University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Further Improvement and Evaluation of an OpenStack Extension to Enforce Cloud-wide Multi-Resource Fairness during VM Runtime

Bogdan Veres
Zürich, Switzerland
Student ID: 14-710-669

Supervisor: Patrick Poullie, Dr. Thomas Bocek
Date of Submission: May 10, 2017

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Managing large amounts of physical resources in clouds or data centers is not an easy task and efficiency plays a decisive role in the performance of instances created by the clients. The initial scheduling of the resources of a Virtual Machine (VM) is not always enough to ensure a fair distribution of physical resources between the users.
This thesis improves an OpenStack service that managed cloud physical resources and defined fairness through a heaviness metric which uses resource consumption information of all instances to determine the heaviness of each user. Based on a number of metrics, the cloud resources are reallocated at run-time to achieve a fair resource distribution among the users. This thesis also describes, classifies and addresses the noisy neighbor problem with the help of the fairness service. Another important topic is the impact of the virtual resources scheduling in the physical resource distribution. The last part of the thesis is formed by a number of tests that asses the performance of the newly created framework in an OpenStack environment. Overall the new service proved to work accordingly in different use-cases, reallocating resources between users and VMs in real time. The new service has also improved the communication between the nodes and is decoupled from the OpenStack services.

Acknowledgments

I want to thank my advisors Patrick Poullie and Dr. Thomas Bocek for their help and support given throughout the thesis. A special thanks to Patrick Poullie for trying to help with all problems and obstacles that emerged during the thesis, and also for beeing available anytime ready to answer all questions.

Contents

Abstract **i**

Acknowledgments **iii**

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Outline 3

2 Related Work **5**

3 The Noisy Neighbor Problem **7**

3.1 Problem Definition 7

3.2 Types 8

3.3 Finding And Addressing The Problem 8

3.4 Dealing With The Problem 8

4 The New Fairness Service **11**

4.1 Multi-resource Fairness and User Heaviness 11

4.2 Measurements Description 12

4.3 The Fairness Framework 13

5 Improving Message Exchange **17**

5.1 Theoretical Insights For Reducing The Message Volume 17

5.2 Implementation Details 18

5.3 Integration With The Fairness Framework 19

6 VCPUs And CPU Time Allocation **21**

7 Evaluation **27**

8 Future Work **31**

8.1 VR Counterparts 31

8.2 Imporving The Message Exchange 32

8.3 Other Improvements 32

9 Conclusions **33**

Bibliography **35**

Abbreviations **39**

List of Figures **39**

A Installation Guidelines **43**

A.1 Prerequisites 43

A.2 Controller Node 43

A.3 Compute Node 44

Chapter 1

Introduction

These days there are many discussions everywhere about clouds, cloud computing, cloud storage and many more. This area has evolved rapidly in the last years, and got adopted by numerous industries and domain fields. A cloud platform provides virtualized scalable computing resources, which are shared across multiple users over the internet, in a technically and administratively scalable manner [1]. Many enterprises obtained huge benefits by using the pay per use cloud outsourcing model, where they saved money not only on the initial investment, but also afterwards with the maintenance of software and hardware resources [3]. This also results in a high end-user adoption in a large number of companies not limited to the IT sector [2].

1.1 Motivation

Usually, in a cloud environment, users are offered a multi-tenant architecture, where applications of many different customers run together in a shared infrastructure [4]. There are three types of cloud environments: The first category is private clouds, where enterprises build their own hardware infrastructure and keep the cloud in a private network, accessible only for them. why break?
The second category is the public/commercial clouds, where physical resources location is not exposed to the clients and everyone has access to it (Examples are: Amazon Elastic Compute Cloud, IBM Blue Clouds or Windows Azure). The last category is represented by the hybrid clouds. This means that the clients keep a private smaller cloud in-house, for regular, normal usage, and, whenever a spike appears in utilization, resources will be allocated in a public cloud to accommodate the sudden needs.

The most widespread use case for a cloud user is to start Virtual Machines (VMs), and then execute its applications or jobs inside the respective VMs. Consequently, the VMs of different users will use the physical resources of the same cloud nodes (CPU time, RAM, disk I/O and network). This also means that VMs of different users will compete with other VMs for physical resources on the same host. If these resources become congested,

their performance will decrease a lot, without the user being aware of the reason for the performance decrease [5].

In order to make sure resources are properly divided between the users, some considerable effort has been put recently into data center resources allocation. Resource allocation is complicated by itself, but in clouds it gets even more complicated due to a number of factors:

- There are many physical hosts that could be spread in different locations and could have different performance
- More recently, VMs can use resources from more than a single host
- Users also have quotas and different permissions which are also taken into consideration
- Since users are not aware of the underlying physical resources, the performance of VMs should reflect as much as possible the defined virtual resources [6, 7, 8]

In the papers [10] and [9] there is shortly described the resource allocation process through two steps which are conducted continuously and in parallel: "In the first step called VM scheduling, the cloud's orchestration layer decides which VM is started next and which node hosts the VM. This step statically multiplexes nodes and is conducted for every VM that is started or live migrated. During the Runtime Prioritization step, a node's hypervisor, i.e., the operating system of a node, allocates the node's physical resources, such as CPU time, RAM, disk I/O, and network access, to the VMs. This step statically multiplexes physical resources of individual nodes and is conducted permanently".

The scheduling step means deciding which node will host a particular VM, while the second step means deciding how many actual physical resources will be allocated. The time-shared resources like CPU, disk I/O and network can be efficiently allocated/reallocated by priorities, while the space-shared resources like RAM or disk space will bring a bit of overhead.
Taking this into consideration means that the second step allows to be flexible when changing the performance of VMs, even if the scheduling happens of different node types [9]. Even so, the VM scheduling is still taken more often into consideration, which leads to inefficient resource management over long periods of time [10]. Many clouds also use VM migration in order to improve the performance, but that comes at a very high cost in terms of overhead.

The initial scheduling of the resources of a VM is not enough to ensure a fair distribution of physical resources between the users. In other words, most of the current approaches are insufficient to ensure multi-resource fairness during runtime, or just consider only a few resources (e.g.: just CPU) [11].

In order to close this gap, the thesis [9] created a nova-fairness service (referred to as Fairness Service) by extending the OpenStack nova services. The new service achieves

fairness among users by gathering constantly information about the usage of VMs and the physical resources used, aggregating this information to the heaviness of users and reallocate physical resources to adapt to the VM new priorities. When the heaviness of users is aligned, all users will have access to a fair share of the physical resources.

The Communications Systems Group (CSG) defined the Greediness Metric (GM), which was proven to result in fair allocations and give incentive to user to configure the Virtual Resources (VR) of their VMs according to the VMs subsequently load, when the GM is deployed as heaviness metric for the FS.

One important example of functionality is the following: If two users X and Y deploy VMs on two hosts A and B, and one physical resource (e.g., CPU, RAM, network access, disk I/O) becomes congested on host A (could also be B). The fairness service ensures that the congested resource is allocated in favor of the user, who puts less stress on the cloud.

This means that even if user X runs more VMs on host A, a larger share of the congested physical resources is allocated to his VMs, if his overall resource utilization (this includes utilization information from host B) is small compared to user Y. Without the Fairness Service, the congested physical resources will be allocated to the VMs of the two users by some standard scheduling algorithm that only considers local/host information unaware to which user the VMs belong to.

Even if the Fairness Service was implemented in [9] and it was proved to achieve the desired functionality through some brief tests, there are many more details left open. Therefore, this Master Thesis address some of these problems, especially the architecture details of the implementation.

1.2 Thesis Outline

In Chapter 2, a few papers that analyze resources distribution in virtualized environments have been discussed, as well as the first version of the fairness framework (referred also as fairness service). Chapter 3 goes deeper in analyzing a common issue in today's clouds, the noisy neighbor problem, and how can the current implementation of the framework improve such situations. Chapter 4 describes at a detailed level the new fairness framework implementation and differences compared to the old version. Chapter 5 explains the communication part of the same new implementation. In Chapter 6 we test OpenStack and the new framework in a simple scenario to check how the number of declared VCPUs affect VM performance in a resource contention situation. In Chapter 7 we test a few use-cases to prove that the framework works accordingly with the new implementation. Chapter 8 describes future steps that should be implemented, in order to improve the fairness framework. The final conclusions are described in Chapter 9.

Chapter 2

Related Work

There are three main approaches that take into consideration multi-resource fairness problems in cloud environments. The first solution is called Bottleneck-based Fairness (BBF) [12, 8, 13, 14], where it is assumed that, with a single bottleneck, there exist other devices which are not fully utilized. As a result, the allocation problem becomes here a scheduling problem and it's similar for a system with multiple bottlenecks [12].

The second approach and also the most heavily used is called Data Resource Fairness (DRF) [15], and is basically a generalization of the max-min fairness to multiple resource types. The max-min fairness maximizes the minimum resources received by a user and, assuming all users have enough demand, each of them will get an equal share. This approach was generalized my adding the concept of weight and sharing the resources proportional to the dominant resource. On short, DRF tries to maximize the minimum dominant share across all users [15].

The third approach is an extension to Proportional Fairness [16], where two classes of rate control algorithm for communication networks are generalized to include routing control and also provide proportional fair pricing implementations. A solution where resources are prioritized during runtime should allow a better management and the ability to adapt to changing needs. There are two papers [17, 18] that study priority functions needed in order to schedule jobs, which could be used also for VM prioritization. The paper [19] also includes runtime prioritization on data centers and allows users to prioritize their own VMs.

As mentioned in the introduction, the paper [9] introduced a fairness service that proved to work accordingly and achieve the desired functionality. However, there were a few important issues that needed to be addressed. From architectural point of view, the nova-fairness is an OpenStack nova service, strongly coupled with the rest of the OpenStack environment. This means that all components are also managed by OpenStack, together with the communication between them.
This strong coupling makes further development very difficult, as well as maintaining the solution. In order to bind the nova-fairness with the rest of the nova services, a lot of overhead code had to be created, which is strongly connected to the structure of the OpenStack. Every time a new version of OpenStack is released, (usually every six months) the

code must be adapted and possible drastically changed to still be compatible with the other nova services.
Because of the increased code complexity, the classes often contained the so called antipattern ”bowl of spaghetti”. This means that part of the code contained very long methods that had tangled code with parts of the OpenStack nova code. The communication between different components of the service is done again with structures from other nova packages, while the communication between nodes is managed both by OpenStack and a third party Redis server [20] with ZeroMq [21] messaging queue.

Because of the difficulty of further extending the existing code, a new approach was taken, to create a new service completely decoupled from OpenStack. Many parts from the initial Fairness Service were still usable, such as the numerous metrics used to map or calculate resources, and also gathering information/reallocation calls to the Libvirt [22] library.

Chapter 3

The Noisy Neighbor Problem

Cloud infrastructures today provide virtualized scalable computing resources, shared across the internet. This means that multiple independent instances of applications operate in a shared environment (or multi-tenant environment) [1]. The hypervisor virtualizes the physical resources such that the instances have the illusion of exclusive access to the given resources [1].

3.1 Problem Definition

To have an efficient environment and to increase resource utilization, resource overbooking proved to be a very good solution [27]. Usually, the tenant applications are competing for the real hardware resources. The effect is even more powerful in case of overbooked systems since there is even a higher number of VMs asking for the same underlying resources. This solution lead to one of the biggest challenges in clouds, the performance unpredictability because of the resources contention [1].

The Noisy neighbor problem refers to a co-tenant that monopolizes the CPU, memory, disk, network or other resources, thus having a negative impact to the performance of other co-tenants. It has been proven that it can occur up to 80% performance loss just because of a collocated instance/VM [26]. Since the performance difference is so high, many data centers allocate significant human resources to investigate this problem.

One interesting example in Amazon EC2 environment where noisy neighbors are not addressed by default, is the Netflix Chaos Monkey service, which closes broken instances in Amazon EC2, or even badly performing ones [23]. So, whenever a VM performance drops significantly due to a noisy neighbor, the service will automatically kill the respective instance and create a new one.

3.2 Types

The types of noisy neighbor differ, depending on the location of the respective co-tenant: in case of the same physical machine (intra node level) the resources consumed could the CPU cores, CPU time, CPU cache, Memory, Disk Storage, Disk speed, disk access, Network bandwidth and others.
If the noisy neighbor is located on a different machine (inter node level), it can still impact negatively the network bandwidth or disk bandwidth in the case of a network attached storage. Since most hypervisors focus on allocating CPU time and memory capacity [26], it is critical that also disk I/O and network bandwidth get to be properly allocated and managed.

3.3 Finding And Addressing The Problem

In order to address the problem, the first step would be to identify if there is a noisy neighbor, in what conditions an instance becomes noisy, and also to constantly monitor the performance for the identification of new noisy neighbors. In the paper [1], there are mentioned two approaches: monitoring the cloud applications and a heuristic based approach.

The monitoring involves capturing metrics such as Memory usage and read/write errors (R/W errors), network bandwidth consumption, storage usage, I/O operations and errors. Since the hypervisor shares the real resources to the VMs, the ”noisy neighbor” problem appears when an instance must wait in turn for the physical resources. Whenever some pre-defined limits are reached for a VM, a noisy neighbor is identified.

Benchmarking the VM provides also metric results that can be compared with other VMs of the same type. In case the performance of two instances of the same type of VMs and on very similar nodes is very different, the presence of a noisy neighbor can be detected.

In large environments that run over large amounts of time continuously, the heuristics based approach uses historical data to predict that a particular application will become too greedy in the future [1].

3.4 Dealing With The Problem

In the same paper [1] there are described three classical ways of dealing with the noisy problem: the first refers to cloning/migrating to another node of the cloud, the second to borrowing resources from other nodes and the third approach refers to a way of defining quotas for the resource allocation.

The first solution will clone/migrate the victim VM on another node (Figure 3.1), based on performance, physical proximity or resource utilization. By moving the victim, the

overhead is smaller, but still present, especially in the network when you have to transfer large volumes of data (in GB).

There are also approaches for live VM migration, mainly based on which phase is carried out first: if first the memory is migrated to the destination node, and then the rest of the migration (applications) we have a pre-copy live migration. If the phases are reversed we have a post-copy live migration [1].

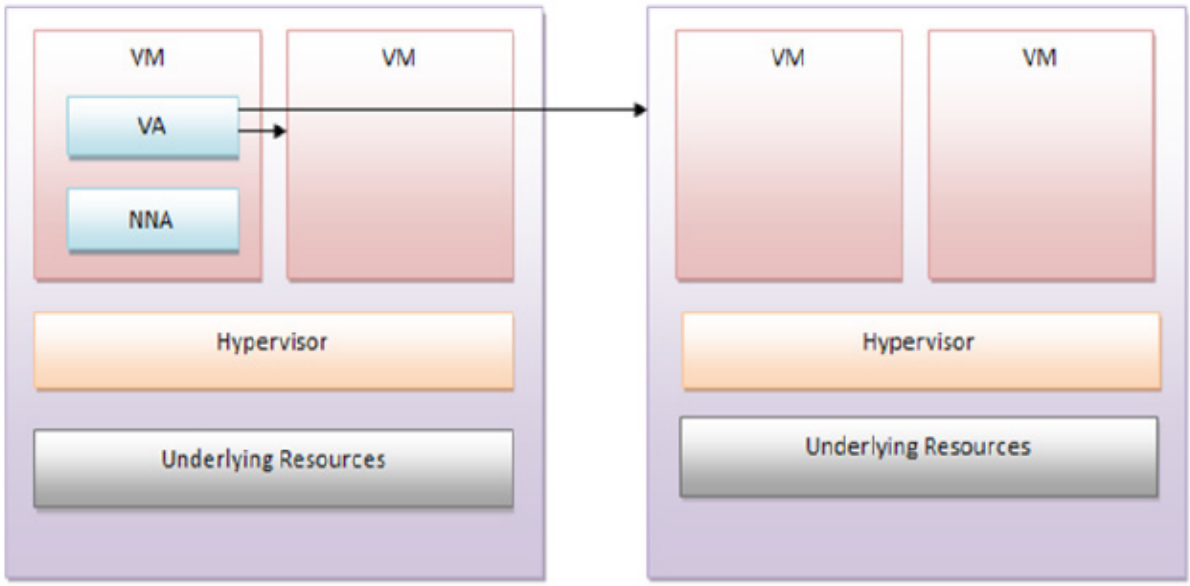

Figure 3.1: Example of a VM migrated to another node [1]

The second solution refers to cases where the victims of a noisy neighbor in an over-provisioned environment can get additional resources from another node (Figure 3.2). There are solutions like Beowulf Allocator [25] and SETI@home [24] that can allocate resources on any cluster in the grid, or even from another cluster [1].

The third approach is the closest to the approach taken in this thesis, and involves defining a quota system for physical resources. The main difference is setting hard limits at the scheduling point of the VM, whereas the fairness service try to achieve runtime prioritization based on these quotas.

In the paper [27] there is also described the solution as a ”fuzzy-logic affinity-aware engine” which helps with in-server VM scheduling. Even if this scheduling has proved to be very efficient, it still does not help in the use case where, during runtime one or more VMs become a noisy neighbor and take most of the resources of that node.

The Fairness service takes into consideration the users quotas, together with actual resource utilization, VM numbers, VM types and VM utilization by monitoring all the time all nodes. In cases where a sudden consumption spike happens for a VM and the user does not have the necessary quotas compared to other users on the same machine/node, the runtime prioritization will intervene. This means that commands will be sent to the hypervisor to reallocate the physical resources accordingly, and minimize the impact of

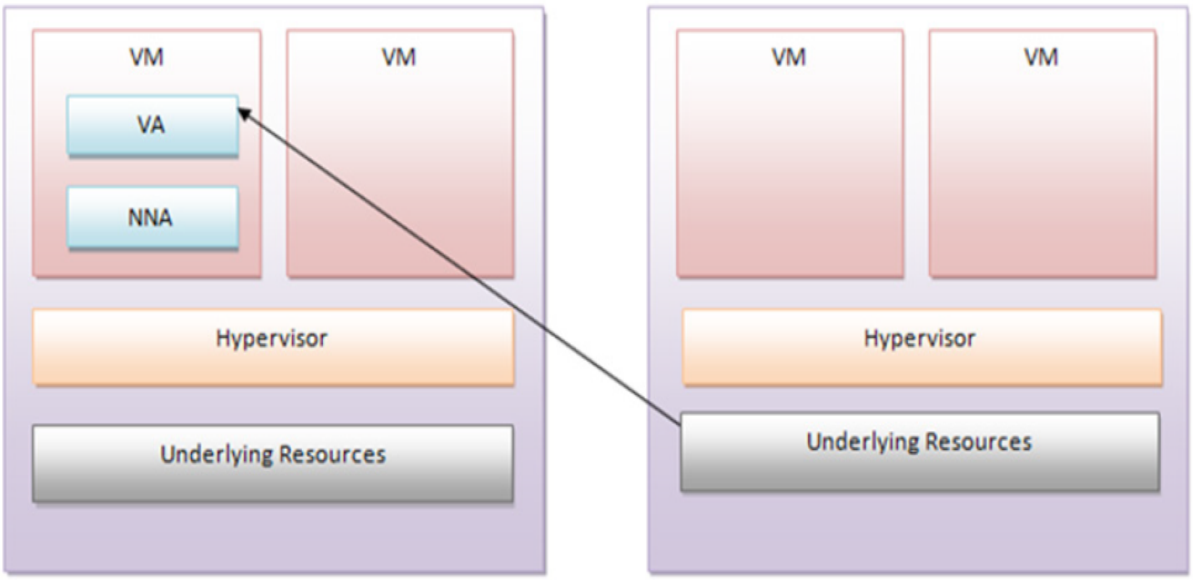

Figure 3.2: Example of a VM using resources from another node [1]

noisy neighbor to the affected VMs. If the problematic VM goes back to a normal utilization rate, the reallocation process will be triggered again and restore the previous resource priorities.
As an example, in the Chapter 6 we can notice a use case where there is a user with two VMs using all CPU shares of the host. If a second user has a VM of the same type on the host, with the help of the Fairness service, it will get more CPU time than the other two VMs in an situation of overprovisioning resources.

Chapter 4

The New Fairness Service

As discussed in the second chapter, the first version of the Fairness service proved to work accordingly and prioritize resources at runtime in a more ”fair” manner. However, the strong coupling with the other nova services made further development and maintenance too much of a difficult task.

The new solution starts by creating a decoupled service that will run on the controller node, as well as on each of the nodes and communicate between them independent of OpenStack. The basic algorithms and logic for the metrics, heaviness calculation and resource reallocation are kept from the first version, while the source code will be refactored, more efficient and easier to understand. This will help to keep the application in a running state for the next developers and the process of understanding what is already implemented kept simple.
The idea behind the decision comes from the Agile methodology, where it is recommended to start building the simplest version of an application that run properly. Afterwards, in each sprint, the application shall grow, but always be keep it in a running state, or reverted if something gets broken [28].

4.1 Multi-resource Fairness and User Heaviness

As explained in paper [9], it is hard to define uniquely runtime fairness, because resources such as CPU time, RAM, disk I/O, and network access, are shared, while users have different demands in order to satisfy their workloads. Since the only useable information from the hypervisor is the resource consumption, the fairness will be defined as ”prioritizing cloud users inversely to their heaviness”. In other words, the users that put less load on the system will get a higher priority compared to the users that spend a lot of resources. As a consequence, defining the user fairness directly depends on defining the user heaviness.

As mentioned before, the heaviness algorithm will be kept from the previous fairness. Therefore, the user heaviness is defined as the sum of heaviness of the user's VMs, which at its turn is defined by a metric that maps a resource consumption vector to a scalar [9].

The first configuration that is part of the heaviness algorithm is the VM flavor configuration. In order to understand how VM configuration affects heaviness, the following example is presented: There are three VMs, each of a different size from small to large (in terms of virtual resources assigned to it), and all created on the same node. All of them have to execute the same workload and utilize the same resources.
Usually, the size of the VM reflects how many actual resources the VM is expected to use and, respectively, how many resources gets reserved for it. This means that executing the workload on the small VM will result in a higher than expected load for that VM host, while executing on the large VM will result in a higher amount of unutilized resources. In contrast, executing the load on the medium VM will not result in many resources wasted or overused [9].

Consequently, the metric will influence the heaviness by lowering it if the users configured properly the VM. If the VM overuses or waste resources, the heaviness for the respective VM and user will increase.

Idle VMs will get the same treatment as the VMs which waste resources. This will force users to stop creating a large number of unused VMs and thus, increasing the overall over-provisioning factor (physical resources are reserved for each created VM).

The resources utilized by a VM is the most decisive factor that is part of the heaviness calculation. This information is collected periodically at runtime, and referred to as Runtime Utilization Information (RUI). The service will also collect the hardware information for the whole cloud cluster (all available nodes), referred to as Node Resource Information (NRI), and it will also be part of the heaviness calculation. The sum of all node's NRI will be referred to as Cloud Resource Supply (CRS).

Since both of these resources values are heterogenous and measurable in different units, they will be normalized to a scalar that will represent all of them equally.

Lastly, the heaviness will also take into consideration users' resource quotas beside the afford mentioned NRI, CRS and VM flavors.

4.2 Measurements Description 

To achieve as much efficiency as possible in virtualized environments, clouds use to overcommit resources (such as CPU) by a certain degree/factor. For example, a cloud can overcommit CPU with a factor of 5, meaning that, there can be 5 times more VCPUs that hardware CPUs in the system. This also means that the amount of resources a cloud budgets for a VM not only depends on the clouds overcommit factors in addition to the defined virtual resources.

From the previous implementation [9] we take the definition of VM endowment as the share of the VM's node's physical resources that are proportional to the VM's virtual resources. This definition will help define how much resources are budgeted to a certain VM and how they compare to the initial VM resources. To give an example, if a VM has

three VCPUs and runs on a host, which has VMs with a total of 15 VCPUs, then the VMs CPU endowment is 1/5 of the nodes CPU cycles.

The quotas of users are also not directly translatable to amounts of physical resources a cloud allocates for a user, even if the cloud size or number of user changes. In order to get a quota relatable to the real resources, the fairness quota will be defined as the overall cloud resource supply divided by the number of users. The resulting value will be the amount of physical resources a cloud can allocate to that user [9].

Since all measurements are on the same scale, the heaviness of a user can be defined as the sum of VM endowments, which also contains the over-commitment factors, plus the heaviness of each of its VM. The fairness quota will be subtracted from the user heaviness at the end and the result will be a scalar [9].

4.3 The Fairness Framework 

The OpenStack environment consists of two types of nodes: the controller node is the cloud coordinator which takes care of VM scheduling, access, keeps all the information in a dedicated database, has a communication server and in general, coordinates the whole cloud. From a different perspective, the controller is the single point of failure for the OpenStack cloud. The compute nodes are hosting the VMs and take care of processing the VM workloads.

The Fairness Framework (mention also as service) is built as a python service that has to run on the controller node, as well as on each compute node. As mentioned before, the service is decoupled from OpenStack and can run independently of it. The most important actions the service should execute on the controller node are:

- Collecting all NRIs from the compute nodes
- Assembling the CRS and sending it back to the nodes
- Forward heaviness to all nodes at specified time intervals

On the compute nodes, the Fairness service has to take care of the following:

- Gather and send the NRI to the controller node
- Save the received CRS to use in heaviness calculation
- Every time a user heaviness vector is received:
  - Gather RUI and recalculate the heaviness
  - Send forward the updated heaviness to the next compute node
  - Calculate the new priorities and reallocate resources if needed

The communication between the nodes is established using ZeroMQ in a peer-to-peer manner, independent of OpenStack communication. The details will be presented in the next section 5.4.

However, some information is required from OpenStack, such as the list of users, quotas or VMs. This is achieved by calling the public OpenStack web API, authenticating with administrator credentials and requesting the needed information. All http requests are made synchronous, such that the data is received exactly when needed.

In the current version of the Fairness Framework, there are six resources tracked: CPU time in seconds, memory in kilobytes, disk read speed, disk write speed, network receive speed and transmit speed.

The CPU time represents the amount of time needed by the CPU to complete certain tasks and, therefore, measures CPU usage. Since the CPUs on the nodes can be quite different, one period of time from a CPU can be much more performant than the same amount of time from another CPU. In order to normalize the performance, the Linux operating system uses the so called "BogoMIPS" [29].

The NRI represents the hardware resource capacities for a node, or in other words, the performance of the host. Since the hardware components do not usually change (while OpenStack is running), the NRI will remain unchanged throughout the program run. Consequently, the NRI will be sent only one time to the controller node, and the newly assembled CRS (the sum of all NRIs) will also be sent only one time. All nodes will store the CRS and use it accordingly every time is needed.

Regarding resources, the NRI uses various system calls to get the hardware information. In addition to the above-mentioned resource types, the NRI will also get the number of cores of a CPU and weight them using node's BogoMIPS to obtain the overall performance.

The RUI is collected periodically by each compute node for every successfully scheduled VM of every active user. In order to get the information about the usage of resource, the libvirt [22] library is used. Libvirt provides a user-friendly API and can connect to many hypervisors. In paper [9] is explained that for time-shared resources (CPU time, disk I/O, network access), the libvirt provides the accumulated resource consumption since boot time.
The framework will calculate the current measurement period by subtracting the accumulated consumption at the beginning of the period from the accumulated consumption at the end of the period.

After the CRS has been sent to the compute nodes, the controller node will initialize a vector of all users that have active VMs and their default heaviness (in the source code also referred to as greediness). Afterwards, at a configurable interval, the controller will send to the nodes this vector.
When a node will receive it, it will start to collect the RUI and then recalculate the heaviness for all VMs hosted on it. As also explained in paper [9, 10] the input elements for calculating heaviness are:

- The previous user heaviness
- The cloud CRS
- Scales and over-commitment factors
- VM endowments
- Current RUI

After the calculation and transformation to a generalizable scalar, the new vector of user heaviness will be calculated. That involves each time subtracting from the value of each user the old heaviness value and adding the new one. This is needed since the VMs of a user most likely will be hosted on more than a single node, thus making it impossible to calculate the entire user heaviness on a single machine; a user u_x heaviness will be the result of all nodes that have an active VM owned by u_x.
Also, to be remembered that the user heaviness represents the sum of VMs heaviness owned by that user, no matter on which node they are hosted.

As soon as the new vector is calculated, it will be send to the next node, in order for that node to start its calculation.

The next very important step for the compute node is to calculate priorities for all VMs and the reallocate the resources accordingly. As stated in [9], the framework reallocates resources, based on weights. For example, if a VM has a CPU weight of 1 and the second VM has a weight of 2, the second VM will get twice as much CPU time. However, if that time is not used, the first VM can still use the remaining if needed.
This represents an important feature compared to assigning hard limits, which is not that efficient: resources can get easily wasted, unused or even overused. Basically, the full capacity of the node can be utilized, even by smaller VMs. Following are the definitions for mapping heaviness to weights, as reused from previous implementation [9].

CPU shares are prioritized based on the ratios of shares. A CPU-share distribution of 100 shares for CPU 1 and 200 shares for CPU 2 has the same effect as 512 shares for CPU 1 and 1024 shares for CPU 2. The framework maps priorities to CPU shares in the range [1,100]. RAM soft-limits are assigned from 10 MB to the maximum amount of RAM available to the VM. Disk weights range from 100 to 1000 with 100 representing the lowest and 1000 the highest weight. These limits are defined by libvirt and have to be adhered to.

Network priorities were translated in previous Fairness implementation into HTB qdisc classes ranging from 1:0 to 1:98 with 1:0 being the lowest and 1:98 the highest priority. However, this part is not currently used since a better, more efficient implementation is being developed in parallel by another thesis. This method will be merged into the current Fairness implementation in later stages.

The reallocation process is executed by setting the respective priorities define above in libvirt. The library will send the appropriate commands to the chosen hypervisor at runtime, the result being visible in real time.

Chapter 5

Improving Message Exchange

By default, the OpenStack communication is achieved by having a RabbitMQ [31] server host on the controller node that acts as a centralized message broker. This means that all messages sent between nodes, will also traverse and be processed by the controller node. Since this is not scalable and strengthens the single point of failure disadvantage of the system, the first Fairness implementation introduced a decentralized version, as described in [9].

However, the implementation still resulted in a high message volume, which is quadratic in the number of nodes and produced periodically. This is the case, since every node sent directly messages to every other node. In addition, there were third party libraries involved in the communication process, such as Redis message queue, which added to the code complexity and maintenance effort.

The next sections will describe the new implementation for the message exchange and the integration within the general Fairness Framework.

5.1 Theoretical Insights For Reducing The Message Volume 

As explained in Chapter 4, the CRS message is exchanged only once, which means it is considered insignificant compared to the heaviness message that is sent periodically, for an undefined amount of time. Thus, the heaviness message volume will have to be investigated and reduced accordingly.

We have defined the following protocol in order to reduce the message volume: Consider a user heaviness vector $m \in R^u$ that cycles through all nodes in a predefined order and its value gets updated every time it reaches a new node. Suppose that the node order is given by the nodes' indices ($1 <= I < n$), node n_i sends message m to $n_i + 1$ and node n_n sends message m to node n_1. In this scenario, node $n_i + 1$ and n_1 are called the successor of n_i and n_n, respectively.

The message m has one entry for each user with at least an active VM, $u_x \in U$ and it is initialized with $-1 \cdot s(e_u(u_x))$. We can assume here that the x^th entry of m corresponds to the user u_x. After the first cycle, the message m will contain the calculated heaviness for all users in the cloud, for every node with at least an active VM. This is a result of each compute node updating the corresponding values for each hosted VM.

Afterwards, the message m will continue to cycle through the nodes, will give the user heaviness and will also be updated by each node. As mentioned in chapter 4, each node will first subtract the old value he added in the previous cycle and an add the new value.

More specifically, every node n_i will maintain a vector $m_{local} \in R^u$, which stores the value n_i added to m in the previous cycle. Therefore, m_{local} will be initialized with zeros and when n_i receives m from the predecessor, $m := m - m_{local}$. Every element in the array of users of m_{local} is then overwritten by zero and for each VM $v_a \in a(n_i)$, $h_v(v_a)$ is added to the x^th entry of m_{local}, whereat x is obtained by $o(v_a) = u_x$.

The CRS message will also use the same technique to have a complexity of $O(n \cdot u)$. The message will start from the central entity and traverse once all nodes n_i.

After m has traversed the circle twice (first traverse is the CRS message), every node knows the heaviness of each user and consequently, the message volume will become $2 \cdot n \cdot u \in O(n \cdot u)$. Therefore, the total message volume is bounded by $O(n \cdot u)$. Every node only receives and sends messages from one other node, which will ensure a low communication overhead. From a different perspective, all nodes, including the central entity (controller in this case) will be part of a communication circle.

5.2 Implementation Details

The ZeroMQ implementation followed closely the above mention theoretical description and creates a peer-to-peer network. As we can see in Figure 5.1, each node has a server that receives messages and a client that will connect to the next neighbor's server and send messages.

The topology of the network is a circle, where the controller node is the first peer and the rest of the nodes follow a certain order. In this case, ZeroMQ will create sockets for communication, where you have to bind and get a permanent connection between two nodes (until intentionally closed or crashed). More specifically, the client of one node will connect to the server of the next node and so on, until the circle is complete.

However, there exist an initial discovery step in the implementation, executed only once at system startup. When the fairness service on the controller node starts up, it will first connect to the local database and get the IP addresses of all active nodes in the system.

Afterwards, the ZeroMq server on the controller will startup and wait for three types of messages: neighbor and NRI message, CRS message and heaviness (HVN) message. The server on the compute nodes will start with the startup of the fairness service and wait for CRS and HVN messages.

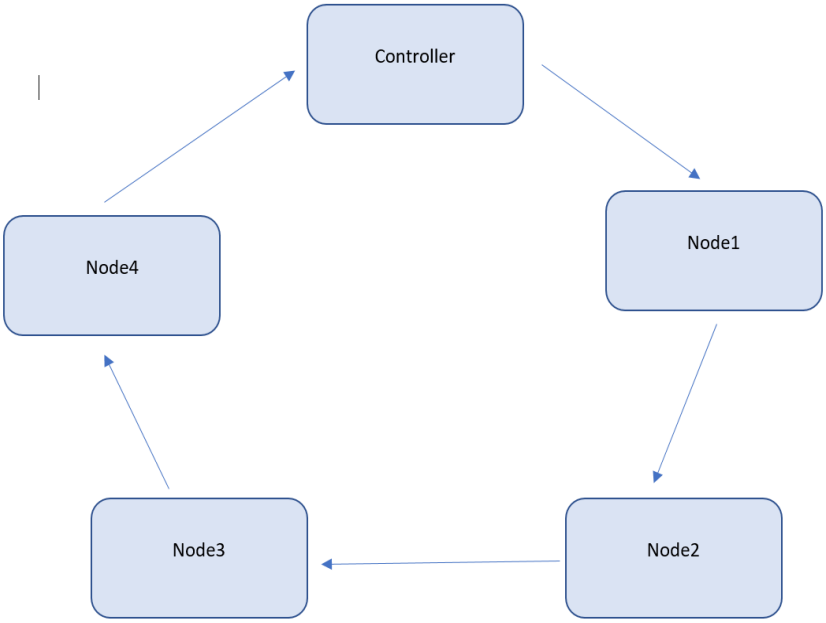

Figure 5.1: The circular topology of the Fairness Framework

5.3 Integration With The Fairness Framework

The neighbor and NRI message is part of the discovery step, the only time when nodes connect directly to the controller. This is needed in order for each node to find its neighbor and then connect to it. In order to reduce a cycle, the NRI is gathered by each node before and sent into the same message as with the neighbor message.

The neighbor message contains the IP address of the sender (the node that request a neighbor) and it will get as a return another IP address of the neighbor to connect to. The NRI will contain a vector of physical resources, as described in Chapter 4, which will be added to a CRS vector.

Since the controller keeps the list of all nodes previously taken from database, as soon as each node from the list has connected (identified by IP address), gave their own NRIs and received their neighbor, it will know when all nodes are ready. After creating the CRS vector out of all NRIs received, the controller will start the single CRS ring by connecting to the first node (the neighbor of the controller) and sending the CRS vector. When the CRS vector arrives back to the controller, thus getting the second type of message, the controller will start the HVN ring.

The HVN message will contain the initial vector of user heaviness, calculated as described in Section 6.1. When the HVN message will arrive back to the controller node, the fariness service will have to manage the third type of message.

The first step here is to calculate the amount of time it took for the message to go through the whole ring, then to wait an interval of time equal to the initial configured interval

minus the time it took for the message to come back. More specifically, $delta_{final} = delta_{interval} - delta_{circle}$. Afterwards, the received HVN message will be forwarded to the first compute node and the cycle of heaviness will be started again.

For each fairness service of the compute nodes, the first step is to calculate the host NRI. After having the required vector, the client part of the service will connect one time to the pre-configured controller address, send its own IP address and NRI, and wait for the neighbor. When the neighbor is received back (synchronously), the client will permanently connect to its neighbor.
The order of nodes booting up is not important since the ZeroMQ implementation knows to wait until the neighbor to connect to is available, and only then actually connect to it. For example if a node n_x tries to send a message to node n_y, but n_y is not yet booted up, the message will wait in an internal queue of n_x until the connection with n_y is available. This has been tested successfully even in different cases where a message is sent before the node is started.

The first type of message, a compute node waits for is the CRS message. After getting it, the fairness service will store it and use it in every new heaviness calculation that comes next.

When an HVN message arrives, the service will start to get the RUI, information about the received users and their VMs and then calculate the heaviness according to chapter 4. Here, the OpenStack API will be called in order to get the required information synchronously. The next step includes transforming the required value to normalized scalar and assembling the new vector of user heaviness as per Section 6.1.

After successfully updating the heaviness for each user that has active VMs on the current node, the updated HVN message will be sent to the next node. One of the most important steps is after heaviness is converted to weights and priorities, and the reallocation process starts. With the new values, the libvirt library will be called and VM underlying resources changed.

This HVN cycle will run indefinitely, the interval being configured in an external configuration file on the fairness service of the controller node.

Chapter 6

VCPUs And CPU Time Allocation

CPU time allocation, usually measured as a percentage of the CPU's capacity, is one of the most important resources allocated by a hypervisor, especially in a multi-tasking environment. In clouds, the CPU time for a CPU core is split between the VMs that run on that core, or applications within the VMs that utilize the respective physical core. In cases of over-provisioning, the split gets to be even more important, since more VMs compete for the same CPU time.

In order to investigate the benefits of the Fairness service regarding CPU time allocation, a specific use-case has been analyzed. Suppose one node n is shared by two users, u_x and u_y, with the same quotas and exists only one type of VM flavor. When VMs are instantiated, each of them will receive the same amount of CPU time, without taking the owner into consideration. This means, for example, if u_x creates two VMs and u_y creates only one VM, user u_x will get much more CPU time.

The last information for this use-case is about the application of the two users. U_x can split his workload on multiple VMs/cores while u_y is only able to run his application on a single core in a VM. It is a common issue in virtualized environments to be able to scale horizontally rather than vertically. It is easy for any application to run faster on a single bigger VM, but scaling vertically always has a limit; you cannot increase the speed/core of a processor infinitely, nor memory space or hard disk. The solution is to create/adapt your application to scale horizontally, so to be able to scale with the number of VMs.

Out of the box, the behavior of OpenStack allocates the same amount of CPU time for each VM, which means y will have half the CPU time of u_x and half of the performance respectively, even though they have the same quotas. In the Figure 6.1 we can observe the above-mentioned CPU sharing, which we consider not be fair between the two users.

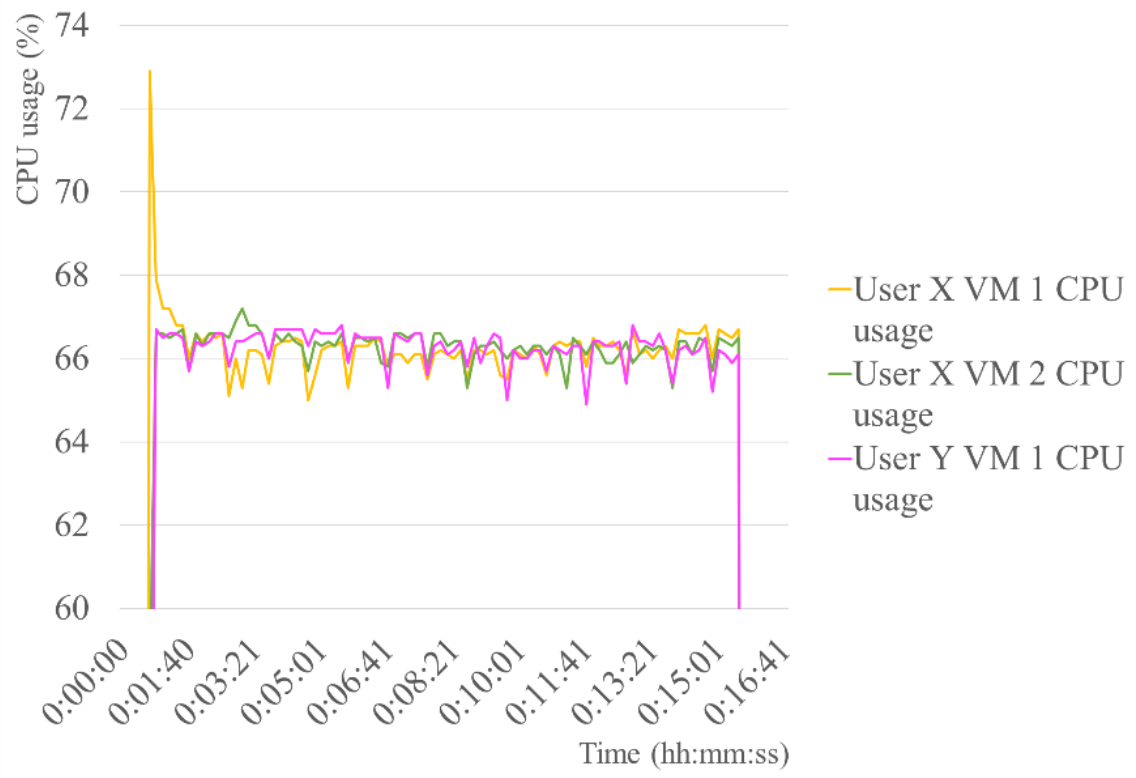

Figure 6.1: Default OpenStack CPU allocation, each VM has one VCPU

One solution offered by OpenStack is referred to as NUMA topology and CPU pinning [30]. This set-up allows a very specific configuration of what VMs get how much CPU, but is not supported by all hypervisors (e.g.: not supported by Hyper-V). CPU pinning allows a VM to have its own dedicated cores, with no sharing of the cores even in overcommitting environments. However, having dedicated cores means the system will not be efficient enough, especially in cases the CPU will not be used 100%, and rather waste resources.

As we can observe in Figure 6.2, if we use the fairness service in this scenario, the CPU time will be shared differently: user u_y will get about twice the amount of a VM of u_x.

Another possibility would be to increase the VCPUs of the VM of u_y to two and assume u_y can still scale on two cores of a single VM. Given the fact that VCPUs are also taken into consideration by the fairness service, the first user will get the same amount of CPU as the second one.

By default, OpenStack will schedule to the VM with two VCPUs approximately the same amount of CPU time as for two VMs with one VCPU together. Basicaly, all existent four VCPUs will get approximately the same amount of CPU time, as seen in Figure 6.3, where the pink VM with two VCPUs has twice the amount of CPU time compared to a VM with one VCPU.

The more interesting scenario is where the user u_y has the above-mentioned setup, but can only use one of the two VCPUs at full speed (only one VCPU out of two is stressed, the other one is idle), a typical case where an application cannot scale to more than a core. As seen in the Figures below 6.3 and 6.4, with OpenStack defaults only, the VM has

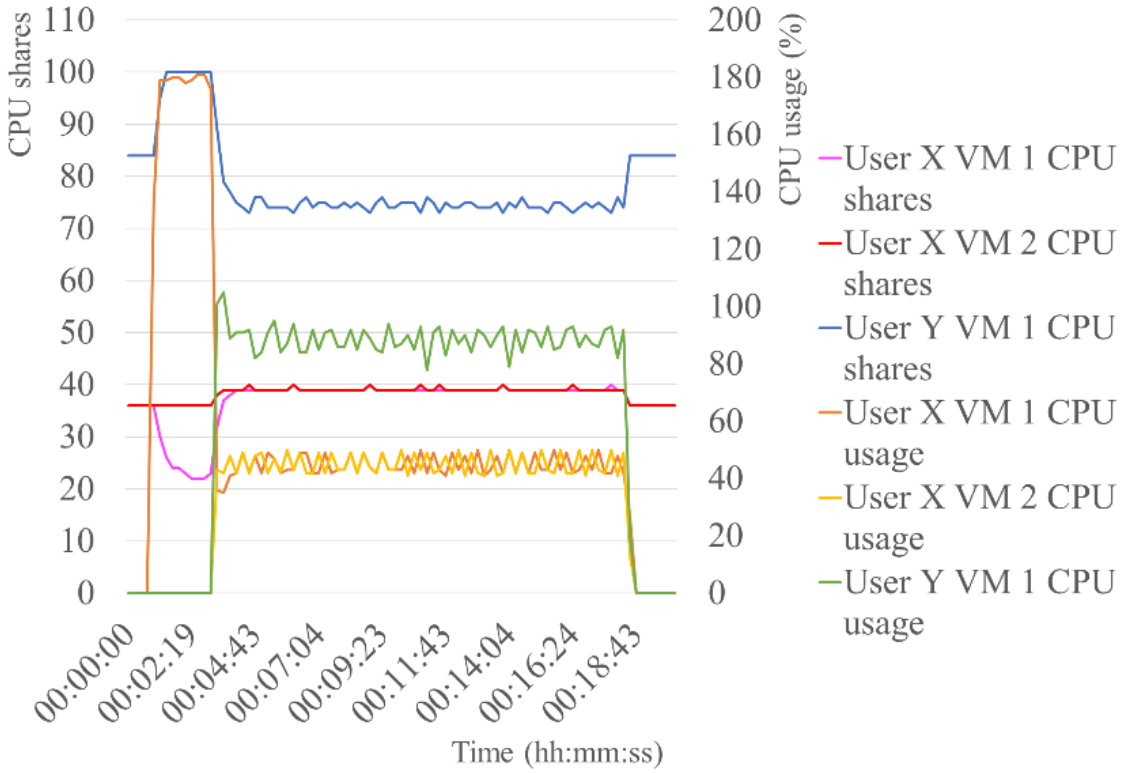

Figure 6.2: CPU allocation with the Fairness service on, each VM has one VCPU

around 90 CPU time (from a maximum of 200 in this case, 100 per core), while each VM of the other user got around 50. After starting the Fairness service (described in detail in the next chapters), we can notice that the VM of u_y dropped to around 50. This behavior means that the VM is penalized for scheduling two VCPUs, but using only one of them at full capacity. The rest of the CPU time will be split between the two other VMs.

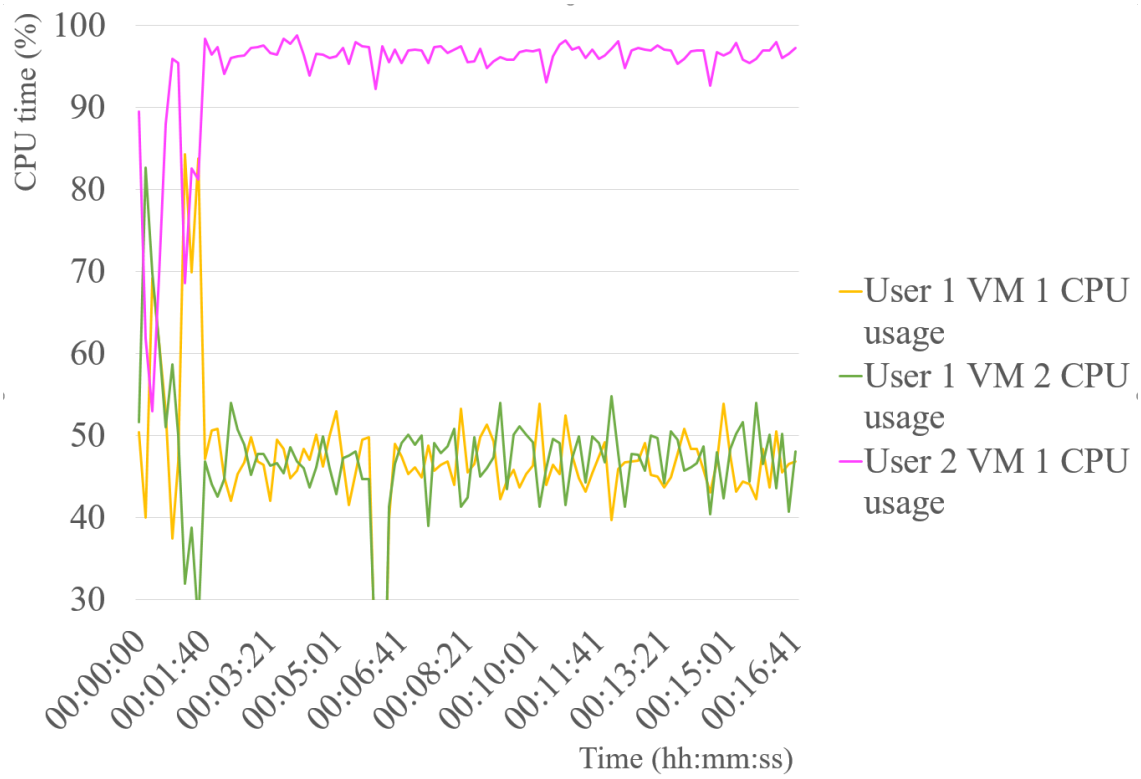

Figure 6.3: CPU allocation with OpenStack defaults, User 1 has two VMs, each with one VCPU and User 2 has one VM with two VCPUs

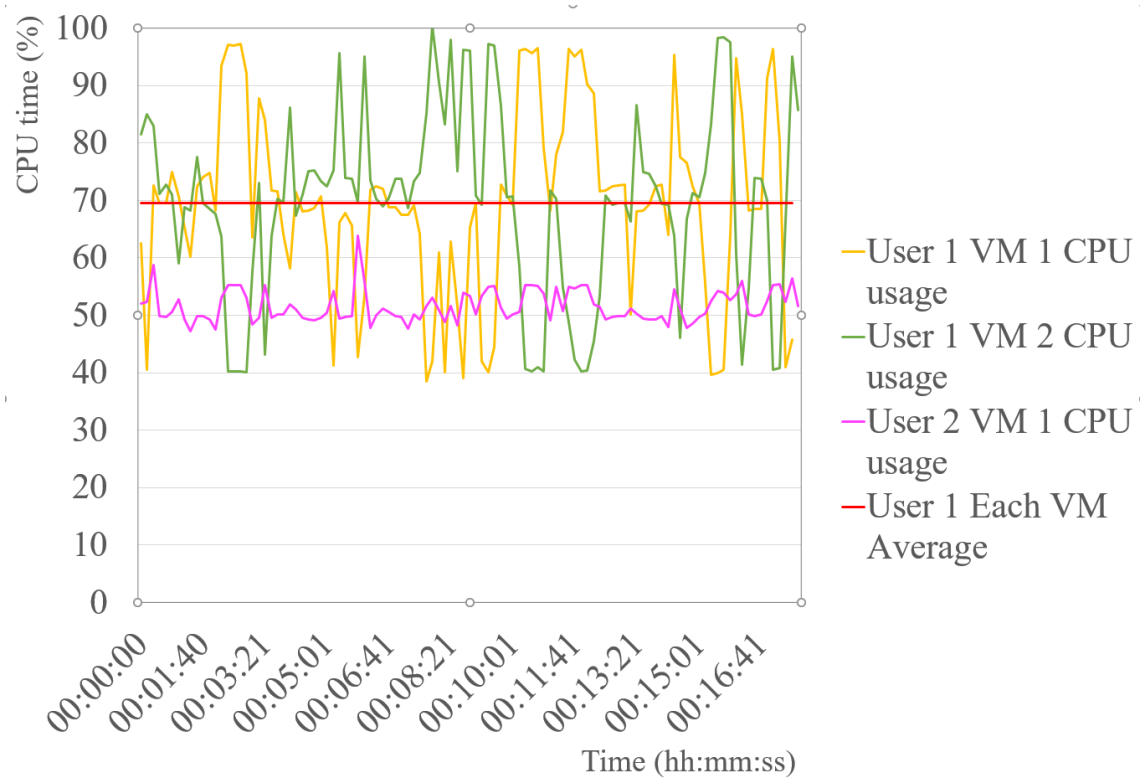

Figure 6.4: CPU allocation with Fairness service on, User 2 has a VM with two VCPUs, only one VCPU is stressed, the other VCPU is idle

Chapter 7

Evaluation

In order to test the fairness framework, a fresh installation of an OpenStack environment has been prepared previous to this paper. The OpenStack version used was Newton, installed on an Ubuntu 16.04 server following the official tutorials [9]. The system consisted of a controller node and five compute nodes. All tests were conducted over a period of approximately 15 minutes.

The first tests for the new fairness framework were presented in the previous chapter. They include testing on a single node two users with the same type of VMs. One of them has two VMs and the other one has only one. The difference when running with the fairness service and without it was clear and proved that the framework works accordingly.

In the next tests, the single VM of the user was given one more VCPU. Here the framework showed the performance penalty in cases users schedule resources that they do not use, compared to the default behavior of OpenStack, as seen in Chapter 5.

The multiple nodes tests are usually more important since they are much closer to a real-life situation. Even if our framework focuses on fairness at node level, there are still many situations taken into consideration: users that have VMs on more than a single node, over-commitment for some of these and inter-node communication. We tested the setup also present in paper [9] in order to verify if the service still performs as intended with the new implementation.

The setup consists of two compute nodes and three users. The first user only has VMs on the first node, the second has on both nodes and the third user has a VM on the second node. User 3 will have an idle CPU, while User 1 and first VM of User 2 will compete for the CPU on the first compute node.

In the Figure 7.1 we notice, that without the framework, The VM 2 of User 2 can use the entire resources of node 2, since User 3 does not request almost any CPU time. The situation on node 1 is different, since the two VMs compete for the CPU Time and, get they get equal amount of CPU time.

After starting our framework (Figure 7.2), the resource distribution will change: Since both users have the same quotas, but User 1 only 1 VM on node 1 and User 2 has another

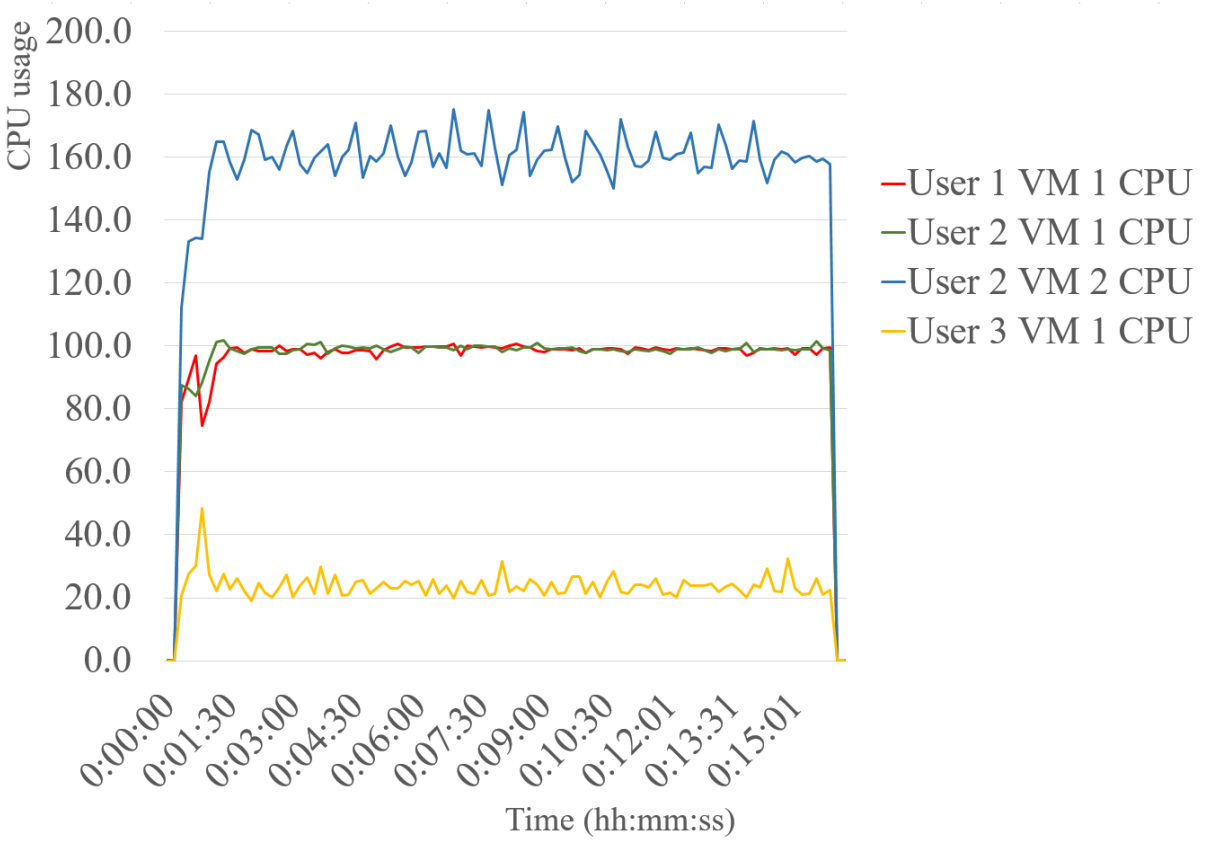

Figure 7.1: Default OpenStack CPU allocation, multinode setup: The VM of User 1 and VM 1 of User 2 belog to node 1, the other two VMs belog to node 2

VM on the node 2, the first user will get more CPU than the second one.
Running another VM on a different node, will significantly reduce the priority of resources on the current node.

These easy evaluation setups proved that the new implementation of the framework still works as the previous one, since the algorithms applied the same logic. This expected result confirms the finding in the paper [9]. However, after the full implementation of the network reallocation and messaging, a full suite of tests is recommended.

The tests should reflect as much as possible the reality, which means more nodes and many more VMs. The expected results could be defined before for several test cases which will include the resource contention for all resource types. Nodes connecting and disconnecting from the OpenStack network could be another type of use-case. Even if the amount of time needed to create and asses such tests is quite high, the importance remains unquestionable in order to create a real stable product.

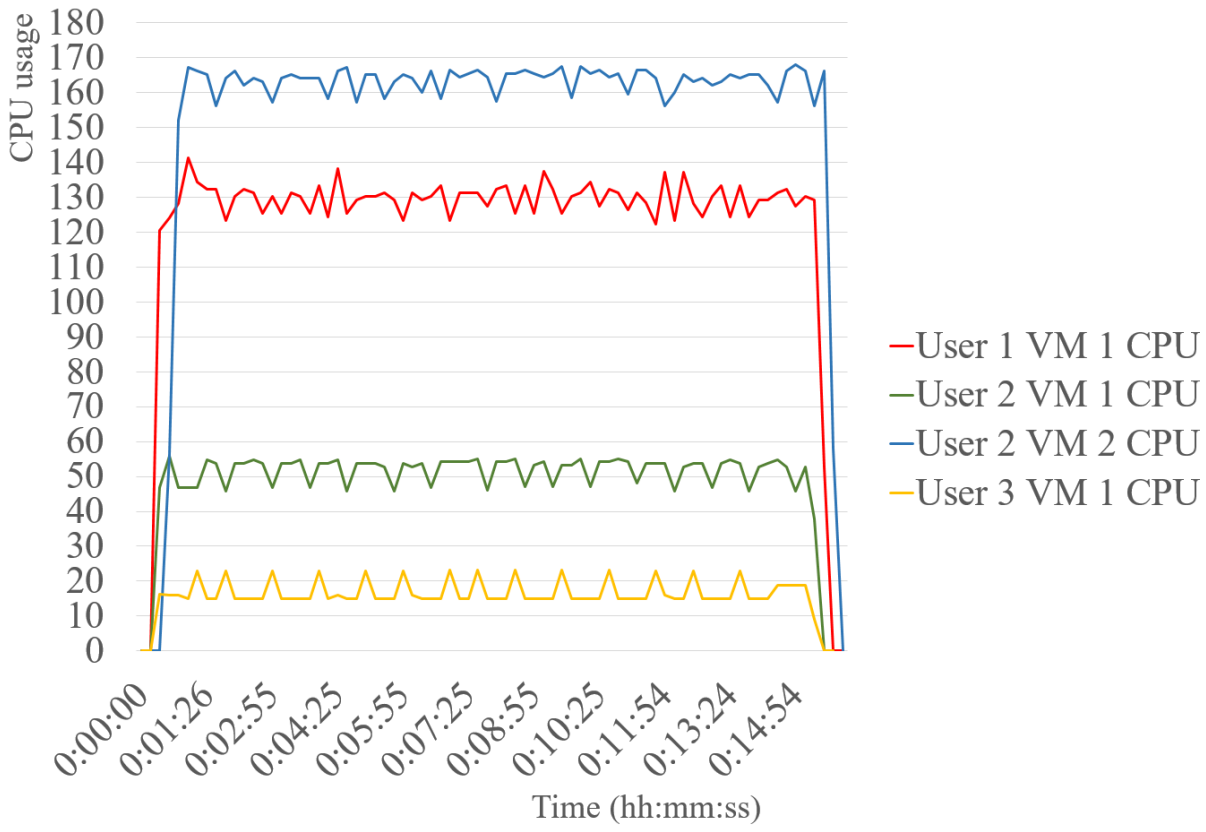

Figure 7.2: CPU allocation with Fairness Framework on, multinode setup: The VM of User 1 and VM 1 of User 2 belog to node 1, the other two VMs belog to node 2

Chapter 8

Future Work

It is important to define the next steps that come after this initial version, in order to ensure a good direction for the project in the next development phases. One first important step is to further consolidate the new implementation and adding virtual resources counterparts for the missing ones.

Another important step would be to further implement the messaging system, together with failure detection or recovery if nodes shut down.

8.1 VR Counterparts

When we want to create a new VM, we have to choose from a set of predefines VM Flavors. These flavors define certain amounts of virtual resources which will be allocated to the respective VMs. However, OpenStack does not have a virtual counterpart for every physical resource. For example, you can define VCPU, VRAM or virtual disk space, but cannot define virtual disk I/O or IOPS, nor virtual network access.

Since our framework assumes the existence of a virtual counterpart for each physical resource, service should derive the virtual counterparts for the missing physical resources such as disk I/O or network access.

In the current implementation, the fairness service equally partitions resources such as virtual disk I/O or network access among the VMs in order to obtain the amount of respective virtual resources. For example, if a node with a bandwidth of 50 Gbit/s hosts five VMs, each VM's will have a limit set to 10 Gbit/s.
Since VMs vary significantly in size, at least in terms of VCPU and VRAM, the derivation should take these sizes into account, when obtaining the virtual counterparts of disk I/O, network access and other resources not currently part of the VM flavours (VM flavors contain no virtual counterpart). For example, a VM with a lot of VCPUs could get the same proportion of network as the proportion of VCPUs to CPUs.

8.2 Imporving The Message Exchange

Currently, our fairness framework has implemented a simple architecture that proved to work in the current setup. However, for the future, it is highly recommended to also implement mechanisms that would bring robustness, security, scalability and fault-tolerance.

A good example is the situation when more nodes join the system, without restarting the whole OpenStack environment, and, even more, when a node shuts down/leaves the system. The fairness system should properly treat these cases and update the required NRI/CRS vectors for all compute nodes. Another example could be in the future, where many servers start to support hot-plug of memory, different cards or even CPUs. This means that, while running, one could just add more RAM to the system and our service should also update the relevant information.

One solution that can be applied is going further with the peer-to-peer network implementing a reliable DHT network. In this way, the communication should not just be fault-tolerant and robust, but also scale to a much higher number of nodes. Currently, ZeroMQ used in the framework could indeed support such implementation and provide the necessary communication mechanisms.

8.3 Other Improvements

There are a number of other possible improvements that could help the framework to be more efficient and accurate.

Improving network reallocation is a very important topic, since currently libvirt can only set hard limits for the network traffic. The first fairness implementation used Linux specific tools like TC [33] to overcome this, but the recommended solution should be based on libvirt, as for the other resource types. Finding a more reliable solution for this would prove to be valuable for the overall network control of the VMs.

Adding more physical and virtual resource type could also increase the resource distribution accuracy. For example, disk space, disk IOPS, graphic cards performance and others could be introduced.

As mentioned in the previous chapter, a large testing suite should also be part of the next implementation phases, because of the numerous use-cases needed to be assessed.

Chapter 9

Conclusions

This paper investigated the fairness in virtualized environments and tries to bring a solution that would better distribute the physical resources to the VMs. This approach enforces the fairness at runtime and allows the OpenStack environment to react to resource contentions and reduce the need for VM recreation or rescheduling [9].

It has been investigated if some of the todays problems can be fixed with the help of the fairness framework. The noisy neighbor is a very important one that can be addressed by the runtime resource scheduling of the fairness service.

Because of complexity, maintenance and reusability issues, a new decoupled version of the fairness framework has been implemented, while keeping the logic of algorithms the same. The communication has also been completely redesigned and implemented as a decentralized peer-to-peer network between the OpenStack nodes. The new version of the service proved to work accordingly on an in-house OpenStack installation, with one controller and five compute nodes.

The test cases revealed that the new implementation keeps the same behavior as initially; this means that, in the same scenarios, the resources allocated to the VMs were around the same percentage. After all parts are implemented, further tests should be created for all different resources to ensure the fairness of the system.

The fairness framework proved to work correctly in basic situations, ensuring resource allocation fairness whether it was a resource competition, or idle nodes not using the initially allocated resources.

Bibliography

[1] A. Makroo, D. Dahiya: A Systematic Approach to Deal with Noisy Neighbour in Cloud Infrastructure, School of Engineering and Technology, Ansal University, Gurgaon - 122003, Haryana, India, May 2016.

[2] L. Schubert, K. Jeffery. Advances in Clouds. European Commission, Publications Office of the European Union, Luxembourg, 2012.

[3] Vaquero LM, Rodero-Merino L, Caceres J, Lindner M. A break in the clouds: towards a cloud definition. ACM SIGCOMM Computer Communication Review. 2008 Dec 31; 39(1):50-5.

[4] Bezemer CP, Zaidman A. Multi-tenant SaaS applications: maintenance dream or nightmare? Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), ACM. 2010 Sep 20; 88-92.

[5] D. Breitgand, Z. Dubitzky, A. Epstein, A. Glikson, I. Shapira. SLA-aware resource over-commit in an IaaS cloud. Network and service management (cnsm), 2012 8th international conference and 2012 workshop on systems virtualization management (svm), pp 73-81, Oct 2012.

[6] S. Zahedi, B. Lee. Ref: Resource elasticity fairness with sharing incentives for multiprocessors. Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, pp 145-160, New York, NY, USA, 2014. ACM.

[7] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11), pp 24-24, Boston, MA, USA, March 2011.

[8] D. Dolev, D. Feitelson, J. Halpern, R. Kupferman, N. Linial. No Justified Complaints: On Fair Sharing of Multiple Resources. 3rd Innovations in Theoretical Computer Science Conference (ITCS'12), pp 68-75, Cambridge, MA, USA, January 2012.

[9] P. Poullie, S. Mannhart, B. Stiller. Extending OpenStack to Adapt VM Priorities during Runtim, University of Zürich, Department of Informatics, March 2016.

[10] P. Poullie, S. Mannhart, B. Stiller. Virtual Machine Priority Adaption to Enforce Fairness Among Cloud Users, University of Zürich, Department of Informatics, March 2016.

[11] D. Klusáček, H. Rudová, M. Jaroš. Multi resource fairness: Problems and challenges. In N. Desai and W. Cirne, editors, Job Scheduling Strategies for Parallel Processing, pp 81-95, Neu-veden, 2014.

[12] Y. Etsion, T. Ben-Nun, and D. G. Feitelson. A Global Scheduling Framework for Virtualization Environments. 2009 IEEE International Symposium on Parallel Distributed Processing, IPDPS'09, pp 1-8, Rome, Italy, May 2009.

[13] A. Gutman and N. Nisan. Fair Allocation without Trade. 11th International Conference on Autonomous Agents and Multiagent Systems, Vol. 2 of AAMAS'12, pp 719-728, Valencia, Spain, June 2012.

[14] Y. Zeldes and D. G. Feitelson. On-line Fair Allocations Based on Bottlenecks and Global Priorities. 4th ACM/SPEC International Conference on Performance Engineering, ICPE'13, pp 229-240, Prague, Czech Republic, April 2013.

[15] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Heterogeneous Resources in Datacenters. Technical Report UCB/EECS2010-55, EECS Department, University of California, Berkeley, CA, USA, May 2010.

[16] F. Kelly et al. Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability. The Journal of the Operational Research Society, 49(3):237-252, March 1998.

[17] D. Klusáček and H. Rudová. Multi-resource Aware Fairsharing for Heterogeneous Systems. In: Walfredo Cirne and Narayan Desai (Editors), 18th International Workshop on Job Scheduling Strategies for Parallel Processing, Revised Selected Papers, JSSPP 2014, pp 53-69. Springer International Publishing, May 2015.

[18] D. Klusáček, H. Rudová, and M. Jaroš. Multi Resource Fairness: Problems and Challenges. In: Narayan Desai and Walfredo Cirne (Editors), Job Scheduling Strategies for Parallel Processing, Vol. 8429 of Lecture Notes in Computer Science, pp 81-95. Springer, Berlin/Heidelberg, Germany, 2014.

[19] H. Liu and B. He. Reciprocal Resource Fairness: Towards Cooperative Multiple-Resource Fair Sharing in IaaS Clouds. 2014 IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, pp 970-981, New Orleans, LA, USA, November 2014.

[20] Redis Labs. Redis. http://redis.io/, 2015, Online, accessed April 2017.

[21] OpenStack Foundation. OpenStack Wiki: ZeroMQ. https://wiki.openstack.org/wiki/ZeroMQ, 2014. Online; , accessed April 2017.

[22] Red Hat, Inc. Libvirt: Internal Drivers. https://libvirt.org/drivers.html, 2016. Online, accessed April 2017..

[23] Netflix SimianArmy, https://github.com/Netflix/SimianArmy/wiki/The-Chaos-Monkey-Army, January 2015. Online, accessed April 2017.

[24] Anderson DP, Cobb J, Korpela E, Lebofsky M, Werthimer D. SETI@ home: an experiment in public-resource computing. Communications of the ACM. 2002 Nov 1; 45(11):56-61.

[25] Hu F, Evans JJ. Power and environment aware control of Beowulf clusters. Cluster Computing. 2009 Sep 1; 12(3):299-308.

[26] E. Caron, J. Rouzaud-Cornabas: Improving users' isolation in IaaS: Virtual Machine Placement with Security Constraints, UniversitÂ´e de Lyon, Lyon, 2014.

[27] L. TomÃ¡s, J. Tordsson . Reducing Noisy-Neighbor Impact with a Fuzzy Affinity-Aware Scheduler, University of Castilla-La Mancha, Spain, 2015.

[28] B. Meyer. Agile!, Springer International Publishing, Luxembourg, 2014.

[29] BogoMIPS, https://www.linux-mips.org/wiki/BogoMIPS, April 2015. Online, accessed April 2017.

[30] Virt driver pinning guest vCPUs to host pCPUs, https://specs.openstack.org/openstack/nova-specs/specs/juno/approved/virt-driver-cpu-pinning.html. Online, accessed April 2017.

[31] RabbitMQ, https://www.rabbitmq.com. Online, accessed April 2017.

[32] OpenStack Foundation. OpenStack Installation Guide for Ubuntu 16.04.https://docs.openstack.org/newton/install-guide-ubuntu/, 2016. Online, accessed April 2017.

[33] TC, http://lartc.org/manpages/tc.txt, accessed May 2017.

Abbreviations

API	Application Programming Interface
CC	Cloud Computing
CRS	Cloud Resource Supply
I/O	Input/Output
IOPS	Input/Output Operations per Second
MQ	Message Queue
NRI	Node Resource Information
PM	Physical Machine
R/W	Read/Write
REST	Representational State Transfer
RUI	Runtime Utilization Information
URL	Uniform Resource Locator
VM	Virtual Machine

List of Figures

3.1 Example of a VM migrated to another node [1] 9

3.2 Example of a VM using resources from another node [1] 10

5.1 The circular topology of the Fairness Framework 19

6.1 Default OpenStack CPU allocation, each VM has one VCPU 22

6.2 CPU allocation with the Fairness service on, each VM has one VCPU . . . 23

6.3 CPU allocation with OpenStack defaults, User 1 has two VMs, each with one VCPU and User 2 has one VM with two VCPUs 24

6.4 CPU allocation with Fairness service on, User 2 has a VM with two VCPUs, only one VCPU is stressed, the other VCPU is idle 25

7.1 Default OpenStack CPU allocation, multinode setup: The VM of User 1 and VM 1 of User 2 belog to node 1, the other two VMs belog to node 2 . 28

7.2 CPU allocation with Fairness Framework on, multinode setup: The VM of User 1 and VM 1 of User 2 belog to node 1, the other two VMs belog to node 2 29

Appendix A

Installation Guidelines

In this appendix, there are provided additional technical information that would help to install and test the Fairness Framework in an OpenStack Environment. The instructions are meant to be executed with root permissions, on an Ubuntu 16.04 Linux distribution. Other distributions or operating systems have not been tested.

A.1 Prerequisites

The first step into testing the framework is having an OpenStack environment installed. The Fairness Framework was successfully tested in a system formed by a controller and five individual compute nodes.
The version of OpenStack called Newton has been successfully tested; other versions of OpenStack should work as well, as long as the URLs of the OpenStack API are updated correspondingly to the API version. The installation of OpenStack was done according to the official tutorials for the version Newton and Ubuntu 16.04.

A.2 Controller Node

In order to install the fairness service on the controller the following steps are required:

- Copy the source code into the local Python installation on the controller server, or any other folder, as long as there is an entry in the environment variable $PYTHON\_PATH$ with the path of that location
- Enter the folder *openstack_fairness_service*, open the file $fairness.ini$ and add the corresponding properties required to authenticate to the OpenStack API. The ring interval which specifies how often the HVN message is sent, must be also set (in seconds).

- While in this folder, the addition libraries must be installed with the following command:

```
(sudo) pip install -r requirements.txt
```

- Navigate to the folder *fairness/controller_service* and execute the following command to start the fairness service on the controller node:

```
(sudo) python controller_manager.py ../../fairness.ini
```

A.3 Compute Node

The following step are required to be executed on each individual compute node:

- Copy the source code into the local Python installation on the controller server, or any other folder, as long as there is an entry in the environment variable *PYTHON_PATH* with the path of that location
- Enter the folder *openstack_fairness_service*, open the file *fairness.ini* and add the corresponding properties required to authenticate to the OpenStack API. The Controller Node IP address must be provided so that the node will know how to connect to the OpenStack Controller.
- While in this folder, the addition libraries must be installed with the following command:

```
(sudo) pip install -r requirements.txt
```

- Navigate to the folder *fairness/node_service* and execute the following command to start the fairness service on the controller node:

```
(sudo) python node_manager.py ../../fairness.ini
```