This bean returns the instance of the placement policy that was set in the `configuration.properties`. The bean contains a switch for the possible policies. To incorporate the `NewPlacementPolicy` just add a new switch case and set the policy in the `configuration.properties`.

1. ONEScheduler is available on `https://github.com/GabiP/ONEScheduler`.

B Tables of Standard Deviations

	1	2	4	8	16	32	64	128	256	512
2	13	14	15	14	19	19	18	17	17	17
4	18	16	113	19	24	13	15	15	16	17
8	20	43	55	85	63	51	95	59	73	40
16	145	86	80	96	133	131	116	100	105	86
32	102	69	95	91	174	153	173	120	151	131
64	131	63	128	128	138	176	189	163	135	171
128	33	123	121	183	174	137	177	186	155	228
256	26	184	47	183	177	173	25	25	215	232
512	24	29	32	24	37	43	25	22	411	424
1024	21	29	29	41	27	23	22	192	237	512

Table B.1: Standard Deviations for Heterogeneous Data

	1	2	4	8	16	32	64	128	256	512
2	17	15	18	19	17	17	23	18	25	24
4	20	20	14	20	17	20	20	24	23	20
8	85	83	74	78	59	80	84	79	49	53
16	98	101	104	112	131	131	101	83	107	124
32	138	150	131	136	174	172	171	172	171	159
64	132	136	173	173	181	184	178	188	235	203
128	34	189	191	180	159	199	154	164	174	255
256	28	24	29	21	23	162	229	276	268	298
512	18	20	17	25	30	181	354	340	376	340
1024	19	19	17	21	19	262	511	453	334	195

Table B.2: Standard Deviations for Homogeneous Data (Small VMs on Medium Hosts)

	1	2	4	8	16	32	64	128	256	512
2	10	17	19	21	16	13	17	21	22	20
4	19	15	20	14	26	20	26	25	16	25
8	36	43	86	69	78	68	77	77	74	70
16	56	41	98	104	135	104	88	83	101	90
32	64	106	103	89	150	105	175	165	170	160
64	99	87	128	132	131	184	133	172	176	181
128	128	136	29	24	18	135	195	226	199	209
256	21	22	31	21	24	156	153	231	254	205
512	20	27	22	161	22	30	160	148	398	160
1024	22	16	19	16	19	28	169	260	253	545

Table B.3: Standard Deviations for Homogeneous Data (Medium VMs on Medium Hosts)

	1	2	4	8	16	32	64	128	256	512
2	10	17	19	21	16	13	17	21	22	20
4	19	15	20	14	26	20	26	25	16	25
8	36	43	86	69	78	68	77	77	74	70
16	56	41	98	104	135	104	88	83	101	90
32	64	106	103	89	150	105	175	165	170	160
64	99	87	128	132	131	184	133	172	176	181
128	128	136	29	24	18	135	195	226	199	209
256	21	22	31	21	24	156	153	231	254	205
512	20	27	22	161	22	30	160	148	398	160
1024	22	16	19	16	19	28	169	260	253	545

Table B.4: Standard Deviations for Homogeneous Data (Big VMs on Medium Hosts)