Nilson Rubens de Moraes Filho

Improving Load Balancing in Virtualized Environments using Pearson's Correlation

Sorocaba, SP

May 29, 2018

Nilson Rubens de Moraes Filho

Improving Load Balancing in Virtualized Environments using Pearson's Correlation

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC-So) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Linha de pesquisa: Engenharia de Software e Sistemas de Computação.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCC-So

Supervisor: Prof. Dr. Fábio Luciano Verdi

Sorocaba, SP

May 29, 2018

Moraes Filho, Nilson Rubens de

Improving Load Balancing in Virtualized Environments Using Pearson's Correlation / Nilson Rubens de Moraes Filho. -- 2018.
108 f. : 30 cm.

Dissertação (mestrado)-Universidade Federal de São Carlos, campus Sorocaba, Sorocaba
Orientador: Fábio Luciano Verdi
Banca examinadora: Mateus Augusto Silva Santos, Gustavo Maciel Dias Vieira
Bibliografia

1. Load Balancing. 2. Virtualization. 3. Pearson Correlation. I. Orientador. II. Universidade Federal de São Carlos. III. Título.

Ficha catalográfica elaborada pelo Programa de Geração Automática da Secretaria Geral de Informática (SIn).
DADOS FORNECIDOS PELO(A) AUTOR(A)
Bibliotecário(a) Responsável: Maria Aparecida de Lourdes Mariano – CRB/8 6979

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Nilson Rubens de Moraes Filho, realizada em 29/05/2018:

[illegible]

Prof. Dr. Fabio Luciano Verdi
UFSCar

Prof. Dr. Mateus Augusto Silva Santos
Ericsson

[illegible]

Prof. Dr. Gustavo Maciel Dias Vieira
UFSCar

Certifico que a defesa realizou-se com a participação à distância do(s) membro(s) Mateus Augusto Silva Santos e, depois das arguições e deliberações realizadas, o(s) participante(s) à distância está(ao) de acordo com o conteúdo do parecer da banca examinadora redigido neste relatório de defesa.

[illegible]

Prof. Dr. Fabio Luciano Verdi

To my parents, Vilma (in memoriam) and Nilson
To my wife Márcia
To my sons Beatriz, Felipe, Júlia and Luiza.

Acknowledgements

I would like to acknowledge and thank,

God, as the source of all inspiration. *"For the Lord gives wisdom, from his mouth come knowledge and understanding"* (Prv 2, 6).

My parents, that always believed in my potential and encourage me to keep moving forward.

My wife, that supported me during this long journey and understood all the time and effort that it took me to get here, helping me to surpass all difficulties.

My kids, that had patience during all the countless weekends that I had to study.

To my family, in special my sisters, that were always on my side.

The companies Convergys and Qlik, for allowing me to pursue this dream.

My director Cesar Ripari and manager Alexandre Zacarias for supporting me at work during the conclusion of this dissertation.

Prof. Fábio, for all the knowledge transfer, helpful insights, persistence and support to assist me getting this project done.

All UFSCAR PPGCCS teachers, for the excellent classes and dedication.

All UFSCAR employees, especially Roberto Romualdo Marvulle, that supported us during the whole course.

My colleagues from the LERIS laboratory, especially André Beltrami.

UFSCAR, to seek and provide high quality education for its students.

São Paulo Research Foundation (FAPESP), for the financial support, grant # 15/19766-9.

"I have the strength for everything through him who empowers me"
(Phil 4,13)

Abstract

Virtualização é um dos alicerces da computação em nuvem pois permite melhor utilização de recursos computacionais em um centro de dados. Existem diferentes abordagens para virtualização que oferecem a mesma funcionalidade, mas com diferentes níveis de abstração e métodos. Neste sentido podemos citar o uso de Máquinas Virtuais e contêineres. Definimos Elemento Virtual (EV) como sendo uma máquina virtual ou contêiner, e usaremos este conceito para generalizar a nossa proposta de balanceamento de carga. O balanceamento de carga pode ser realizado através da migração dos EVs, reduzindo o consumo de energia, disponibilizando uma melhor distribuição dos recursos computacionais e permitindo que clientes movam EVs de um provedor de nuvem para outro que ofereça melhor SLA ou custo. Existem alguns métodos que tratam da melhora do balanceamento de carga em um centro de dados. Um deles utiliza o coeficiente de correlação de Pearson relacionado ao consumo de CPU, para migrar EVs de um servidor sobrecarregado para outro que possua melhor disponibilidade de recursos. O coeficiente de correlação de Pearson estima o grau de dependência entre duas quantidades. Já a migração em tempo real é uma característica da virtualização que permite que um EV seja transferido de um servidor para outro, mantendo a execução dos processos ativos. Porém migrar um EV que possua forte dependência em relação ao tráfego de rede interno do servidor, pode gerar um aumento do consumo de recursos computacionais do ambiente. Isto devido ao aumento do consumo de rede ocasionado pela migração do EV para outro servidor que, topologicamente, esteja distante do servidor atual. Esta dissertação tem como objetivo definir uma heurística para melhorar o processo de decisão de migração de EVs. A heurística utiliza o coeficiente de correlação de Pearson e leva em conta não só o consumo de CPU mas também o tráfego de rede interno entre EVs. Os resultados mostraram que o uso da heurística, em um ambiente com tráfego de rede interno entre EVs, melhorou o processo de decisão em 18%, comparado com o método que considera apenas o coeficiente de correlação baseado em CPU.

Palavras-chaves: Virtualização. Máquina Virtual. Contêiner. Correlação de Pearson. Migração em tempo real. Balanceamento de Carga. Nuvem. Centro de Dados.

Abstract

Virtualization is one of the foundations of cloud computing as it allows better utilization of computing resources in a data center. There are different virtualization approaches that offer similar functionality, but with different levels of abstraction and methods. In this sense we can mention the use of Virtual Machines and containers. We define Virtual Element (VE) as a virtual machine or a container, and we will use this concept to make our load balancing approach generic. Load balancing can be achieved through live migration of VEs, reducing energy consumption, enabling better distribution of computational resources and allowing customers to move VEs from a cloud provider to another one that may offer better SLA or costs. There are some methods that addresses the load balancing improvement in a data center. One of them applies the Pearson correlation coefficient related to CPU usage, to migrate VEs from an overloaded host to another that have better availability of resources. The Pearson correlation coefficient estimates the dependency level between quantities. Yet, live migration is a virtualization feature that allows a VE to be transferred from one equipment to another, keeping the active processes running. However, to migrate a VE that has strong dependency with the internal network traffic from a host, can create an increase in the overall network consumption due to the migration of the VE to another server, topologically distant from the current host. This dissertation defines a heuristic that has as objective improve the migration decision process of VEs. The heuristic applies Pearson's correlation coefficient and takes in consideration not only CPU consumption, but also the internal network traffic between VEs. Results shown that the application of the heuristic improved the decision process in at least 18% compared to a method that considers only CPU correlation coefficient.

Key-words: Virtualization. Virtual Machine. Container. Pearson Correlation. Live Migration. Load Balancing. Cloud. Data Center.

List of Figures

List of Tables

List of abbreviations and acronyms

API	Application Programming Interface
CaaS	Container as a Service
CoW	Copy-on-write
CRIU	Checkpoint Restore In Userspace
GB	Gigabytes
HA	High Availability
HPC	High Performance Computing
IaaS	Infrastructure as a Service
I/O	Input/Output
JSON	JavaScript Object Notation
KB	Kilobytes
MB	Megabytes
OS	Operating System
PaaS	Platform as a Service
REST	Representational State Transfer
RX	Received Data
SaaS	Software as a Service
SLA	Service Level Agreement
TX	Transmitted Data
VE	Virtual Element
VM	Virtual Machine

Contents

1 Introduction

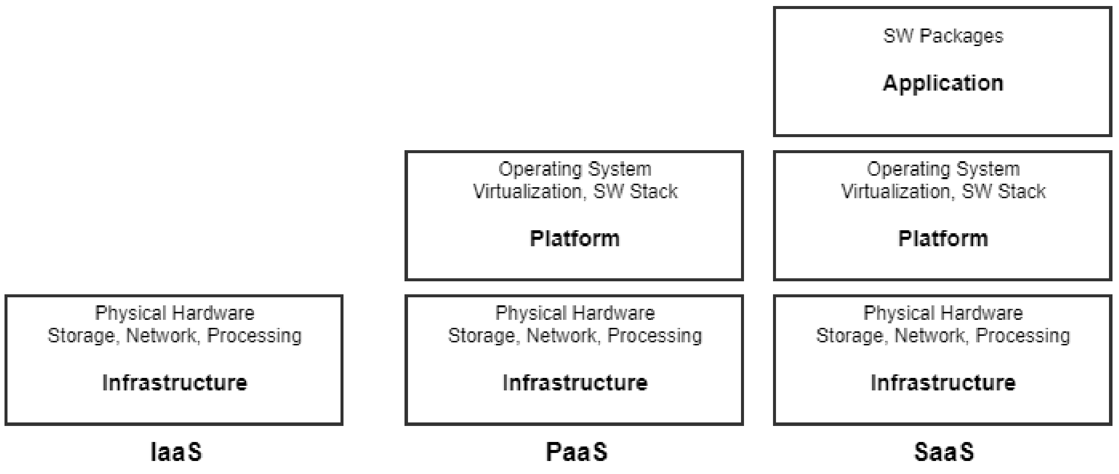
Data center is a facility responsible for handling servers, mainframes, data storage and networking. The name data center comes from the fact that, in a centralized environment, all the devices are physically located in the same place. Cloud computing is an extension of this paradigm, in which a series of virtualized resources, like hardware and development platforms, can be exposed as services and be accessed through the network (BUYYA et al., 2009). These resources can be dynamically reconfigured and adjusted to different loads, allowing an optimized allocation of resources, concept called elasticity. Resources are typically explored in a pay per use model in which the infrastructure provider guarantee a certain level of service defined in a service level agreement (SLA)(VAQUERO et al., 2008). Companies can reduce their operational costs using the same services in the cloud, instead of providing them on premises (BUYYA et al., 2009). Providers like Amazon, Google, Microsoft, IBM, among others, have established data centers around the world in order to provide cloud services.

Virtualization is one of the cloud computing pillars, allowing providers to maximize their hardware processing power, offering elasticity and scalability of resources. There are different virtualization approaches that offer similar functionality, but with different levels of abstraction and methods. Virtual Machines (VMs) are one of the most used methods. VMs work with hardware emulation through a component called hypervisor, that offers isolation and it is responsible to execute different kernels or operating systems (OSs) under the same physical hardware. Example of solutions that work with hypervisors are VMWare (VMWARE, 2018), KVM (KVM, 2018), Xen (XEN, 2018) and Microsoft Hyper-V (HYPER-V, 2018). Nevertheless, in order to provide security and isolation of process and applications, this type of virtualization has a high performance impact.

Another method, in which computational resources are allocated more efficiently, is called virtualization by containers (JOY, 2015). Containers are a lightweight version of the OS, not having the VMs overload. Containers do not emulate I/O and offer isolation, portability, reduced initialization time and better performance compared to VMs (LI; KANSO, 2015). In order to provide isolation, containers use Linux OS functionalities like namespaces, control groups and chroot (LI; KANSO; GHERBI, 2015)(FELTER et al., 2015). As examples of container solutions we can mention OpenVZ (OPENVZ, 2018), LXC (LXC, 2018), Rkt (RKT, 2018) and Docker (DOCKER, 2018). The last is a good example of container platform that has been widely used in the market (RIGHTSCALE, 2016). Containers include a complete environment including application, libraries and configuration, making deployment much easier. As containers are a lightweight version of the OS, one physical server can execute more containers instances than VMs.

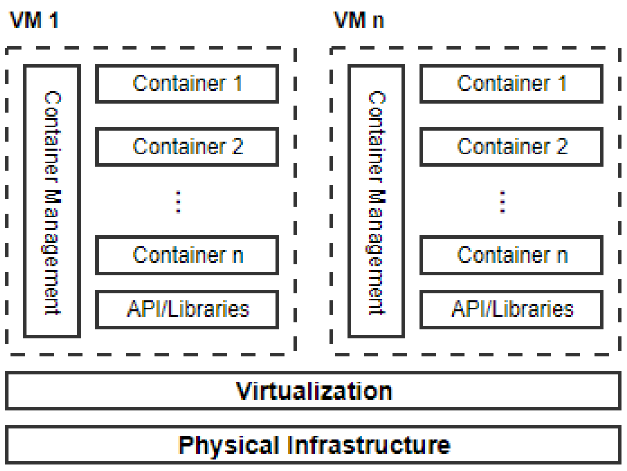
With the adoption of containers in the data center, a new type of service model, called CaaS (Containers as a Service) has appeared (PIRAGHAJ et al., 2015). This service complements the existing IaaS (Infrastructure as a Service), PaaS (Platform as a Service) and SaaS (Software as a Service)(MELL; GRANCE et al., 2011) service models. Figure 1 details how the cloud service models have evolved over time:

Figure 1: Service models in a cloud environment.



PaaS makes extensive use of VMs, although virtualization by VMs imposes an overhead regarding resource consumption for applications that demand elasticity (LI; KANSO; GHERBI, 2015). Figure 2 details the new CaaS service model, in which we have containers running inside VMs. This model has been adopted in data centers as offers VMs security and containers scalability.

Figure 2: Container as a Service (CaaS).



Real-time live migration is a virtualization feature that allows a VM or container image to be transferred from one equipment to another, keeping the active processes running (ELSAID; MEINEL, 2014)(CELESTI et al., 2010)(ASHINO; NAKAE, 2012). VM migration involves the transfer of larger files, making the VM migration process costly, from the network consumption perspective and platform down time. As container images are smaller, the migration process does not have the same cost.

VM and container migration can be used to better distribute the load depending on CPU utilization and network traffic. Migration and load balancing are also important in order to consolidate running applications in less servers during periods of low utilization (green computing)(KANSAL; CHANA, 2012). Another motivation is to allow enterprises to freely choose the best cloud provider depending on a series of factors, like cost, availability or better SLAs. In this scenario, enterprises could choose to live migrate VMs or containers from one cloud provider to another (hybrid cloud) (LINTHICUM, 2016).

Let us define host as any physical server in a data center environment and Virtual Element (VE) as any virtual machine or container. The VE concept will be used in this document to make our approach generic.

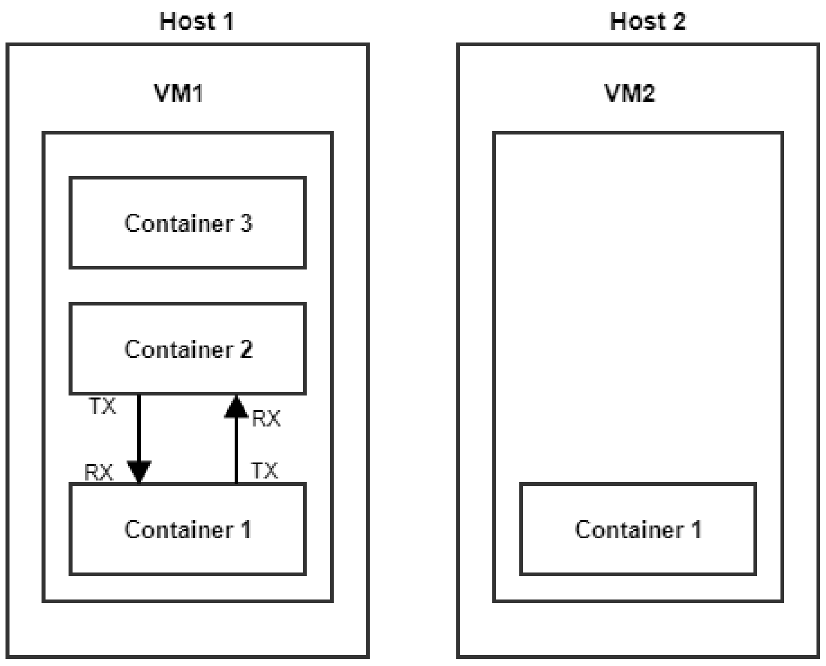
Problem and Objective

One of the problems for data centers and cloud providers is to better distribute the work load between available hosts. A better distribution will reduce power consumption and provide better allocation of available resources. The objective of this dissertation is to present a heuristic that improves the live migration decision process of VEs taking into consideration not only CPU usage, but also network data traffic.

Figure 3 will help clarifying the problem and the objective of this work. Let us suppose that we have two hosts in one rack. Each host has one VM running, defined here as VM1 and VM2 respectively. VM2 has one container running that consumes very low CPU. Therefore, host 2 is under-loaded. VM1 is running 3 containers. Container 1 and container 2 have network traffic between each other. Suppose that host 1 becomes overloaded as its CPU goes above a specified threshold. The problem here is to decide which of the 3 containers should be migrated to host 2 in order to improve load balancing. One of the existing migration models uses Pearson correlation coefficient (PEARSON, 1895) monitoring only CPU consumption to make a decision (PIRAGHAJ et al., 2015). Consider that container 2 is the one that is consuming more CPU at host 1. Based on this model, container 2 should be migrated. However, if we migrate container 2 to host 2, we will improve load balancing, but, additional traffic in the network will be generated. Our proposed approach is to apply a heuristic that takes also in consideration network traffic in the decision process, using Pearson correlation coefficient as well. It is important to take into consideration network traffic, as tests have shown that power consumption raises with the increase in network traffic (MORABITO, 2015).

The main idea is to avoid migration of VEs that have data traffic inside the same

Figure 3: Decision process example.



host. If we migrate a VE that has network activity inside the same host, we may improve overall CPU utilization, but we will increase network traffic between hosts. An architecture was implemented to validate the assertiveness of the heuristic. It is not the intent of this dissertation to discuss how live migration of VEs is implemented for the different solutions available. Our focus is on the decision process itself.

The contributions of this dissertation to the scientific community are:

- Apply the Pearson correlation coefficient (PEARSON, 1895) with CPU and network traffic during the migration decision process. Piraghaj et al. (PIRAGHAJ et al., 2015) monitored only at CPU consumption. We extend this concept looking at network traffic as well;

- Perform live migration of VEs that have low correlation coefficient for internal network traffic and high correlation for CPU usage. Main objective is to better distribute the work load at the data center taking in consideration both CPU and network traffic;

- Define a heuristic as foundation to the decision migration process. The heuristic score depends on the Pearson correlation coefficient for CPU and network traffic. Depending on the calculated score, the decision of migrating the VE is taken or not.

Document Structure

The remaining of this dissertation is organized as follows:

- Chapter 2: it presents the theoretical foundation and related works that are the basis for this project;
- Chapter 3: it describes the Pearson correlation coefficient and how it relates with this project. Furthermore, a heuristic is defined and its application is described in the decision process;
- Chapter 4: it details the architecture developed to validate the heuristic and all its elements. A description of the implementation is explained;
- Chapter 5: it describes the test methodology and its respective results;
- Conclusion: it contains our final remarks and future works that can be developed based on this project.

2 Related Works

This chapter describes the works that are related to this project. The first Section provides basic concepts that are important for the understanding of the scope of the dissertation. The following Section details work that is relevant and that served as motivation to the development of the presented solution.

2.1 Basic Concepts

This Section describes basic concepts of the cloud environment and the available cloud service models, like IaaS, PaaS, SaaS and CaaS. The VM and container virtualization approach are defined here and how these technologies have been used in data centers. Live migration is a fundamental characteristic that will be discussed as well. The next Section will expand the concepts presented here.

Buyya et al. (BUYYA et al., 2009) defines cloud computing as:

> A type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computer resource(s) based on service level agreements established through negotiation between the service provider and consumers.

Enterprises are interested in the cloud model as it represent a cost reduction compared to the on-premises model, in which companies need to invest in all the infrastructure. Buyya et al. (BUYYA et al., 2009) show how infrastructure based on virtualization of resources is utilized to offer scalability and elasticity.

The National Institute of Standards and Technology (NIST) (MELL; GRANCE et al., 2011) define five essential characteristics for the cloud model:

- On-demand self-service: consumer can provision computing resources as needed;
- Broad network access: capabilities are available over the network;
- Resource pooling: computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical or virtual resources allocated on demand. There is a sense of location independence, i.e, the customer has no control or knowledge over the exact location of the provided resources;
- Rapid elasticity: capabilities can be elastically provisioned and released, in some cases automatically, depending on demand. To the consumer, the available capabilities appear to be unlimited and can be appropriated in any quantity at any time;

- Measured service: resource usage can be monitored, controlled and reported, providing transparency for the cloud provider and to the consumer.

Vaquero et al. (VAQUERO et al., 2008) associated cloud to a new way to provide computing infrastructure. Cloud paradigm shifts infrastructure to the network in order to reduce management costs with hardware and software. The authors defined cloud as a large pool of usable and accessible virtualized resources that can be dynamically adjusted to a variable load. Resources are charged in a pay-per-use model that guarantees an agreed SLA. IaaS is defined as a large set of computing resources, like storage and processing capacity, that, through virtualization, can be dynamically offered to companies to run their services. PaaS is an additional abstraction level in which the platform is offered to run customer services. Yet, SaaS is an alternative to run applications, using services that are hosted in the cloud.

According to the National Institute of Standards and Technology (NIST) (MELL; GRANCE et al., 2011) the service models in a cloud environment are:

- Software as a Service (SaaS): the capability offered to consumers to use provider's application, running on a cloud infrastructure. The consumer does not manage or control the underlying cloud infrastructure, with the exception of some application configuration;

- Platform as a Service (PaaS): the capability provided to the consumer to deploy onto the cloud infrastructure consumer's applications. The consumer does not manage or control the underlying cloud infrastructure, but has control over the deployed application;

- Infrastructure as a Service (IaaS): the capability provided to the consumer to provision processing, storage, networks, and other computing resources where the consumer is able to deploy and run software, which can include OSs and applications. The consumer does not manage or control the underlying cloud infrastructure, but has control over OS, storage and deployed application.

NIST also defines the possible deployment models:

- Private Cloud: cloud infrastructure provisioned exclusively by a single organization to their multiple consumers (business units). It may be owned, managed and operated by the organization, third party or a combination of them. It can be on-premises or off-premises;

- Public Cloud: cloud infrastructure provisioned for open use by general consumers. It may be owned, managed and operated by a business, academy, government or a combination of them. It exists on the premises of the cloud provider;

- Hybrid Cloud: cloud infrastructure is a composition of two or more distinct cloud infrastructures (private or public) that remain unique entities. There is a standardized or proprietary technology that enables data and application portability.

Clark et al. (CLARK et al., 2005) mentioned that a VM encapsulate access to a set of physical resources. Amaral et al. (AMARAL et al., 2015) defined virtualization as a technology that allows the data center to offer on-demand and elastic resources. VMs have been widely used in traditional data centers and cloud environments (private, public and hybrid clouds), nevertheless, the interest in container technology has increased as containers are lightweight and fast. Containers provide less overhead compared to VM because they do not emulate a full physical hardware virtualization. Containers use Linux cgroups and namespaces to provide isolation of processes and file systems. Containers biggest advantage is the capability to run a copy of the Linux OS without running a hypervisor. The authors also discussed a new trend called microservices. A system can be developed in small sets, called microservices, that can be developed, managed and scaled independently. As microservices are a small set of the whole system, they fit very well with the concept of the lightweight containers.

Live migration is a powerful feature in a virtualized data center that can be used for load balancing, reduce power consumption, disaster recovery, among others. Elsaid & Meinel (ELSAID; MEINEL, 2014) studied the VM live migration impact considering network resources and power consumption.

Regarding performance, Felter et al. (FELTER et al., 2015) developed a comparison between VMs and containers. Their final results showed that containers have an equal or better performance in almost all proposed scenarios. The authors also detailed how containers implement isolation.

As far as the Pearson correlation coefficient (PEARSON, 1895) is concerned, Filho & Júnior (FILHO; JÚNIOR, 2009) described that the origin of this coefficient came from the research of Karl Pearson and Francis Galton (STANTON, 2001). According to the authors, Pearson correlation coefficient r is a measure of the linear association between variables. In statistical terms, two variables are associated when they have similarities in their score distribution. The linear model assumes that the increase or decrease of one unity in variable X, generates the same impact in Y. In our study, we have used Pearson correlation coefficient in order to verify if the CPU and network traffic of a VE is associated with the CPU and network traffic of a host system. All the details about the Pearson correlation coefficient will be presented in Section 3.

2.2 Related Works

There are several studies regarding the use of VMs in a cloud environment. The interest in containers has increased due to the benefits provided by this architecture. The combination of VMs and containers, concept called as CaaS, is a trend as it provides VM security and containers performance (RIGHTSCALE, 2016). Live migration of VMs and containers is an important feature for load balancing and high availability in a data center.

Bernstein (BERNSTEIN, 2014) mentioned that hypervisors and containers are part of the existing cloud environment. He details also the existence of a concept called bare metal cloud, in which the same characteristics of cloud technology are offered, but only in physical servers. The author observed that hypervisor based deployments are ideal when applications require different OSs on the same cloud. As containers share an OS, their deployment is much smaller in size compared to VMs, and its restart time, much quicker. Due to this characteristic, a container deployment can host hundreds of containers compared to VM deployments. Bernstein approached Docker (DOCKER, 2018), an open source project, as a container technology that extends LXC containers (LXC, 2018) in order to provide faster Linux application development. Docker containers are created using base images. A base image can consist of just the OS or pre-built applications. The author commented about benchmarks that showed that containers are much faster than VMs, but also discussed that many deployments have decided for a hybrid implementation, with containers and VMs. One example of management solution that provides full VM and container integration is Proxmox VE (PROXMOX, 2018). Proxmox is a management platform with full integration with KVM hypervisor (KVM, 2018) and LXC containers.

Microservices is a technology trend that has leverage the container architecture. A system can be developed in small sets called microservices that can be managed and scaled independently. Amaral et al. (AMARAL et al., 2015) analyzed performance of microservices in this environment. The cost of using microservices with containers is the computational overhead of running an application in different processes, and the increase in network traffic between containers. The increase in network traffic between containers is an issue that this work will address. Authors mentioned Kubernetes (KUBERNETES, 2018), an open source container cluster management solution developed by Google, that provides load balancing and failure management for containers. The objective of Kubernetes is to make the management of large number of microservices easier. Docker announced that has incorporated native support to Docker Swarm (DOCKERSWARM, 2018) and Kubernetes in its platform. Docker Swarm is also a cluster management solution provided by Docker. Another container management system is called LXD (LXD, 2018). LXD is a layer on top of LXC that provides a REST API to manage containers and offers to the user the same experience as VMs. It is possible to run all flavors of containers inside LXD.

Energy efficiency is one driver to the study of live migration and consolidation of

VMs and containers. Piraghaj & Buyya (PIRAGHAJ et al., 2015) developed a framework for container consolidation in a CaaS environment, in order to get a more energy efficient data center. The authors developed an algorithm that is invoked when the host CPU reaches a specific threshold. Then, they used the Pearson correlation coefficient (PEARSON, 1895), based on CPU consumption, to decide if a container should be migrated or not. The article analyzed the correlation between host and container CPU. Our work take this concept as a reference, and adds network traffic at the live migration decision process. As the objective of the authors were to reduce power consumption in the data center, they used live migration to consolidate the work load in the smaller number of VMs as possible, and consequently, the smaller number of physical hosts. If there were hosts with no VMs and containers running after the migrations, the framework powered off these hosts. From the other side, if the framework detected that there were not enough VMs to run the processes, it invoked a module that creates more VM instances.

Still in the energy efficiency field, Morabito (MORABITO, 2015) compared power consumption of VM and containers. While both technologies present similar power consumption in idle state and also during CPU/memory stress test, during network tests VM showed a higher energy consumption compared to containers due to the fact that network packets need to be processed by extra layers in hypervisor environments. Network tests were done using iperf Linux tool (IPERF, 2018). The work developed by Morabito showed the importance of considering network traffic during load balancing in order to avoid unnecessary resource consumption.

Kansal & Chana (KANSAL; CHANA, 2012) described the importance that migration, server consolidation and load balance have to green computing. The authors discussed existing load balancing techniques for cloud computing. They defined green computing as the implementation of policies and procedures that improve the efficiency of computing resources, reducing energy consumption and environmental impact.

Vigliotti & Batista (VIGLIOTTI; BATISTA, 2014) divided the VM allocation problem in two. The first, is the creation and placement of a new VM on a host, and the second, is optimization of the current VM allocation. The authors focused in the first problem. Our work has targeted the second problem, i.e, optimizing current VE distribution. The objective of the authors were to make the data center more energy efficient, minimizing the number of physical hosts used and maximizing the number of VMs per host. Computers are more efficient when they are operating near 100% of their capacity. Energy efficient VM allocation take advantage that idle hosts can be suspended in order to reduce power consumption and, in case that workload increases, they can be reactivated. Authors have used Knapsack and Evolutionary Computation strategies in order to solve this problem. In this same topic, Zhang et al. (ZHANG et al., 2013) discussed relationship based VM placement. The authors mentioned the importance to evaluate how VMs communicate before deciding where to place them. For instance, if two VMs that communicate frequently are put in different physical servers, the CPU allocation criteria can be met, but can provide poor performance due to the frequent communication occurred between the VMs. They have developed a framework for VM placement that considers the software packages running on each VM. Based on the software packages, the framework make assumptions on how the VMs communicate, and decided where is the best place to run them.

Real time migration is an important feature available, not only in hypervisors, but also in containers, and can be used for load balancing, reduction in power consumption, disaster recovery, among others. Elsaid & Meinel (ELSAID; MEINEL, 2014) studied the overhead of VM live migration in the overall data center performance. Through live migration, a running VM is transferred from one physical server to another, with little interruption. Therefore, the study of resource consumption during VM live migration needs to be considered. The costs associated with live migration are: migration time and migration down time, overhead in power and CPU utilization and network bandwidth consumption. The authors described the VM live migration process using VMotion, a solution from VMWare (VMWARE, 2018), and created models to calculate the time to migrate and power consumption during the process. With those models, an estimation of live migration performance was done. Elsaid & Meinel have a concern regarding network bandwidth consumption, an issue that our work will address.

VM image size to be transferred during the migration is also a concern. Celesti et al. (CELESTI et al., 2010) explored the VM live migration process and described the concern of transferring GBs of data over the network. Authors focused on how to reduce the VM image size to be transferred during migration.

Ashino & Nakae (ASHINO; NAKAE, 2012) analyzed VM migration to different hypervisor implementations. The authors reinforced the importance that VM migration is taken in cloud solutions, and that a VM migration method, between different hypervisor solutions, will be required. Current live migration process, when the destination is a different hypervisor, involves converting the VM image and the transfer of larger image files. VM image conversion can create boot problems at destination. The authors proposed a different migration method, that was destination dependent. With this approach, VM image was in general 20% smaller than the regular image.

Linthicum (LINTHICUM, 2016) mentioned that current approaches for PaaS and IaaS create a platform lock-in, making the migration from one cloud provider to another very difficult. Containers is a efficient way to create workload that can be transferred from cloud to cloud due to its portability. Container live migration will remove the cloud provider lock-in, allowing customers to live migrate their containers from one provider to another based on better SLAs, costs or resource allocation. He described the need of having an orchestration tool to perform the live migration and monitor the whole process. Companies, like Jelastic (JELASTIC, 2018), are implementing this type of orchestration tool.

As container adoption has increased, discussions about high availability (HA) and, consequently, real time migration has appeared. Romero & Hacker (ROMERO; HACKER, 2011) were one of the precursors in the study of container real time migration for parallel applications, especially with OpenVZ (OPENVZ, 2018). The authors decided to work with containers, compared to VMs, due to its lightweight characteristics. With containers, the amount of data to be dumped, moved and restarted is much smaller than compared to VMs. This characteristic will make the process quicker, what is very important when we talk about parallel applications.

Li & Kanso (LI; KANSO, 2015) compare VM and containers for achieving HA. One important aspect that the authors approached is a comparison between stateless and statefull containers. The first can have their state replicated in a storage and be deployed behind load balancers. Failures are transparent, as the load balancer will send traffic to healthy containers. Nevertheless, for statefull containers, in which the state of the application and the network stack needs to be maintained, the live migration approach is a viable alternative. VM and containers live migration is implemented through the checkpoint/restore capability. CPU compatibility is also required to ensure that a VM can perform normally at destination host after migration. OpenVZ (OPENVZ, 2018) container technology was one of the first to implement the chekckpoint/restore functionality, however it was implemented as loadable modules in the OpenVZ kernel. The lack of integration with native Linux kernel reduced the adoption of OpenVZ. To solve this situation, CRIU (CRIU, 2018) project was created, moving most of the checkpoint capability outside the kernel, into user space. With CRIU, one can freeze a running application and save it as a set of files in disk. Later these files can be used to restore the application and start it exactly from the point where it was frozen. Docker has adopted CRIU as a solution for checkpoint/restore. Currently this functionality is only available in Docker in experimental mode. Later it will available for production environments. Still in the HA topic, Kanso has developed another work with Gherbi (KANSO; HUANG; GHERBI, 2016), discussing the use of Kubernetes to manage containarized applications across multiple hosts. The authors mentioned that containers have been adopted to accelerate the development and operation of microservices, as microservices, in general, are loosely coupled and have independent life cycles and deployments. They have tested the effectiveness of Kubernetes (KUBERNETES, 2018) to manage HA. Containers are grouped in what is called a pod. Kubernetes manages the availability of the pods, restarting a container just in seconds, controlling where the pod will run. The authors concluded that Kubernetes is a solution capable of managing stateless containers in a HA environment. The same authors extends the discussion of containers and HA in another article (LI; KANSO; GHERBI, 2015).

The comparison between containers and VMs is also done by Xavier et al. (XAVIER et al., 2013) in a High Performance Computing (HPC) environment. The authors mentioned that the use of virtualization in HPC was avoided due to the performance overhead. However, with the enhancements on container virtualization, solutions like LXC (LXC, 2018) and OpenVZ (OPENVZ, 2018) have been adopted in HPC environments. Techniques like live migration, checkpoint and resume are important in such environments.

Celesti et al. (CELESTI et al., 2010) defined cloud federation as an environment in which VMs can be migrated from a cloud to another. Their objective is to improve VM migration in such environment. VM migration implies the transfer of the VM image, causing consumption of network bandwidth and cloud resources. They proposed a mechanism that reduced the amount of data to be transferred. Each VM has a disk image, in a given hypervisor format, containing the file system and guest OS. There are two types of VM migration: hot (live) and cold. The biggest difference is that in hot migration the VM does not loose its status, and users does not notice any change, while in cold, users notice a service interruption. Downtime is defined as the time difference when a VM is turned it off in the source, and turned it on in the destination. In a hot migration the downtime is negligible. The authors proposed a combination of pre-copy and post-copy techniques to reduce down time. VM migration is a very expensive process, implying the migration of GBs of data.

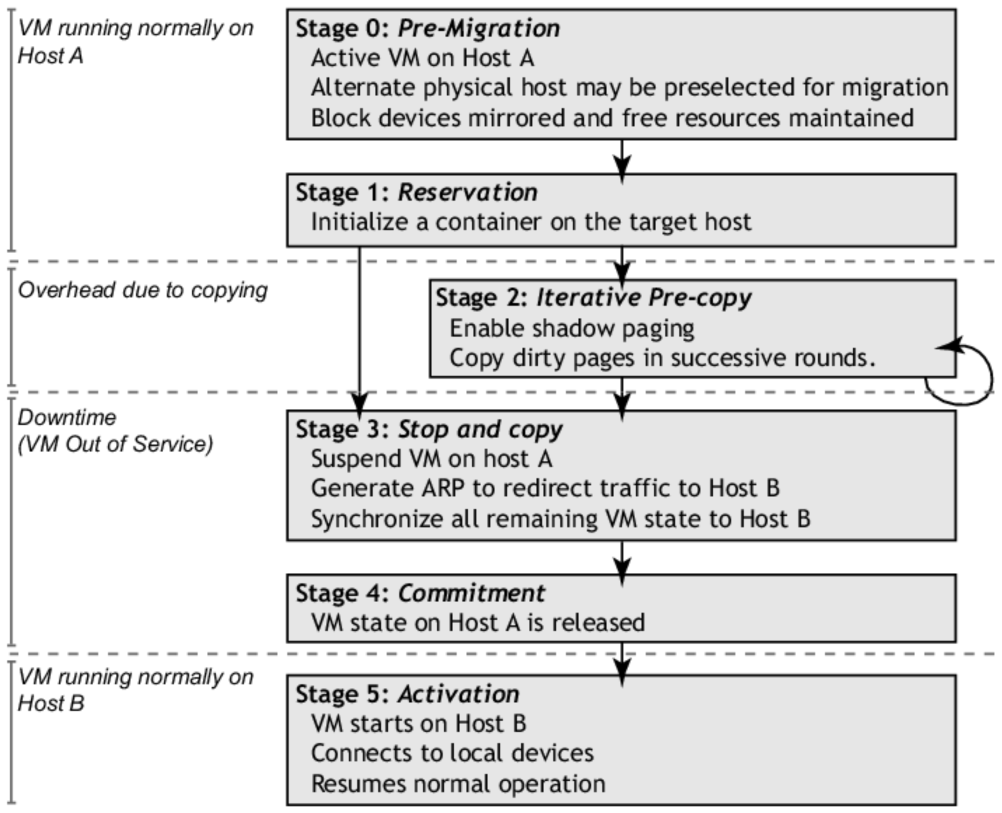
Clark et al. (CLARK et al., 2005) described the logical steps to migrate a VM. They emphasize that migration is a powerful tool for cluster administration, and analyzed ways to live migrate VMs, reducing downtime and total migration time. For the authors, migrating a VM consists of transferring its memory image from source to destination server. Figure 4 represents the migration time-line and required steps in order to perform the VM migration.

Voorsluys & Buyya (VOORSLUYS et al., 2009) discussed the cost of live migrating a VM, and mentioned the cold and hot live migration approaches in order to perform this task. Popular hypervisors work with hot migration in order to reduce down time, instead of the cold migration, that do a stop-and-copy mechanism.

Synytsky (SYNYTSKY, 2016) described use cases for container live migration: hardware maintenance without downtime, load re-balance, HA within data centers and change of cloud vendor. The author mentioned some bottlenecks to container live migration, like applications with big amounts of data and fast changing data. Latency and data volume may be blockers to successful live migration. Figure 5 describes how container live migration works.

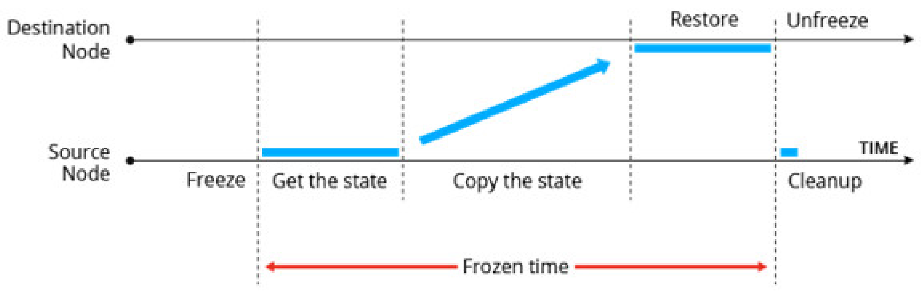
Figure 5 shows the steps to perform the container migration. First, the platform freezes the container at the source node, gets its state and blocks memory, processes, file system and network connections. After the freeze process is complete, all files are copied

Figure 4: VM Migration Phases.


VM running normally on Host A
Stage 0: Pre-Migration
Active VM on Host A
Alternate physical host may be preselected for migration
Block devices mirrored and free resources maintained
Stage 1: Reservation
Initialize a container on the target host
Overhead due to copying
Stage 2: Iterative Pre-copy
Enable shadow paging
Copy dirty pages in successive rounds.
Downtime
(VM Out of Service)
Stage 3: Stop and copy
Suspend VM on host A
Generate ARP to redirect traffic to Host B
Synchronize all remaining VM state to Host B
Stage 4: Commitment
VM state on Host A is released
VM running normally on Host B
Stage 5: Activation
VM starts on Host B
Connects to local devices
Resumes normal operation

Source: Clark et al. (2005, p. 5)

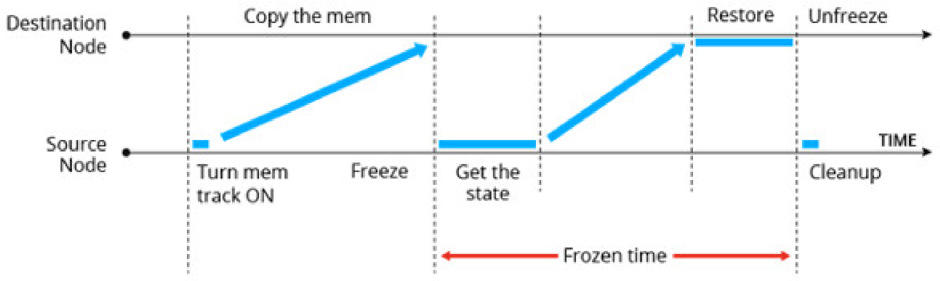
Figure 5: Container Migration Phases.



Source: Synytsky (2016, p. 3)

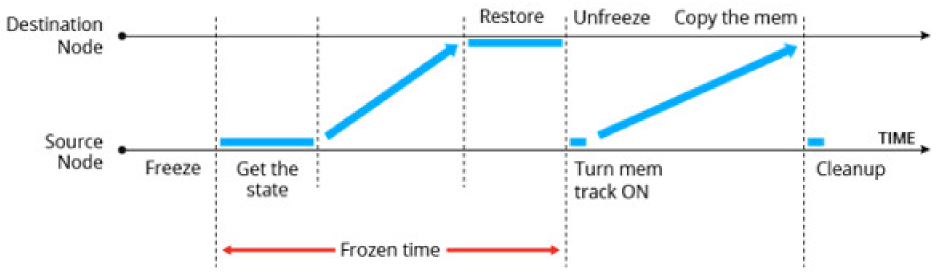
to the destination node. The destination platform restores the state and unfreezes the container. Then, there is a quick cleanup process at the source node. Note that there is a frozen time period during the migration process. The objective is to reduce this interval of time. There are two approaches, according to the author, in order to provide reduced frozen times, one is called pre-copy memory (Figure 6) and the other is post-copy memory (Figure 7).

Figure 6: Container Migration Pre-Copy Memory.



Source: Synytsky (2016, p. 4)

Figure 7: Container Migration Post-Copy Memory.



Source: Synytsky (2016, p. 4)

In the pre-copy memory, during the migration process, memory from the source node is transferred to the destination until it reaches a minimum. At this point the container is frozen and the remaining data, including its state, is transferred to the destinations and the container is restored. While post-copy memory, or lazy migration, the container is frozen, the initial state is got, the fastest changing memory pages are transferred to destination and restore it. The rest of the state is copied to the destination in background mode.

Kalim et al. (KALIM et al., 2013) focus on the issue of maintaining network connection state following a live VM migration beyond a subnet. Hypervisors rely on Reverse Address Resolution Protocol (RARP) to maintain a network state following a migration within a subnet. Migration beyond subnets is a challenge, as IP addresses of the network interface normally change, generating a disruption of the service. If applications does not implement re-connections, migration results in disconnections. They proposed a backward compatible extension for TCP/IP, which decouples the naming of endpoints from the naming of the flows. The decoupling guarantees that a change in the IP address does not impact the connection states, as connections will be identified by a label instead of the IP address.

Current network protocols impose significant management overhead in large data centers. Mysore et al. (MYSORE et al., 2009) proposed PortLand, a set of Ethernet compatible routing, forwarding and address resolution protocol to make management easier. They mentioned that usually data centers topology are inter connected as multi-rooted tree. Based on that, Portland employs a lightweight protocol to enable switches to discover their position in the topology. Portland uses Pseudo Mac Addresses (PMAC) to encode their position and suggests a fat tree topology. One aspect that motivated the study of server location in the data center was VM migration.

As far as monitoring is concerned, Grozev & Buyya (GROZEV; BUYYA, 2016) approached an interesting aspect, in which they do not only monitor CPU and memory, but also which VM configuration and type is more adequate for an specific application. The authors proposed a dynamic method for selecting VMs to be migrated, using machine learning techniques. In the proposed environment, the auto-scaling monitor module receives information every 5 seconds from the VMs. As far as CPU is concerned, they have collected information from the /proc/cpuinfo Linux kernel file. The objective of the auto-scaling module is to provide the ideal VM type for application servers.

The related work analysis reveals a vast study about the VM usage in the cloud environment and the growing relevance that containers are getting (RIGHTSCALE, 2016). The usage of both, VM and containers, in the data center, is a solution that has shown benefits from the security standpoint and from scalability and elasticity. Real time migration is an important feature, available not only in hypervisors but also in containers, and can be used for load balancing, reduction in power consumption, disaster recovery, between others. VM solutions, like VMWare (VMWARE, 2018) and Xen (XEN, 2018), provide live migration mechanisms. Container live migration has evolved with the implementation of the checkpoint/restore functionality through CRIU (CRIU, 2018).

This dissertation defines a heuristic that use Pearson correlation not only for CPU, but includes network traffic in the migration decision process. None of the works above mentioned takes into account CPU and network traffic at the same time. Piraghaj & Buyya (PIRAGHAJ et al., 2015) have considered only Pearson correlation coefficient for CPU in order to make a decision of which container should be migrated. Elsaid & Meinel (ELSAID; MEINEL, 2014) have focused in monitoring and modeling VM live migration, not including containers in the discussion. This project will prolong this concept including containers.

3 Pearson Correlation Model

In this Section, we describe the Pearson correlation coefficient and the heuristic developed to assist during the migration decision process. The objective of the heuristic is to determine if a VE is a good candidate to be migrated or not, considering CPU and data network traffic. By definition, a heuristic is designed to find an approximate solution, making the decision process quicker compared to complex methods that look for an exact solution.

The work of Filho & Júnior (FILHO; JÚNIOR, 2009) was the basis to explain the most important concepts of the Pearson Correlation model presented below.

3.1 Pearson Correlation Coefficient

The Pearson correlation coefficient (PEARSON, 1895) estimates the dependency level between quantities. If we consider n samples of two variables x and y, represented here for x_i e y_i, the Pearson correlation coefficient is calculated by Equation 3.1, in which $\bar{x}$ and $\bar{y}$ represent the arithmetic mean of x and y, respectively. The coefficient r_{xy} ranges from [-1,+1] (PIRAGHAJ et al., 2015). The signal indicates the positive or negative relationship direction between variables, and the value, the strength of the relationship. When the coefficient has a value of -1 or +1, it is called a perfect correlation. Otherwise, when the coefficient has a value of 0, it means that there is no linear relation between variables.

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{3.1}$$

The closer the coefficient is to 1, the more dependent the variables are. When variables are dependent, they have greater probability to reach peak/valley together. Consider, for example, a VE called X and a host system Y. If the CPU correlation coefficient between X and Y is close to +1, means that CPU of VE X is contributing for a CPU peak at host Y.

Filho & Júnior (FILHO; JÚNIOR, 2009) described that the origin of the Pearson coefficient comes from the research of Karl Pearson and Francis Galton (STANTON, 2001). According to the authors, Pearson correlation coefficient r is a measure of the linear association between variables. The authors also mentioned two concepts that are important to understand the coefficient: association and linearity. In statistical terms, two variables are associated when they have similarities in their score distribution. The coefficient is a measure of shared variance between variables. The linear model assume that the increase or decrease of one unity in variable X generates the same impact in Y. Therefore, Pearson correlation coefficient requires shared variance, and that this variation should be distributed linearly.

In order to interpret the coefficient at this dissertation, we considered the study developed by Christine & John (CHRISTINE; JOHN, 2004) that adopted the classification defined at Table 1 for the coefficient r. During the interpretation of the results in Chapter 5, we will make reference to this classification. The level of association is important to our work because it will define when the CPU or network traffic of a VE is strongly associated with the host. The objective is to migrate VEs that have CPU strongly associated with the host and network traffic weakly associated.

Table 1: Pearson correlation coefficient score levels.

Coefficient Score r	**Classification**
0.0	no relation
0.1 to 0.3	weak
0.4 to 0.6	moderate
0.7 to 1.0	strong

Source: (CHRISTINE; JOHN, 2004)

The closer the score gets to 1 (independent of the signal), the greater is the linear statistical dependency degree between variables. In the opposite side, the closer to 0, the lesser is the relationship strength.

According to Filho & Júnior (FILHO; JÚNIOR, 2009) the following observations can be done that are relevant for our work:

1. Pearson correlation coefficient does not differ between dependent and independent variables. Therefore, the correlation between X and Y is the same between Y and X;
2. The correlation value (score) does not change if you change the measurement unity of the variables. As an example, if you have a variable in kilograms, the score will be the same if you use grams;
3. The coefficient is dimensionless, i.e, it does not have a unity that defines it. If you have a coefficient with value $r = 0.4$, it cannot be interpreted as 40%, for example. Also it can not be interpreted as been twice as strong than $r = 0.2$;
4. Correlation requires that variables are quantitative;
5. The observations should be independent.

Chen & Popovich (CHEN; POPOVICH, 2002) estimated that Pearson correlation coefficient, and its derivations, are chosen 95% of the time to describe relationship patterns between variables.

3.2 Proposed Heuristic Model

We are addressing the problem of deciding which VE to migrate based on CPU consumption and network traffic. Given that we are collecting data from the CPU consumption and network traffic over time, we can use the Pearson correlation coefficient presented on Section 3.1 in order to analyze if the VE resource consumption has a strong relationship with the host consumption.

Table 2 has a list of the symbols that will be used in this section with their respective definition.

Table 2: Symbol Definitions.

Symbol	**Definition**
r_{CPU}	Pearson coefficient based on CPU
$r_{network}$	Pearson coefficient for network (TX+RX)
r_{TX}	Pearson coefficient for TX data traffic
r_{RX}	Pearson coefficient for RX data traffic
h_{TX}	heuristic based on r_{CPU} and r_{TX}
h_{RX}	heuristic based on r_{CPU} and r_{RX}
h_{Tot}	proposed heuristic $= h_{TX} + h_{RX}$
h_{CPU}	heuristic based on CPU $= r_{CPU}$
T_{CPU}	threshold for CPU consumption
TP_{CPU}	threshold for r_{CPU}
$T_{heuristic}$	threshold for h_{Tot}

CPU is an important variable in our analysis. The migration decision process starts when the CPU level reaches an established threshold. The main objective is to better load balance the workload, distributing the VEs accordingly, considering CPU and network traffic. Therefore, in our evaluation, CPU will have a higher level of importance and this will be considered while applying the heuristic. As far as the network traffic is concerned, there are two components that should be taken in consideration. The transmitted traffic (TX) and the received traffic (RX), between VEs located in the same host. As far as network traffic, the purpose is to avoid migration of VEs that exchange data with other VEs inside the same host. If we migrate such VEs, we may get an improvement in the CPU allocation, but the network traffic between hosts will increase, consuming extra network resources. The work of Morabito (MORABITO, 2015) showed the importance of considering network traffic during load balancing in order to avoid unnecessary resource consumption. Also, the extensive use of microservices technology, will create more data exchange between VEs, increasing the importance of considering network traffic during the migration process. (AMARAL et al., 2015).

There are three variables to take into consideration during the migration decision process: CPU, TX and RX. Therefore, a heuristic was defined to make the decision process easier, and also to include the premise that CPU should have a higher weight. The heuristic was developed based on the Pearson correlation coefficient and takes into consideration the level of correlation between the host CPU and the VEs running inside the host. Furthermore, looks at the network traffic between VEs inside the host. The objective is to migrate a VE that has high correlation with host CPU but low correlation for data traffic between VEs inside the host. Therefore, we avoid migrating VEs that generate high data traffic inside the host. If we migrate such VEs, we can better distribute the load, but generate higher network data traffic between hosts. The heuristic is defined in equation 3.2:

$$h_{heuristic} = \frac{1 - (weight * r_{CPU})}{(1 + (weight * r_{CPU}) - (weight * r_{network}))} \tag{3.2}$$

During simulations, the weight equals to 0.5 showed good results according to the premise of moving high correlated CPU and low correlated network traffic. Therefore, we decided to use 0.5 as the weight factor.

For network traffic, it is necessary to consider TX and RX between VEs in the same host (Figure 3). To make the calculation process simpler, we decided to treat RX and TX separately, and then, add the results. Two variables were defined, h_{TX} to calculate the Pearson correlation coefficient for TX traffic (equation 3.3) and h_{RX} for RX traffic (equation 3.4).

$$h_{TX} = \frac{1 - (0.5 * r_{CPU})}{(1 + (0.5 * r_{CPU}) - (0.5 * r_{TX}))} \tag{3.3}$$

$$h_{RX} = \frac{1 - (0.5 * r_{CPU})}{(1 + (0.5 * r_{CPU}) - (0.5 * r_{RX}))} \tag{3.4}$$

At the end of the process, both calculations are added in order to generate the heuristic h_{Tot} (equation 3.5).

$$h_{Tot} = h_{RX} + h_{TX} \tag{3.5}$$

The heuristic h_{Tot} places a higher weight for the CPU correlation coefficient compared to the network traffic correlation. The reason is that we only want to migrate VEs from an overloaded host, i.e, a host with high CPU and low network correlation. In order to analyze the heuristic behavior, let us look at some scenarios for h_{TX} presented at Table 3.

Table 3 shows the behaviour of the heuristic at certain coefficient values. According to Filho & Júnior (FILHO; JÚNIOR, 2009), it is unusual to find correlation coefficients in the extremes (0 or 1). However, for our purpose, it is interesting to analyze the extremes in order to verify how the heuristic behaves. Remembering that when the correlation

Table 3: Heuristic scores for h_{TX}.

r_{CPU}	r_{TX}	h_{TX}
0.0	0.0	1.00
0.0	1.0	2.00
0.3	0.3	0.85
0.6	0.6	0.70
0.7	0.0	0.48
0.7	0.6	0.62
0.7	0.7	0.65
1.0	0.0	0.33
1.0	1.0	0.50

Source: Equation 3.3

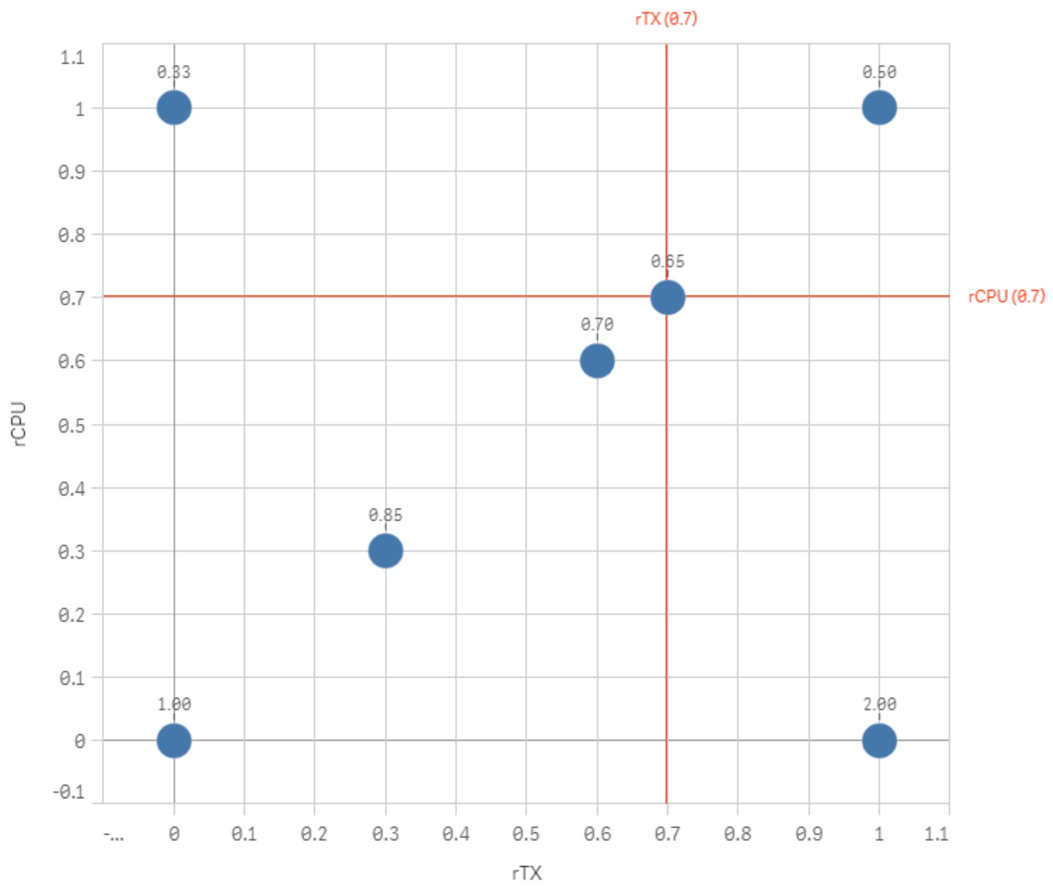
coefficient is equal to 0, there is no association between the variables, and when the coefficient is equal to 1, we have a perfect correlation. We also considered coefficients equal to 0.3 (weak association), 0.6 (moderate association) and 0.7 (strong association) (CHRISTINE; JOHN, 2004). For strong associations, we also analyzed some variances in the correlation coefficient for TX network traffic (r_{TX}).

Let us assume that for a particular VE, the Pearson correlation coefficient for the CPU, here called r_{CPU}, is equal or close to 0. In other words, there is no association between the CPU consumption of the VE and the host. In this scenario, there are two alternatives regarding network traffic, one with strong relationship between VE and host ($r_{TX} >= 0.7$) and the other without relationship. In the first case, if we assume a perfect correlation ($r_{TX} = 1.0$), then the score of h_{TX} is equal to 2.00 and for the last case, considering $r_{TX} = 0.0$, the score for h_{TX} is equal to 1.00. If we calculate the score of h_{TX} with weak correlations ($r_{CPU} = 0.3$ and $r_{TX} = 0.3$), we will get $h_{TX} = 0.85$. For moderate correlations ($r_{CPU} = 0.6$ and $r_{TX} = 0.6$) the score of h_{TX} will be 0.70. Let us look some scenarios in which the coefficient represent strong relationship. If we look when r_{CPU} is equal to 0.7 (strong) and $r_{TX} = 0.6$ (moderate) we got a score of 0.62 for h_{TX}. Now if we look when r_{CPU} is equal to 0.7 (strong) and $r_{TX} = 0.7$ (strong), the score of h_{TX} is equal to 0.65. In the case of perfect association ($r_{CPU} = 1$) and no association with data traffic ($r_{TX} = 0.0$), we will have a score of 0.33 for h_{TX}. When there is association with data traffic ($r_{TX} = 1.0$), h_{TX} will be equal to 0.50. When we compare the heuristic with the model based on CPU only (PIRAGHAJ et al., 2015), in which migration happens when $r_{CPU} >= 0.70$, the score of h_{TX} will be equal to 0.48.

Figure 8 represents the values present in Table 3.

In the y axis we have r_{CPU}, and in the x axis we have r_{TX}. The value above the plotted point is the score for h_{TX}. Two reference lines were drawn as well, showing when the Pearson correlation coefficient is equal to 0.70 (strong correlation). Notice that the intersection of r_{CPU} and r_{TX}, when the coefficients are equal to 0.70, provides a score of $h_{TX} = 0.65$. Looking at Figure 8, we notice that heuristic scores greater than 0.65 represent,

Figure 8: Heuristic h_{TX}



in most of the cases, VEs that should not be migrated. Therefore, we assume 0.65 as a threshold for h_{TX}, meaning that values lesser or equal to 0.65 should be considered for migration. For our test cases, results showed that the heuristic is a good approximation for most of the cases.

The same reasoning applies exactly to h_{RX}. Therefore, the threshold for h_{RX} is 0.65 as well.

As defined by equation 3.5, h_{Tot} is equal to the sum of h_{TX} and h_{RX}. Then, the threshold $T_{heuristic}$ for h_{Tot} is equal to 0.65 plus 0.65, resulting in a score of 1.30. If h_{Tot} is greater than 1.30, the module should continue with the monitoring process without any migration. Otherwise, if h_{Tot} score is less or equal to 1.30, the VE is a candidate to be migrated. Equation 3.6 summarizes what was just described:

$$Heuristic = \begin{cases} h_{Tot} <= T_{heuristic} \text{ - migration} \\ h_{Tot} > T_{heuristic} \text{ - do nothing} \end{cases} \tag{3.6}$$

The heuristic h_{Tot} can be fine tuned, through changing the *weight* from equation 3.2 or changing the value of $T_{heuristic}$. Notice that if you reduce the value of the threshold

$T_{heuristic}$, the model will be more conservative, migrating less VEs. If you increase the threshold, the model will more flexible, migrating more VEs. Therefore, we can change the value of $T_{heuristic}$ to tune our model. For our tests, we will consider the threshold equals to 1.30.

3.3 Heuristic based on CPU

For comparison purposes, let us introduce a heuristic called h_{CPU}, that will be equal to the Pearson correlation coefficient based on CPU only, r_{CPU} (Equation 3.7). As mentioned in Chapter 2, Piraghaj & Buyya (PIRAGHAJ et al., 2015) considered migration of containers when r_{CPU} reaches a specific threshold. We will use the migration results based on the score of h_{CPU} to compare with the migrations based on h_{Tot}.

$$h_{CPU} = r_{CPU} \tag{3.7}$$

Let us define here another threshold, called TP_{CPU}, which establishes a threshold for r_{CPU}, and consequently, for h_{CPU}. Comparing the highest coefficient of h_{CPU} with TP_{CPU}, will confirm if the CPU from the VE has relationship with the CPU usage from host. If h_{CPU} is greater or equal than TP_{CPU}, then there is a relationship between the host CPU and the CPU consumption of the VE. Equation 3.8 shows the decision process.

$$hCPU = \begin{cases} h_{CPU} < TP_{CPU} \text{ weak relationship - do nothing} \\ h_{CPU} >= TP_{CPU} \text{ strong relationship - migrate} \end{cases} \tag{3.8}$$

For strong relationship, we set up TP_{CPU} as 0.70 (CHRISTINE; JOHN, 2004).

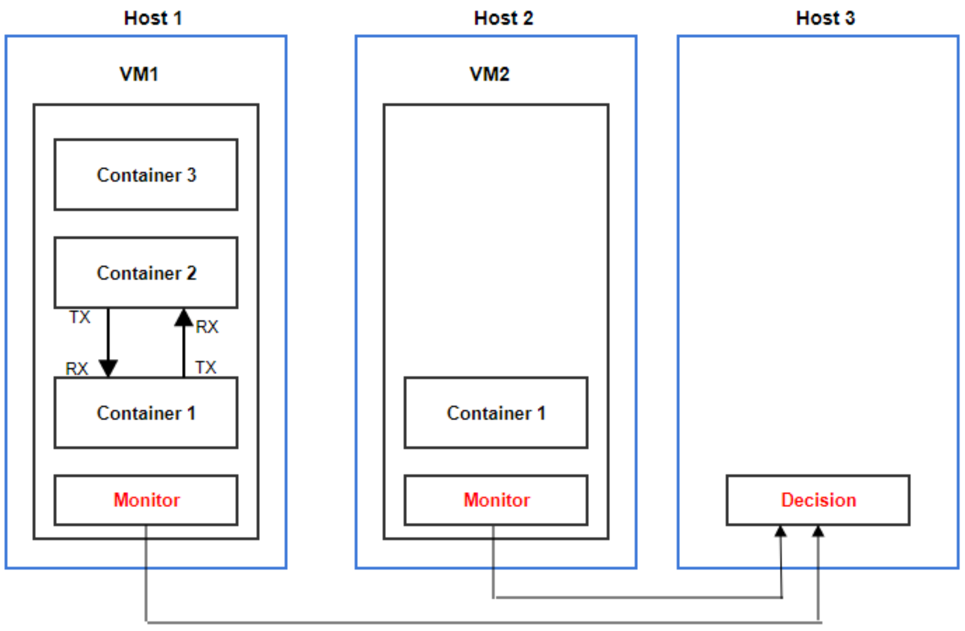
4 Architecture and Implementation

This Chapter details the architecture used to validate the heuristic. It also describes how we implemented it. This implementation will be the basis for the tests accomplished in the next Chapter.

4.1 Decision/Monitoring Architecture

The architecture was developed to validate the proposed heuristic. It is generic in the sense that can support hosts running VMs, hosts running containers or a hybrid environment. By hybrid we consider an environment with hosts running VMs, and inside the VMs, one or more containers (CaaS model). Figure 9 represents the proposed architecture and also the two main modules developed: the Decision and Monitoring modules.

Figure 9: Decision/Monitoring Scenario.

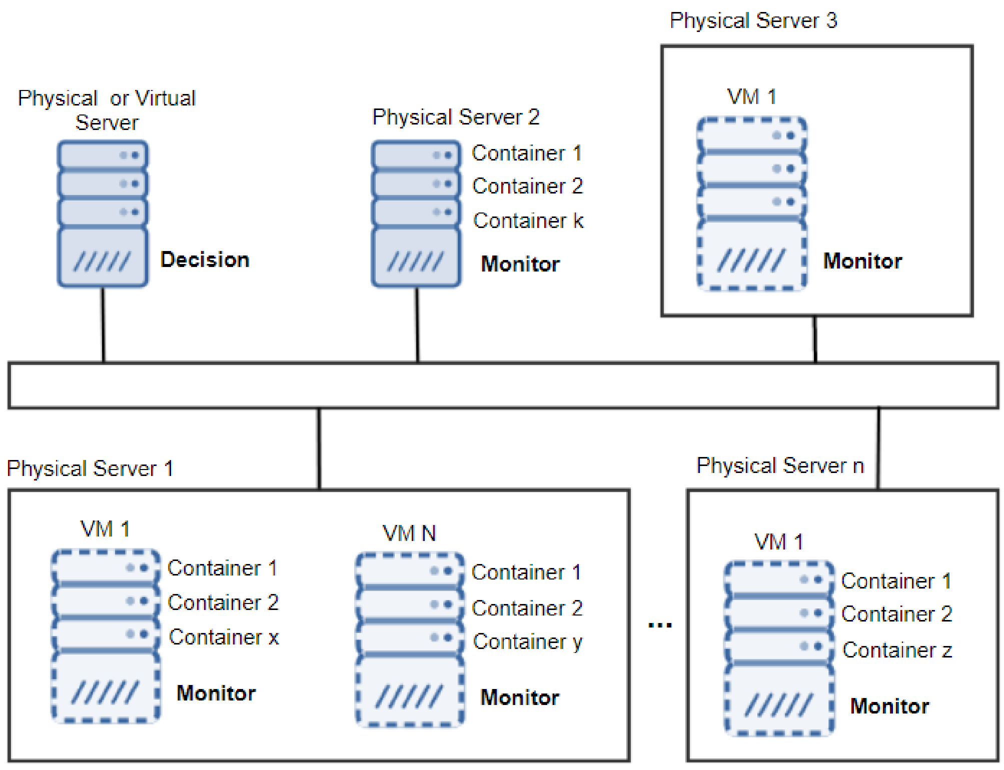


Yet, Figure 10 shows an instantiation of the architecture with different types of configurations:

Below a description of the main modules:

a. **Monitor**: it is responsible for collecting CPU and network data traffic statistics from host and VEs. The collected data is sent periodically to the Decision Module. This module also implements the migration task;

Figure 10: Decision/Monitoring Instantiation.



b. **Decision**: it receives statistics from all hosts running the Monitor Module. This module stores the data so it can be used to calculate the Pearson correlation coefficient and apply the heuristic. When a particular host is overloaded, exceeding the established threshold, the heuristic is applied to check if there is a VE candidate to be migrated to an under-loaded host.

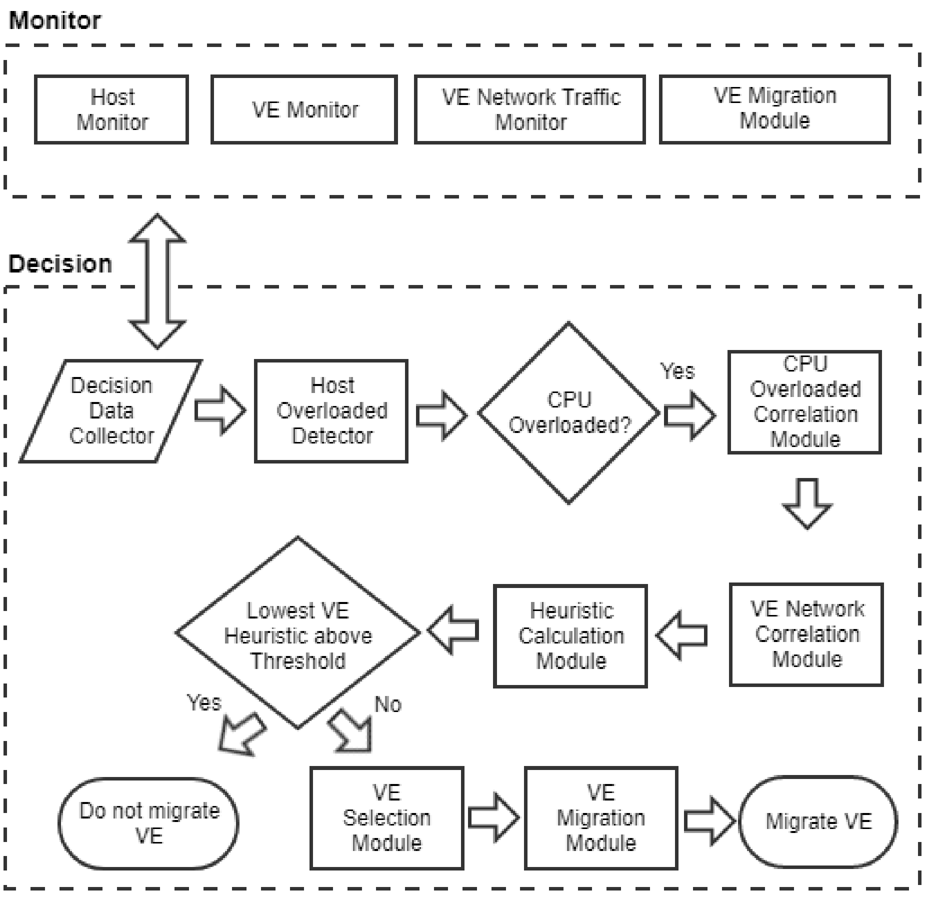
Figure 11 depicts a flowchart of the main modules and tasks executed by the Monitor and Decision modules. Following a description of each task by module.

A. *Monitor Module*

1. Host Monitor: it collects statistics from a host;
2. VE Monitor: it collects statistics from VEs in the host;
3. VE Network Traffic Monitor: it is responsible for collecting network traffic statistics between VEs inside the same host. It ignores the traffic that goes out;
4. VE Migration Module: it executes the commands to perform VE migration.

As far as the VE Network Traffic Monitor is concerned, it is important to describe with more details how the process work. First, this task only cares for network traffic generated by VEs inside the same host. We want avoid migrating a VE

Figure 11: Decision/Monitor Flowchart.



that exchanges data with another VE inside the same host. Migrating such VE, would generate additional external traffic in the network between hosts. As Morabito (MORABITO, 2015) described, containers, and especially VMs, consume more power when submitted to higher network data exchange. Tests were done using iperf (IPERF, 2018), and the author noticed that one of the reasons for the increase in power consumption, was the time that packets need to be processed by extra layers in a hypervisor environment. Amaral et al. (AMARAL et al., 2015) mentioned that the development using microservices will increase network traffic between VEs, as the multiple microservices from an application will need to exchange data among them. Therefore, the VE Network Traffic Monitor module, collects all internal TX and RX traffic generated by VEs, and then, calculates the Pearson correlation coefficients for each VE. Based on the coefficients, we know if the network traffic of a VE is strongly associated with the internal traffic. If a host has a VE that exchanges data with another VE that is located in another host, migrating any of these VEs will not affect negatively the overall network performance, because this traffic is already there in the environment. As an example, if we look at Figure 9, if we migrate container 1 from VM1 to VM2, the network traffic will increase as the internal traffic now will occur over the external network. However, if container 3 from VM1 exchanges data with container 1 of VM2, and if we transfer container 3 to host3, nothing will change in the network performance, as this traffic is already happening over the network.

The VE Migration Module executes two different tasks, one at source host and the other at destination. At source, receives the ID of the VE to be migrated and the destination host. The module executes the necessary commands to checkpoint the VE. At destination, when the VE image transfer is completed, the VE restore process takes place.

B. Decision Module

1. Decision Data Collector: it receives statistics from all hosts running the Monitor module. It stores the data so it can be later used during the Pearson correlation coefficient calculation process and to decide which host is most under-loaded to receive a migrated VE;
2. Host Overloaded Detector: after data is received by the Decision Data Collector, the CPU utilization level from the host, CPU_{Host}, is compared to a static threshold, called here T_{CPU}. If CPU_{Host} is less than T_{CPU}, then the host system is under-loaded, otherwise, the host is overloaded. For our simulation we have set T_{CPU} in 70%, the same threshold value used by Piraghaj & Buyya (PIRAGHAJ et al., 2015). Equation 4.1 details the comparison:

$$HostCPU = \begin{cases} CPU_{Host} < T_{CPU} \text{ - under-loaded} \\ CPU_{Host} >= T_{CPU} \text{ - overloaded} \end{cases} \tag{4.1}$$

3. CPU Overloaded Correlation Module: in case that the host CPU is overloaded, the Overloaded Correlation Module is responsible for calculating the Pearson correlation coefficient between the CPU usage of every VE running in the host, and the CPU of the overloaded host (Equation 3.1). As a result, we will have a coefficient, called here r_{CPU}, for every VE in the host. Let us define another threshold, called TP_{CPU}, which establishes a static threshold for r_{CPU}. Comparing the highest coefficient of r_{CPU} from the host with TP_{CPU}, will confirm if the CPU from the VE has relationship with the CPU usage of the host. Equation 4.2 summarizes the comparison. If r_{CPU} is greater or equal than TP_{CPU}, then, there is a relationship between the host CPU and the CPU usage of the VE. This module will also store the value of r_{CPU} for every VE, that later will be used when applying the heuristic. Likewise, this module will create a log, registering which VE should be migrated considering r_{CPU} only. It is important to save this information, as we will use these results in comparison to the results generated by the proposed heuristic. Algorithm 1 implements similar

logic presented by Piraghaj & Buyya (PIRAGHAJ et al., 2015) that considers migration of containers when r_{CPU} reaches the threshold TP_{CPU}. If the highest r_{CPU} is less or equal than the static threshold TP_{CPU}, than the CPU usage from the VE is not related to the CPU host consumption. Otherwise, if r_{CPU} is greater than TP_{CPU}, then the CPU utilization of the VE is related to the high CPU usage in the host. For our purpose, we will consider TP_{CPU} greater than 0.70 as a strong correlation (CHRISTINE; JOHN, 2004). In Section 3.3 we have defined the heuristic h_{CPU}, that is equal to r_{CPU}.

$$PearsonCPU = \begin{cases} r_{CPU} < TP_{CPU} \text{ - weak relationship} \\ r_{CPU} >= TP_{CPU} \text{ - strong relationship} \end{cases} \tag{4.2}$$

Algorithm 1 VE Migration Selector based on $r_{CPU} >= TP_{CPU}$

```
1: Input: $VE\_Hash, Host, CPU_{Host}$
2: Output: $Host_{dest}$, $VE.MigrateID$
3: if $CPU_{Host} >= T_{CPU}$ then
4:     $VEList \leftarrow host.getVEList(VE\_Hash)$
5:     for all $VEs$ in $VEList$ do
6:         $rCPU \leftarrow$ Pearson($CPU_{Host}$,$CPU_{VE}$)
7:     end for
8:     $VEMigr.r_{CPU} \leftarrow get.HigherPearson(r_{CPU})$
9:     $VE.MigrateID \leftarrow get.MigrationID.HigherPearson(r_{CPU})$
10:    if $VEMigr.r_{CPU} >= TP_{CPU}$ then
11:        $Host_{dest} \leftarrow host.getLowestCpuUtilization(VE\_Hash)$
12:        Send Migration Manager ($VE.MigrateID$,$Host_{dest}$)
13:    else
14:        No migration.
15:        continue;
16:    end if
17: end if
```

Algorithm 1 receives as input a hash table with all VEs and their respective statistics per host. As output, we will have the destination host and the ID of the VE to be migrated. At line 3, we compare if the CPU of the host is overloaded, i.e, if the CPU consumption is greater or equal to the static threshold T_{CPU}. If it is overloaded, at line 4, VEList receives a list of the VEs for the host. Lines 5 and 6 do the Pearson's correlation coefficient calculation for each VE in the host. Line 8 detects which VE has the highest r_{CPU}, and, in line 9, gets the ID of this VE. In line 10, it compares if the highest r_{CPU} is greater or equal to the static threshold TP_{CPU}. If it is greater, then this VE should be migrated. At line 11, retrieves from the hast table, the host most under-loaded to be the one to receive the migrated VE. In line 12, send to the Migration Manager of the source host, the ID of the VE and the host destination in order to have the migration done. If r_{CPU} is less than TP_{CPU}, no migration is required. Notice that, during this process, the network data traffic is ignored.

4. VE Network Correlation Module: This module retrieves network traffic statistics between running VEs from the overloaded host, and the objective is to verify if there is traffic between them. If internal traffic is high correlated, migration should be avoided. For the TX traffic of the VE, this module calculates the Pearson correlation coefficient r_{TX} between the VE and all internal TX traffic at host. The same logic is applied to RX, and the coefficient is called r_{RX}. If we compare the Pearson correlation coefficient to a threshold called $TP_{network}$, we can evaluate if the network traffic of a VE has a strong relationship with the host or not. For our purpose we will consider $TP_{network}$ greater than 0.70 as a strong correlation (CHRISTINE; JOHN, 2004). Equations 4.3 and 4.4 summarize the comparison.

$$PearsonTX = \begin{cases} r_{TX} < TP_{network} \text{ - weak relationship} \\ r_{TX} >= TP_{network} \text{ - strong relationship} \end{cases} \tag{4.3}$$

$$PearsonRX = \begin{cases} r_{RX} < TP_{network} \text{ - weak relationship} \\ r_{RX} >= TP_{network} \text{ - strong relationship} \end{cases} \tag{4.4}$$

5. Heuristic Module: previous modules have calculated the Pearson correlation coefficient for CPU and network, here called r_{CPU}, r_{TX} and r_{RX} respectively. With those values in hand, we can apply the heuristics for h_{TX}, h_{RX} and h_{Tot} and get their respectively scores (Equations 3.3, 3.4 and 3.5). The heuristic score should be compared to a static threshold, called here $T_{heuristic}$. If the lowest score for h_{Tot} is less or equal than the static threshold $T_{heuristic}$, then the VE is the one that should be migrated. Otherwise, if h_{Tot} score is greater than $T_{heuristic}$, then the VE should not be migrated as there is considerably internal traffic between VEs inside the same host. A heuristic is an approximation of the exact solution. Note that if $T_{heuristic}$ is defined too low, the model will be very restrictive, allowing few migrations. Otherwise, if defined too high, the model will be very flexible, allowing more migrations. As defined in Section 3.2, we will work with $T_{heuristic}$ equals to 1.30. Equations 4.5 summarizes the explanation.

$$Heuristic = \begin{cases} h_{Tot} <= T_{heuristic} \text{ - migrate} \\ h_{Tot} > T_{heuristic} \text{ - do not migrate} \end{cases} \tag{4.5}$$

6. VE Selection Module: the Destination Selection module evaluates to which host should the VE be migrated. It verifies, in the list of available hosts, which one

has the lowest CPU utilization and selects it. This module also verifies if the CPU of the destination host can handle the CPU of the source VE, avoiding generating a CPU usage greater than 100% at destination. If the CPU total exceeds, a log is generated indicating that it may be necessary to start new VMs in the environment in order to allow a better distribution of the resources. Therefore, if any data center environment has most of its servers above 70% utilization, there will not be enough available resources for migration. Equation 4.6 details the calculation.

$$CPU = \begin{cases} CPU_{Destin} + VE.CPU_{Source} <= 100\% \text{ - migrate} \\ CPU_{Destin} + VE.CPU_{Source} > 100\% \text{ - do not migrate} \end{cases} \tag{4.6}$$

7. VE Migration Module: The VE Migration task is responsible to send a message to the Monitor module of the overloaded host with the VE that should be migrated and the host destination. The VE Migration Module executes two different tasks, one at source host and the other at destination. The first task is executed at source after receiving the ID of the VE to be migrated and the destination host. This task executes the necessary commands to checkpoint the VE and transfer the images to destination. The second task, performed at destination, restores the VE image just after the VE image transfer is completed.

Algorithm 2 represents the decision process defined in the Decision module. It complements the sequence of tasks detailed in Figure 11 - Decision/Monitoring Flowchart. Below, a short description of the algorithm.

Algorithm 2 receives as input a hash table with all VEs and their respective statistics per host, the host ID and its CPU level. As output, we will have the destination host ID and the ID of the VE to be migrated. When the CPU utilization of a host H, called here CPU_{Host}, is greater or equal than the established threshold T_{CPU}, the decision process gets started (line 3). The Decision module retrieves from the hash all running VEs from host H (line 4). Also it requests network traffic statistics from the Traffic Monitor (line 5) and calculates the total internal traffic generated by VEs for TX and RX inside host H (line 6 and 7). Following, it calculates the Pearson correlation coefficient for CPU, TX and RX for every container at host H (lines 9 to 11). In the sequence, the heuristics is applied and the scores calculated (lines 12 to 14). At the end of the process, the lowest score of h_{Tot} is compared to the static threshold $T_{heuristic}$ (line 16-18). If the score is lower or equal than the threshold, the decision is to migrate the VE, sending its ID and host destination ID to the Migration task (line 20). Otherwise the decision is not to migrate the VE (lines 22 and 23). The destination host is based on the lowest host CPU utilization (line 19).

Algorithm 2 VE Migration Selector based on Heuristic h_{Tot}

```
 1: Input: $VE\_Hash, Host, CPU_{Host}$
 2: Output: $Host_{dest}$, $VE.MigrateID$
 3: if $CPU_{Host} >= T_{CPU}$ then
 4:     $VEList \leftarrow host.getVEList(VE\_Hash)$
 5:     request VE data traffic to Traffic Monitor
 6:     $TX_{Tot} \leftarrow sum.TX.allInternalVEs$
 7:     $RX_{Tot} \leftarrow sum.RX.allInternalVEs$
 8:     for all $VEs$ in $VEList$ do
 9:         $rCPU \leftarrow$ Pearson($CPU_{Host}$,$CPU_{VE}$)
10:         $rTX \leftarrow$ Pearson($TX_{Tot}$,$TX_{VE}$)
11:         $rRX \leftarrow$ Pearson($RX_{Tot}$,$RX_{VE}$)
12:         $h_{TX} \leftarrow$ Heuristic($r_{TX}$,$r_{CPU}$)
13:         $h_{RX} \leftarrow$ Heuristic($r_{RX}$,$r_{CPU}$)
14:         $h_{Tot} \leftarrow h_{TX} + h_{RX}$
15:     end for
16:     $VEMigr.h_{Tot} \leftarrow get.LowerHeuristic(h_{Tot})$
17:     $VE.MigrateID \leftarrow get.MigrationID.LowerHeuristicID(h_{Tot})$
18:     if $VEMigr.h_{Tot} <= T_{heuristic}$ then
19:         $Host_{dest} \leftarrow host.getLowestCpuUtilization(VE\_Hash)$
20:         Send Migration Manager ($VE.MigrateID$,$Host_{dest}$)
21:     else
22:         No migration.
23:         continue;
24:     end if
25: end if
```

4.2 Implementation

In order to verify the proposed solution, an implementation was done in Java. This implementation will be used in Chapter 5, when the tests and results will be discussed. Docker was selected as the container technology and was executed in experimental mode in order to have the checkpoint/restore functionality enabled. Docker has been a technology widely used by the enterprise market (RIGHTSCALE, 2016)(BERNSTEIN, 2014).

Let us detail how the architecture was implemented looking at the flowchart defined in Figure 11 - Decision/Monitor Flowchart.

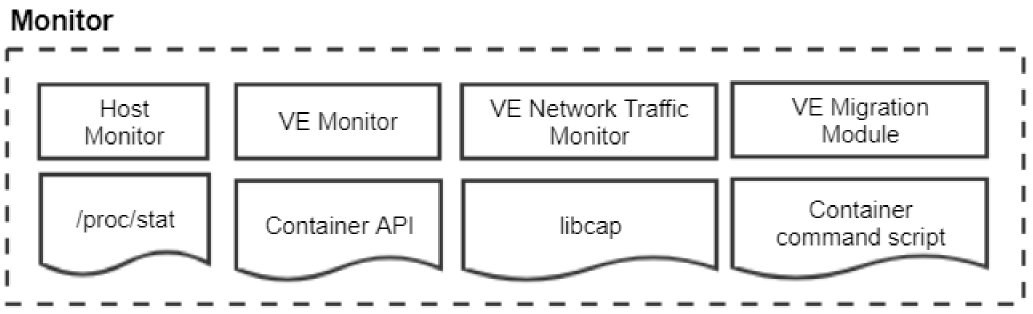
Monitor Module

This module is responsible for collecting data to send to the Decision module. Figure 12 details how the architecture was instantiated considering Docker technology:

Below the details of how each of the modules was developed.

- Host Monitor: This module is responsible to read the /proc/stat file and calculate the percentage of CPU usage from host. The proc file system is a pseudo-file system

Figure 12: Monitor Module - Docker Implementation.



which provides an interface to kernel data structures. In the stat file we find an aggregate entry for CPU with information that allows us to calculate the percentage of CPU utilized. Below, a typical entry for CPU at /proc/stat:

```
cpu 10132153 290696 3084719 46828483 16683 0 25195 0 175628 0
```

The fields, that come in sequence after the word CPU, are (MANPAGE, 2018):

- user: time spent in user mode;
- nice: time spent in user mode with low priority;
- system: time spent in system mode;
- idle: time spent in the idle task;
- iowait: time waiting for I/O to complete;
- irq: time servicing interrupts;
- softirq: time servicing softirqs;
- steal: stolen time, which is the time spent in other OS when running in a virtualized environment;
- guest: time spent running a virtual CPU for guest OS under the control of the Linux kernel;
- guestNice: time spent running a niced guest (virtual CPU for guest OS under the control of the Linux kernel).

We can now calculate the idle and nonIdle CPU consumption for a time interval between $t-1$ and t. The percentage is calculated between these two time intervals.

$$CPU\% = \begin{cases} idle_t = idle_t + iowait_t \\ nonIdle_t = user_t + nice_t + system_t + irq_t + softirq_t + steal_t \\ total_t = idle_t + nonIdle_t \\ total_\Delta = total_t - total_{t-1} \\ idle_\Delta = idle_t - idle_{t-1} \\ CPU_\% = ((total_\Delta - idle_\Delta)/total_\Delta) * 100 \end{cases} \tag{4.7}$$

The fields guest and guestNice are not included in the calculation as they are already counted in the nice and user fields (KERNEL, 2018).

The Monitor module executes this routine every second in order to calculate CPU percentage consumption in the host and send it to the Decision module.

- Virtual Element Monitor: The VE monitor is responsible for getting statistical data from the monitored VE. In order to collect data, we use the Docker API (DOCKER, 2017) to interact with the Docker Engine API. This last is a REST API accessed by an HTTP client, such as wget or curl. The daemon listens on unix:///var/run/docker.sock to allow only local connections by the root user. The API call returns a JSON message that can be parsed to extract the required information. The API will continuously report a live stream of CPU, memory, I/O and network metrics. The difference is that the API provides far more details than the docker stats command.

  For Docker version 1.13, we have used API version 1.25. Below, a message example of the REST API call GET.

  ```
  http://localhost:4243/containers/json?all=0&size=1
  ```

  The response is a long JSON with metrics about the container. Instead of printing the complete JSON response, we have broken the message in parts that are relevant to this work. The result of the GET command presents general information about the containers. The API response for the GET call is presented at Appendix A, Section A.1.

  In order to get all necessary statistical data from the containers, we used the following REST call:

  ```
  http://localhost:4243/containers/<containerID>/stats?stream=0
  ```

where <containerID> is the ID of the container that we want to monitor.

The response is a long JSON with statistics data about the container. Instead of printing the complete JSON response, we have broken the message in parts that are relevant to this work. The API response for is presented at Appendix A, Section A.2.

As far as the network metrics for containers is concerned, the Docker API is the easiest way to get such information. The API returns TX and RX traffic, number of packets and also information of I/O activity, that it is not shown here, as it is not part of the scope of project.

Docker Stats command:

Docker Stats command displays a live stream of container(s) resource usage statistics. Our objective is to calculate the container CPU usage similarly to the command, but using the Docker API. Below an example of the docker stats command.

```
$ docker stats

CONTAINER ID        NAME                CPU %
b95a83497c91        awesome_brattain    0.28%
67b2525d8ad1        foobar              0.00%
e5c383697914        test-1951           0.00%
4bda148efbc0        random.1.           0.00%
```

CPU is reported as a percentage of total host capacity. Therefore, if we have two containers, each using as much CPU as they can, the docker stat command for each would register 50% utilization, though, in practice, their CPU resources would be fully utilized. According to the docker client documentation, the CPU percentage is calculated as follows (DOCKERSTATS, 2016):

$$containerCPU\% = \begin{cases} system_{\Delta} = CPUSystem_{t} - CPUSystem_{t-1} \\ cpu_{\Delta} = CPUContainer_{t} - CPUContainer_{t-1} \\ CPU_{\%} = (cpu_{\Delta}/system_{\Delta}) * numCores * 100 \end{cases} \tag{4.8}$$

The variables are all extracted from the Docker API call. As CPU_{Δ} is the total time consumed by all cores, and $system_{\Delta}$ is also the total time consumed by all cores, then $(cpu_{\Delta}/\ system_{\Delta})$ is the average CPU usage of each core. Therefore, it is required the multiplication by the number of CPU cores in order to calculate the total CPU usage.

- Virtual Element Traffic Monitor: In order to monitor data traffic, the libcap library was used. TCPDUMP (TCPDUMP, 2018), one of the most popular tool for capturing and analyzing packet traces, works with the same approach. According to Stevens et al. (STEVENS; FENNER; RUDOFF, 2004, page 792), libpcap provides implementation independent access to the underlying packet capture facility provided by the OS.

  This module is divided in two sub-modules:

  1. Data Collection: it is the only module not developed in Java. It is a C program that uses the libcap library to monitor the network bridge or overlay created by Docker. It monitors the packet exchange between containers inside the same host. This module creates a file that stores records that have the IP source, IP destination, total payload, elapsed time and a timestamp. This file will be read by the sub-module presented below.
  2. Traffic Monitor: when the Decision module requires data traffic information, it sends a message via socket to the VE Traffic Monitor. This sub-module reads the file created with data traffic information and sends it back to the Decision module.

  The Data Collection module implements a loop using the function pcap_loop to monitor all packets from the docker bridge or overlay. Overlay network is a distributed network among multiple Docker daemons. Docker transparently handles routing of each packet to and from the correct Docker daemon host and the correct destination container. The Data Collection module receives as input the name of the bridge and the IP prefix of the container network. The default bridge name for docker is docker0. Algorithm 3 describes this routine.

Algorithm 3 Docker Data Monitor

```
1: Input: $DockerBridge$, $IPMask$
2: Output: FILE $DockerBridge$.log
3: while TRUE do
4:     Monitor Data Packets
5:     $Pcap\_Loop(DockerBridge)$
6:     Compare if packet is internal
7:     if $(Packet \in IPMask)$ then
8:         Write to $DockerBridge$.log
9:         IPOrigin, IPDestination, PayLoad, Timestamp
10:    end if
11: end while
```

Algorithm 3 receives as input the name of the bridge to be monitored, for example docker0, and the IP prefix of the VEs, for example, 172. As output, we have a file with the same name of the monitored bridge with the extension .log. The file will contain records with the following data: IPOrigin, IPDestination, PayLoad, Timestamp (lines 1 and 2). The module will be executing a loop (line 3), monitoring the data packets from the bridge (line 5). If the IP prefix of the monitored packet matches the one passed as input, we should write the data in the file (lines 7 and 8).

Even tough we have a loop using the function pcap_loop, tests have shown that this module did not create CPU overhead.

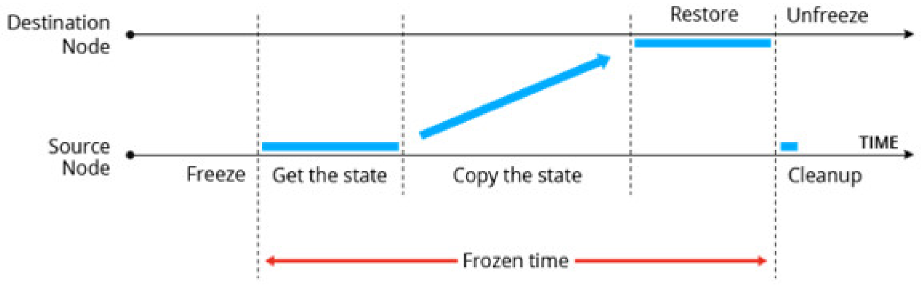
- VE Migration Module: The migration module is also divided in two sub-modules:

  1. Migrate: when the message to start the migration process is received by the VE Migration module in the source host, Algorithm 4 is executed. It is responsible to save the container image, perform the checkpoint and transfer the files to the destination host.
  2. Restore: it monitors at destination host if any file is received from the source host. If it detects new files, it invokes the script defined in Algorithm 5 to perform the container restore process.

Both algorithms will be detailed in the sequence.

The VE migration model follows the stop and copy mechanism for container migration described by Synytsky (SYNYTSKY, 2016) and detailed in Figure 13:

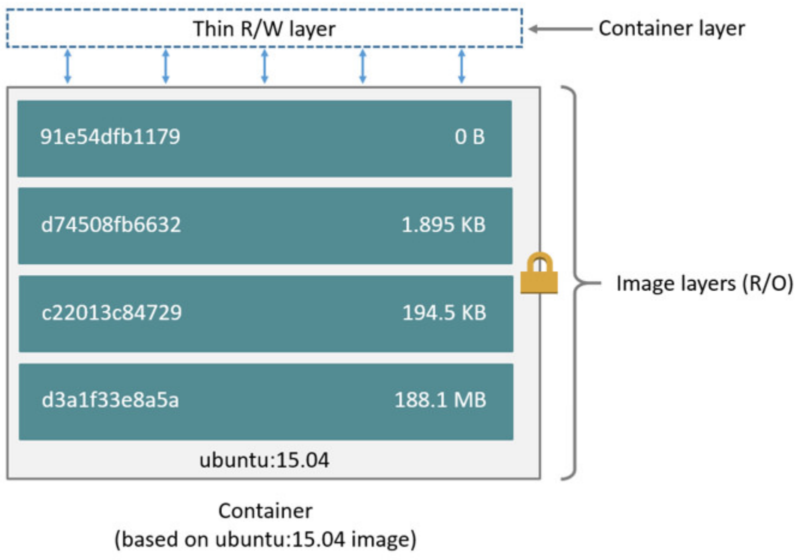
Figure 13: Container Migration Phases



Source: Synytsky (2016, p. 3)

One part of the process is the transfer of the container image and the checkpoint files. In our implementation, it is important to clarify how Docker works. A Docker image is built up from a series of layers. Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The major difference between a container and an image is the top writable layer. All writes to the container that add new, or modify existing data, are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged. Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. Copy-on-write (CoW) is a strategy of sharing and copying files for maximum efficiency. If a file or directory exists in a lower layer within the image, and another layer needs read access to it, it just uses the existing file. The first time another layer needs to modify the file, the file is copied into that layer and modified. This minimizes I/O and the size of each of the subsequent layers (DOCKER, 2018). Figure 14 is an example of how Docker manages an Ubuntu image. There is a container image, that is read only, with the base Ubuntu image and the container layer, which allows read and write. Any changes will be stored in the container layer. Notice that the image size for an Ubuntu image is approximately 190MB.

Figure 14: Container Image and Layers.



Source: https://docs.docker.com/storage/storagedriver/#images-and-layers

Therefore, when we look at Algorithm 4, the script receives as input the container name to be migrated and the destination IP address (line 1). The output will be the elapsed time to perform all the tasks (line 2). With the container name, a docker function called *dockerinspect* can be called in order to get the container image ID (lines 3 and 4). The same command can be used to retrieve the container ID (lines 5 and 6). The next step is to save the image (lines 7 and 8) and then perform the checkpoint (DOCKERCRIU, 2018) to get current container state (lines 9 and 10). During the file transfer, the image is transferred first and then the checkpoint files (lines 11 and 12). Transferring the checkpoint to the destination is not enough, as it is required to have the base image locally in the destination. An optimization could be implemented, if we assume that all base images are installed on the hosts, allowing that we transfer only the checkpoint files. The use of file compression also reduces the size of the files to be transferred, improving the overall migration time. Finally, the process remove the files created for the transfer (line 13) and displays the elapsed time (line 14).

Algorithm 4 Docker Container Migration - Migrate sub-module

```
1: Input: $ContainerName$, $DestinationIP$
2: Output: $TimeElapsed$
3: Finding ImageID
4: $ContainerImageID \leftarrow docker.inspect.Config.Image(ContainerName)$
5: Getting ContainerID
6: $ContainerID \leftarrow docker.inspect.ID(ContainerName)$
7: Saving Docker image
8: $ImageFile \leftarrow docker.save(ContainerImageID)$
9: Checkpointing Container
10: $CheckpointFile \leftarrow docker.checkpoint.create(ContainerName)checkpoint1$
11: Transfer $ImageFile$ to $DestinationIP$
12: Transfer $CheckpointFile$ to $DestinationIP$
13: Remove temporary files
14: Calculate and display elapsed time
```

As far as the Restore process presented at Algorithm 5, the script receives as input the image name (line 1) and as output it displays the elapsed time to perform the task (line 2). There is a process that keeps monitoring if files are transferred to the destination host. When the file transfer is detected, the docker image is loaded (lines 3 and 4). A docker container is created with the same container name (lines 5 and 6) and the container is started from the checkpoint (lines 7 to 9).

Algorithm 5 Docker Container Restore - Restore sub-module

```
1: Input: $ImageName$
2: Output: $TimeElapsed$
3: Load the image
4: $docker.load(ImageName)$
5: Create the container
6: $ContainerID \leftarrow docker.create(ContainerName, ImageName)$
7: Transfer checkpoint files to /var/lib/docker/containers/<$ContainerID$>/
8: Starting container from checkpoint
9: $docker.start.checkpoint(ContainerName)$
10: Remove temporary files
11: Calculate and display elapsed time
```

Decision Module

The Decision model is agnostic and can be used with any type of VE. It expects to receive data from a socket in a specific pattern from the Monitor module. Therefore, any VE, that sends data following this pattern will be interpreted by the Decision Module. The communication with the Traffic Monitor and the Migration module is done through sockets as well.

As mentioned before, the implementation detailed here will be used in the simulations detailed at next chapter.

5 Tests and Results

In this chapter we will detail the tests developed and discuss the results and findings.

The main objective is to avoid migration of VEs that have data traffic inside the same host. If we migrate a VE that has network activity inside the same host, we may improve overall CPU utilization, but we will increase network traffic between hosts. Therefore, results have shown that the proposed heuristic h_{Tot} has generated less migrations when compared to the CPU heuristic h_{CPU} (PIRAGHAJ et al., 2015). Both heuristics apply the Pearson correlation coefficient model, nonetheless the proposed heuristic h_{Tot} includes network traffic in the analysis. The tests have confirmed that less migrations have occurred in an environment with internal network traffic, like the one generated by microservices.

Two types of tests were conducted. One that simulates real traffic, allowing the stress test of the architecture and evaluation of the results against different number of hosts. The other, a testbed in a minimalist environment.

The testbed environment was developed to validate the Decision and Monitor modules, according to what was discussed in Chapter 4 - Architecture. Docker was selected as the container technology for the tests and was executed in experimental mode in order to have the checkpoint/restore functionality enabled. The environment consisted of Ubuntu 16.04 kernel 4.10.0-rc8, CRIU version 2.10, Docker 1.13.1 build 092cba3 and the VMs were running under Virtual Box 5.0.24. The applications were developed in Java, except for the Traffic Monitor that was developed in C, using the libcap library.

The Decision module can be executed in evaluation mode, in which all calculations are done, but the actual migration command is not executed, or production mode, in which live migration is performed.

5.1 Simulation

For the simulation, the Monitor module was adapted to send different loads to the Decision module. The tests simulated an environment with 100, 300, 500, 1000, 2500 and 5000 hosts. Tests were repeated 30 times. Each host had a random number of containers, varying from 2 to 4. CPU level for each container was randomly generated, with the constraint that the total host CPU load could not exceed 100%. The Decision module was developed to be independent of the virtualization technology. The only requirement is that the Monitor sent messages following a defined pattern. Figure 15 displays the total numbers of containers for each of the different loads.

Figure 15: Number of hosts and containers.

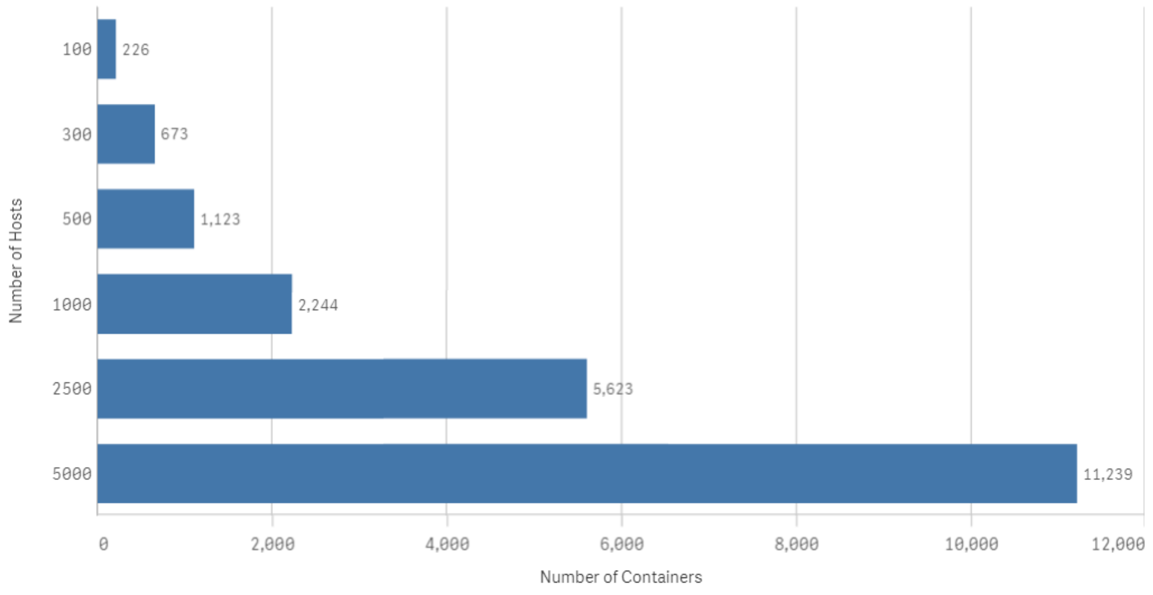
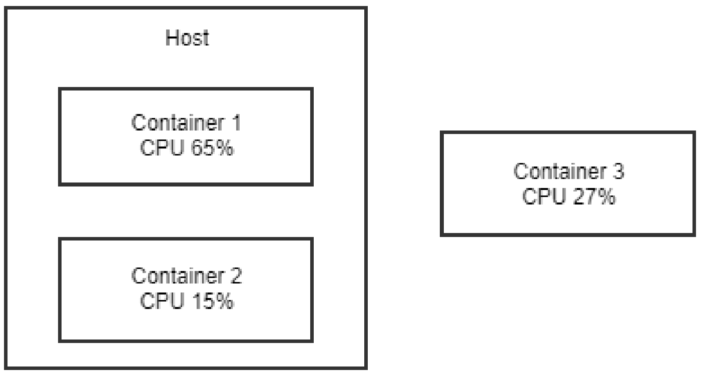


Figure 16 will assist in the explanation of how containers were distributed in our simulation. Suppose that for a host, the random number of containers selected was 3. The CPU usage was also randomly generated. In our example, Container 1 has 65%, Container 2 has 15% and Container 3 has 27% of CPU usage respectively. Containers 1 and 2 are allocated at host, however, Container 3 is discarded, as the total CPU for the host will pass the 100% limit. Therefore, the host will keep the 2 containers, with total CPU of 80%. As far as the network traffic, for every container, the traffic for TX and RX was randomly generated as well. So when the container is allocated to a host, it carries the values for CPU usage and network traffic data.

Figure 16: Container allocation.



Appendix B displays the Figures with the CPU usage distribution for all the simulations performed. As an example, let us detail here the scenario with 1000 hosts. The other scenarios will follow the same reasoning. Our objective is to evaluate the number of migrations using our proposed heuristic h_{Tot}, and compare them with the heuristic h_{CPU}. Figure 17 display the results for the simulation with 1000 hosts. Every Figure has three bar charts. The first one is entitled Initial State, and displays the initial configuration of the data center. It shows the number of hosts per intervals of CPU usage. Considering that our CPU threshold is 70%, there are 755 hosts overloaded. The remaining 245 hosts are considered under-loaded, and will be taken into account to receive a container during the migration process. Given this initial state, the following two charts display the result after applying the heuristic h_{CPU} and our proposed heuristic h_{Tot}. For the first, the migration threshold was defined to migrate containers with Pearson correlation coefficient r_{CPU} greater or equal to 0.70. For the second, to migrate containers when the score of h_{Tot} is less or equal to 1.30. The chart entitled Pearson shows that the number of overloaded hosts went to 529, and the chart called Heuristic that the number of overloaded hosts went to 558. Both methods provided a better load balancing, however the heuristic h_{Tot} generated less migrations due to the analysis of the network traffic. As Morabito (MORABITO, 2015) described, increasing network traffic also increases power consumption in a data center.

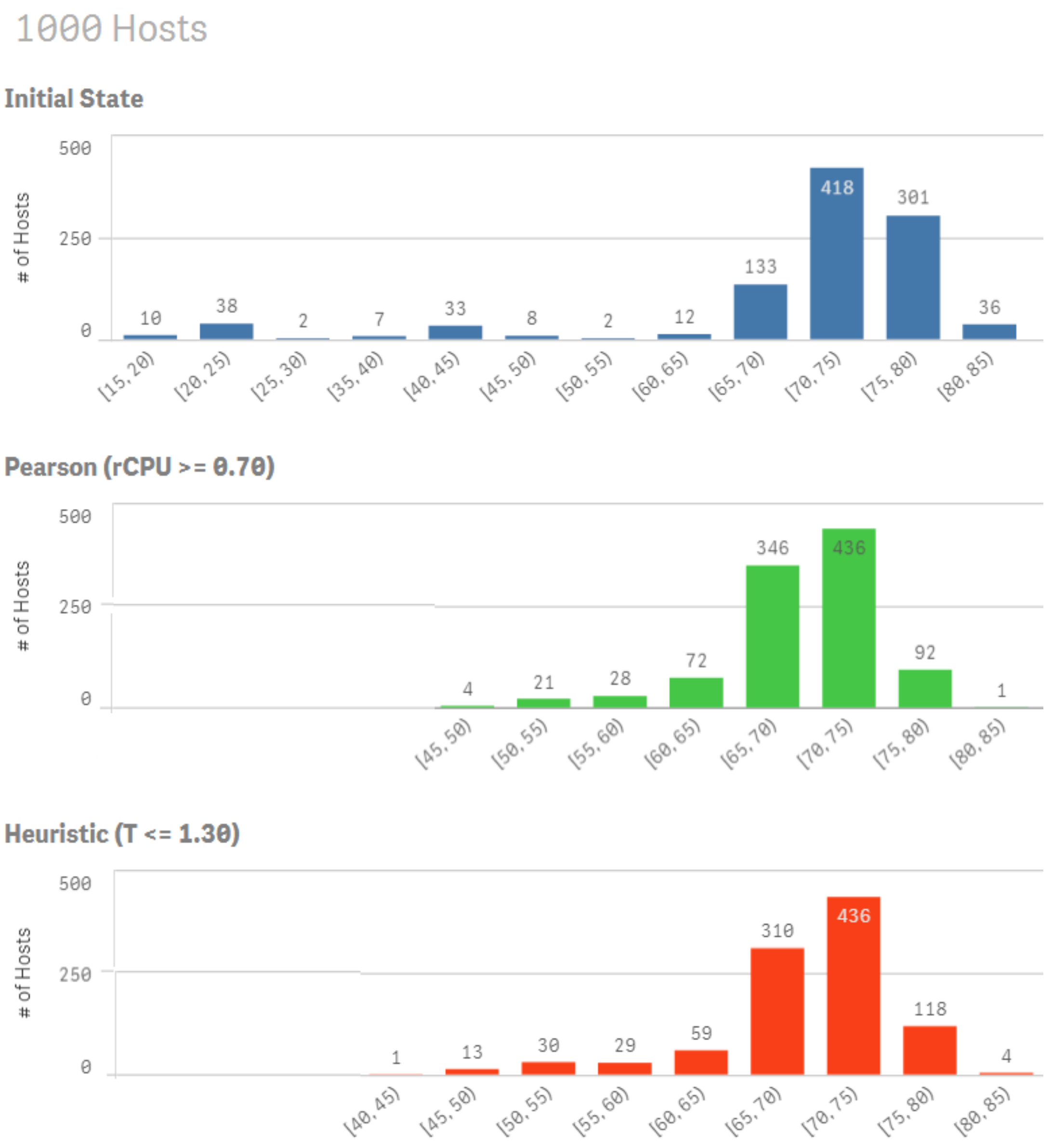
The same analysis can be done for each of the simulations presented at Appendix B. As far as the improvement in load balancing, we can make some comparisons using the tests performed. Let us first analyze the CPU load distribution, and then, the number of migrations performed.

Figure 18 shows the mean distribution of the CPU for each of the tests. The mean is at the 69% range.

The standard deviation (σ) provides the level of dispersion of the data set. Figure 19 displays the standard deviation for the initial state (CPU0), for h_{CPU} and for our proposed heuristic h_{Tot}. The initial state has a higher standard deviation compared to the heuristics, indicating that the CPU loads were spread out over a wider range of values. When we look the standard deviation after applying the two heuristics, we see a lower standard deviation, indicating that the CPU usage is closer to the mean. In all cases, the standard deviation of h_{CPU} was slightly lower than h_{Tot}, just for a small percentage. The reason is that we have more migrations when we apply h_{CPU}. This behaviour was expected, because h_{CPU} does not take into consideration network traffic.

Another analysis that can be done is to look at one standard deviation on either side of the mean. For the normal distribution, this accounts for 68.27% of the data set. Figures 20 and 21 display the results, been CPU0 equals to the initial state, CPU the results after applying the heuristic h_{CPU} and h_{Tot} after applying our proposed heuristic. The initial configuration presents a higher interval. This was expected as σ was higher than the heuristics. After applying the heuristics, the interval is smaller, showing a better

Figure 17: Container Migration Results - 1000 Hosts.



distribution of the workload. The difference between both heuristics is very small. For h_{CPU} the CPU interval is in the range of 64% to 74% and for h_{Tot} in the range of 63% to 75%.

The supporting Tables for Figures 20 and 21, with the statistical data, are presented at Appendix C, Sections C.1, C.2 and C.3.

After looking at the CPU distribution, let us analyze the load balancing. As the CPU threshold was defined as 70%, it is important to check how many hosts were above this mark. These hosts are considered overloaded. Our objective is to reduce the number of overloaded hosts in a data center. Figure 22 displays the number of overloaded hosts for all the simulations. Both heuristic methods, labeled CPU and Heur in the graph, improved

Figure 18: CPU Mean.

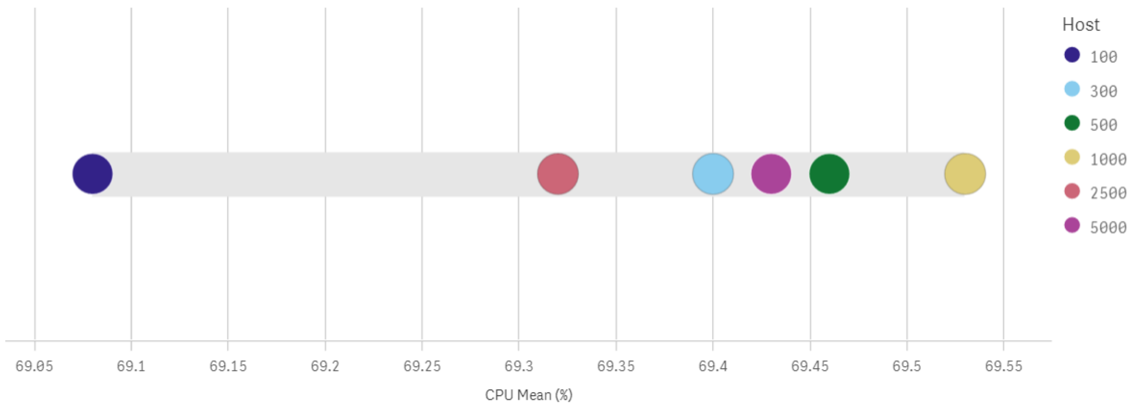
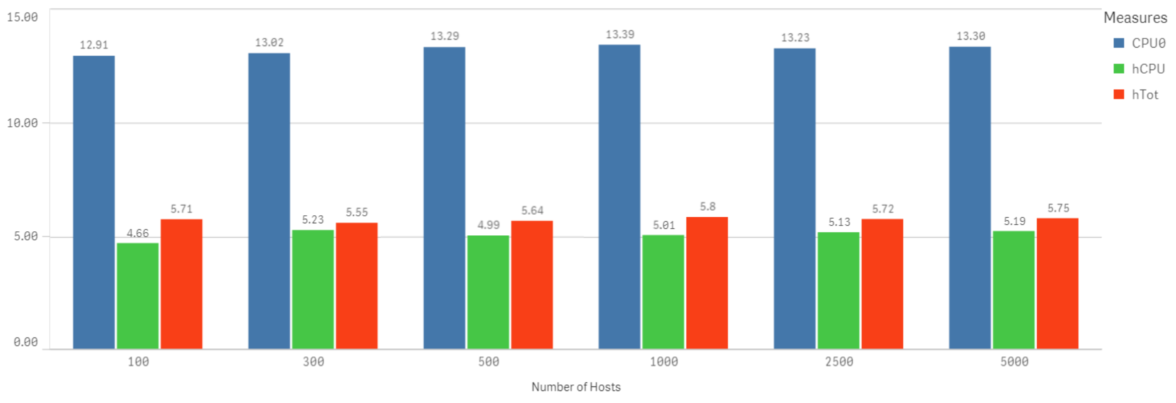


Figure 19: CPU standard deviation.



the load balancing.

Based on the number of hosts with CPU above 70%, we can calculate the percentage reduction on the number of overloaded hosts after applying the heuristics. Figure 23 displays these percentages. The supporting Table for Figure 23 is at Appendix C, Section C.4. Both heuristic methods presented very close results. The h_{CPU} model has shown less overloaded hosts compared to h_{Tot}, nonetheless, this behaviour was expected, as our proposed heuristic migrated less containers due to network traffic.

Table 4 have a description of variables that will be used in the following evaluation.

Figure 24 and Table 5 detail the total number of migrations based on h_{CPU} (column MigrhCPU) and on h_{Tot} (column MigrhTot). The last column calculates the reduction on migrations when you compare both methods (column %MigrReduction). It is important to notice that we had at least a 18.81% reduction in the number of migrations using our proposed approach. Equation 5.1 defines the calculation for column % MigrReduction.

$$MigrReduction = (1 - (MigrhTot/MigrhCPU)) * 100\% \tag{5.1}$$

Figure 20: CPU variance between $\bar{x} - 1\sigma$ and $\bar{x} + 1\sigma$ (a).

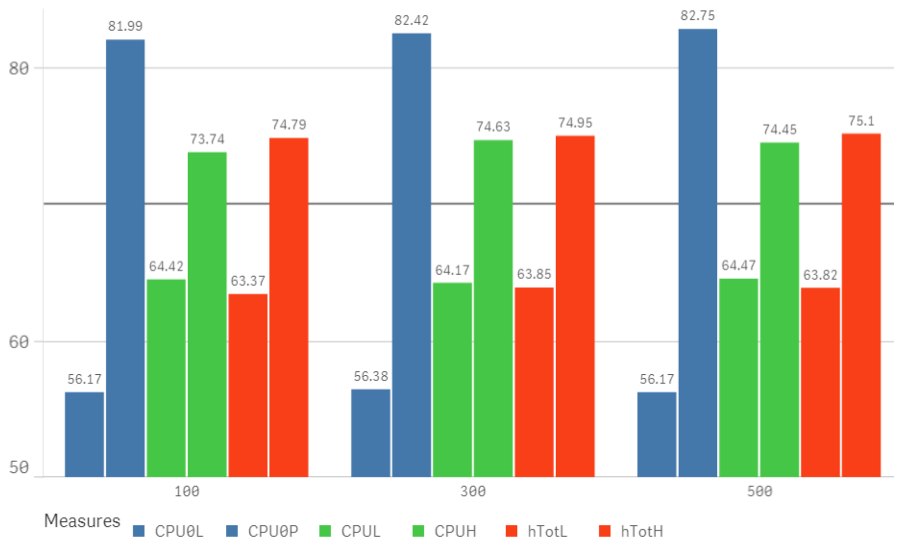


Figure 21: CPU variance between $\bar{x} - 1\sigma$ and $\bar{x} + 1\sigma$ (b).

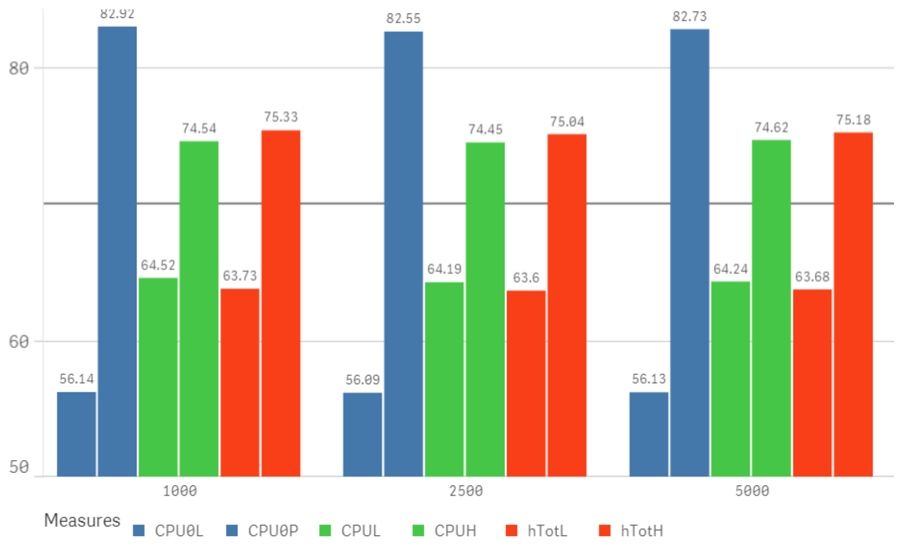


Table 6 helps in analyzing how the network traffic has influenced in the decision process, considering h_{CPU} and h_{Tot}, ΔMigration is defined as the difference between the number of container migrations . Equation 5.2 details the calculation.

$$\Delta Migration = MigrhCPU - MigrhTot \tag{5.2}$$

A comparison can be made between ΔMigration and the number of containers with high Pearson correlation coefficient, considering CPU and network traffic. At Table 6, column High Correlation, displays the number of containers with high correlation coefficient for CPU, TX and RX. Column HighMigration% has the ratio between High Pearson and ΔMigration. The idea is to verify if the reduction in migrations, considering

Figure 22: Number of hosts with CPU above 70%.

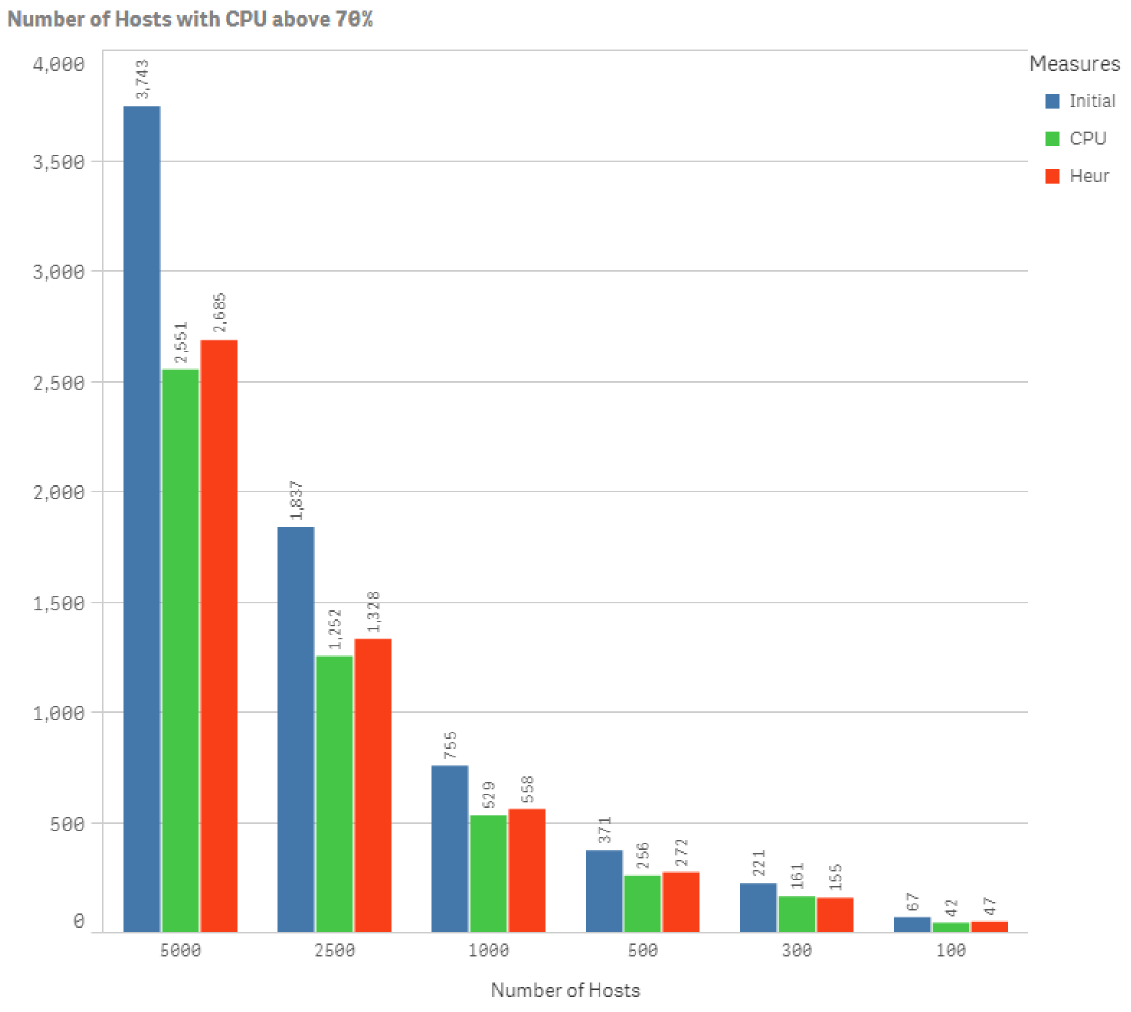


Table 4: Variable Definitions.

Symbol	**Definition**
MigrhCPU	# of container migrations after applying h_{CPU}
MigrhTot	# of container migrations after applying h_{Tot}
MigrReduction	% reduction migrations of h_{Tot} and h_{CPU}
ΔMigration	MigrhCPU - MigrhTot
%Reduction	% reduction migrations of h_{Tot} and h_{CPU}
HighCorrelation	# of containers with strong association of r_{CPU}, r_{TX} and r_{RX}
HighMigration%	(HighCorrelation / ΔMigration) * 100

network traffic, is contained in the calculated number of migration based on our proposed heuristic. Equation 5.3 details the calculation.

$$HighMigration\% = (HighCorrelation/\Delta Migration) * 100 \tag{5.3}$$

Figure 25 displays the results. We have more migrations applying h_{CPU} than h_{Tot}. That is why ΔMigration has a positive sign. As mentioned before, that happened because

Figure 23: Percentage Reduction in the number of overloaded hosts.

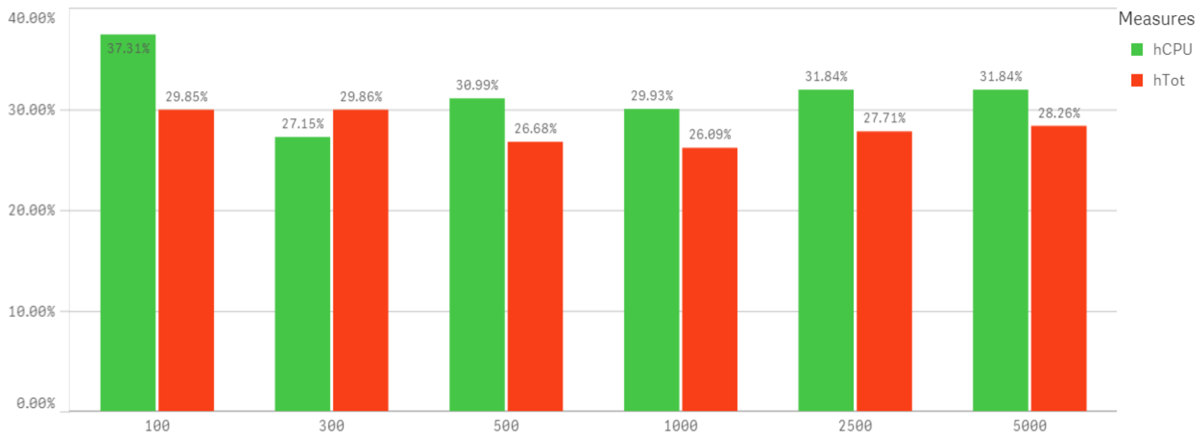


Figure 24: Number of migrations and % Reduction.

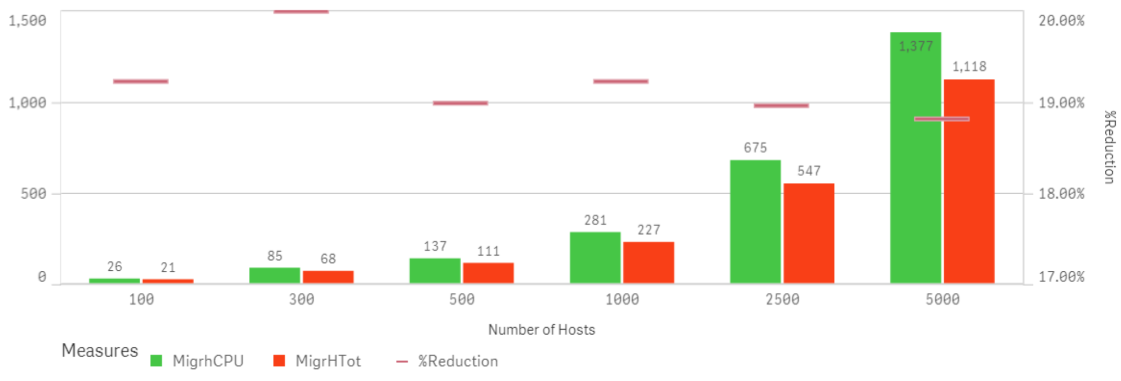


Table 5: Number of migrations.

NumHosts	MigrhCPU	MigrhTot	% MigrReduction
100	26	21	19.23%
300	85	68	20.00%
500	137	111	18.98%
1000	281	227	19.22%
2500	675	547	18.96%
5000	1377	1118	18.81%

the proposed heuristic took in consideration network traffic. When we compare ΔMigration with the number of high correlated containers, we see that the number of high correlated containers is less than ΔMigration. The high correlated containers represent 70% to 75% of ΔMigration. The difference is not exact, because, by definition, a heuristic is designed to find an approximate solution. Therefore, we have between 25% to 30% of containers that were not exactly with high correlation for CPU, TX and RX that were not migrated. For our test scenarios, results showed that the heuristic is a good approximation for most of the cases.

As Morabito (MORABITO, 2015) mentioned, it is important to avoid migration of

Figure 25: Migration Comparison between heuristic and high correlated hosts.

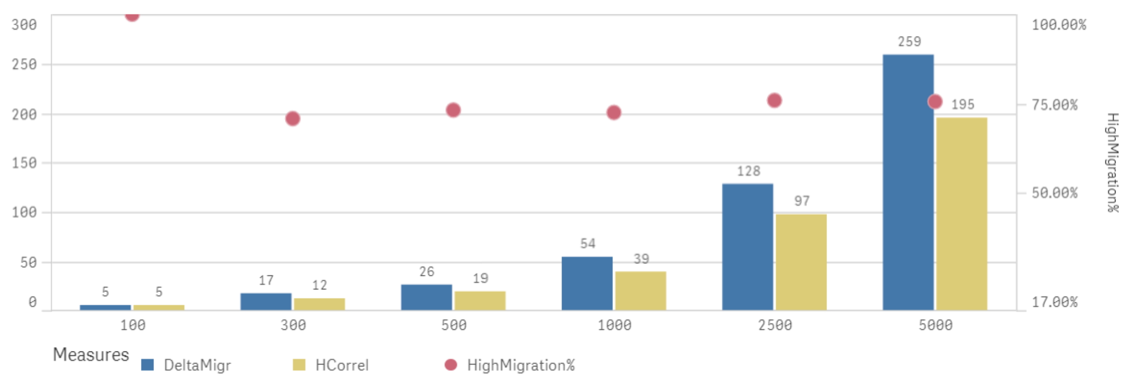


Table 6: Migration Comparison between heuristic and high correlated hosts.

NumHosts	ΔMigration	High Correlation	HighMigration%
100	5	5	100%
300	17	12	70.58%
500	26	19	73.07%
1000	54	39	72.22%
2500	128	97	75.78%
5000	259	195	75.28%

elements with network traffic. Migrating such VEs, will increase overall network traffic, increasing power consumption. The advance of microservices technology will create more data traffic between containers (AMARAL et al., 2015) and, therefore, is important to take network traffic in consideration when deciding if a VE should migrate or not.

It is important to mention that we could make the heuristic model more conservative, setting the threshold $T_{heuristic}$ less than 1.30, or make it more flexible, setting the threshold greater than 1.30. In the first case, we will have less migrations, increasing the restriction in migrating containers that have network traffic. In the second case, more migrations will occur.

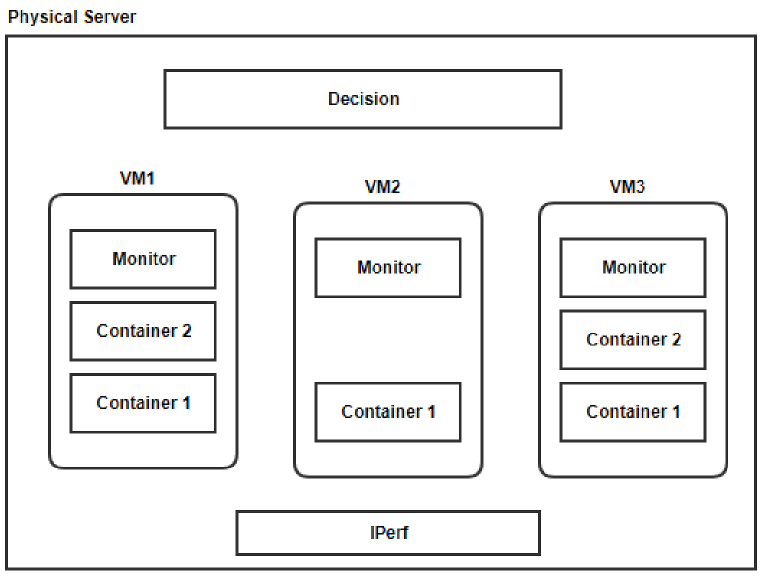
The Decision module also generates a log file that stores the container IDs that could not be migrated due to the fact that there was no host with available capacity to receive them. The log file can be an indication to the administrator that new hosts or VMs should be instantiated to provide more available resources, allowing more migrations and a better distribution of the workload. Such task could be automated, if a module were created to interpret and evaluate this log file.

In conclusion, applying the proposed heuristic generated at least 18.81% less migrations in environments with network traffic. When we look at the number of overloaded hosts, applying both heuristics generated a better load balancing, with a very small difference. However, our proposed heuristic reduced the number of migrations due to network traffic, reducing power consumption, and still provided a better load balancing, with very small difference compared to the heuristic based on CPU.

5.2 Testbed

A testbed environment was developed to validate the heuristic and the modules presented at Chapter 4. Docker was selected as the container technology and was executed in experimental mode in order to have the checkpoint/restore functionality enabled. The environment consisted of Ubuntu 16.04 kernel 4.10.0-rc8, CRIU version 2.10, Docker 1.13.1 build 092cba3 and the VMs were running under Virtual Box 5.0.24. Hardware was a Dell Inspiron 15 Series 5000, Intel Processor Core i7-5500U 5th generation (2.4 GHz, 4MB Cache) and 16GB RAM memory. Figure 26 shows the proposed environment.

Figure 26: Testbed Environment.



The minimized environment is composed of 3 VM's with 4GB of RAM each. Two type of tests were performed.

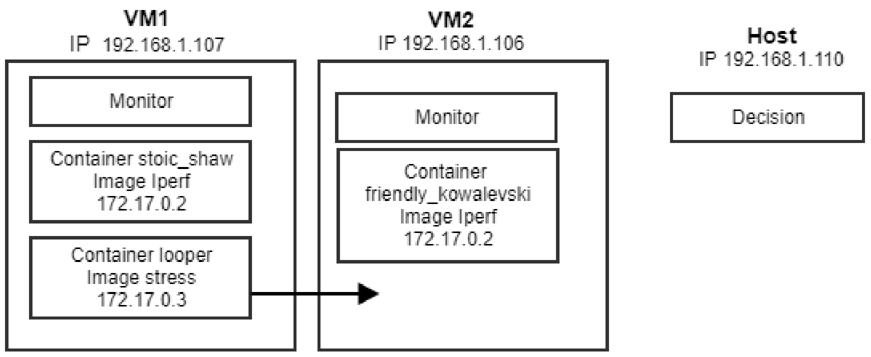
- CPU Stress: stress is a Linux tool that can be used to stress test the environment. The tool was used to stress the CPU usage to values above 70%. The stress tool was executed inside the VM and also inside the containers to test different scenarios.
- Iperf: Iperf is a tool for measurements of the maximum achievable bandwidth on IP networks. It supports various parameters related to timing, buffers and protocols (IPERF, 2018). Iperf was used to simulate traffic between containers and between containers and host.

Below we describe the tests executed in the minimized environment.

5.2.1 Stress command

Figure 27 details the stress test scenario that will be explained in sequence. The purpose will be to run the stress tool at container called looper and perform a live migration to VM2.

Figure 27: Stress CPU Example.



Creating a container

The docker command below runs in background a container called looper, based on a Docker image named stress. Image stress is a ubuntu image, with the stress tool installed on top of it. The looper container will execute a loop with a counter. We will monitor the counter to verify that, after the migration, the process continue running uninterruptedly. The IP address of VM1 is 192.168.1.107.

```
docker run -d --name looper --security-opt seccomp:unconfined stress
  /bin/sh -c ’i=0; while true; do echo $i; i=$(expr $i+1);sleep 1;done’$
```

Checking the running containers at VM1, we found two containers, one called looper and the other called stoic_shaw:

```
root@docker-vm1:# docker ps
CONTAINER ID  IMAGE   COMMAND                  CREATED            NAMES
fa1a543429bf  stress  "/bin/sh -c ’i=0; ..." About a minute ago looper
975d7028f733  iperf   "./bin/bash"             24 hours ago      stoic_shaw
```

To verify that the looper container is running correctly, let us check the output generated by the loop:

```
root@docker-vm1:# docker logs looper
0
```

Stressing the container CPU

In order to stress the CPU usage, the stress tool was executed inside the looper container, bringing the container CPU to near 100%. The option -c is related to the CPU stress and -t means that the tool should run for 30 seconds.

```
docker exec -d looper /usr/bin/stress -c 1 -t 30
```

Let us see how each module behave after the execution of the stress tool.

Decision Module

The Decision Module is running at host IP 192.168.1.110 and is receiving data from two hosts, one with IP 192.168.1.107 (VM1) and the other 192.168.1.106 (VM2). The module identifies that the CPU of VM1 is overloaded and that container looper (container IP 172.17.0.3) should be verified. It also selects VM2 as the destination host. For each container, the Decision module calculates Pearson correlation coefficient, applies the heuristic and calculate the scores for h_{Tot}. After identifying that container looper has the lowest score and that is below the stipulated threshold, it decides to migrate the container. It sends a migration message to the Migration Module at VM2. The output generated by the Decision module is presented at Appendix A, Section A.3.

Monitor Migration Module - Sub-Module Migrate

After receiving the message, the Monitor Migration module starts the migration process at VM1, saving the container image, checkpointing the current state and transferring the files to the destination host. The output generated by the Migrate sub-module is presented at Appendix A, Section A.4.

In order to save the image and perform the checkpoint, the following commands were executed. ImageId and ContainerId can be obtained via docker inspect command, passing ContainerName as parameter.

```
    IMAGEID=(docker inspect --format="{{.Config.Image}}" CONTAINERNAME)
    CONTAINERID=(docker inspect --format="{{.Id}}" CONTAINERNAME)
    docker save -o IMAGEID.tar IMAGEID
    docker checkpoint create CONTAINERNAME checkpoint1
```

Monitor Migration Module - Sub-Module Restore

At VM2, the file transfer was detected and the restore process takes place. The output generated by sub-module Restore is presented at Appendix A, Section A.5.

In order to load the image and restore the container state, the following commands were executed at the script:

```
    docker load -i IMAGENAME.tar
    CONTAINERID=(docker create -P --name CONTAINERNAME IMAGENAME)
    docker start --checkpoint checkpoint1 CONTAINERNAME
```

At destination, we already had one container running called friendly_kowalevski:

```
root@docker-vm2:# docker ps
CONTAINER ID  IMAGE  COMMAND      CREATED        NAMES
9c7a2c0b9cb4  iperf  "./bin/bash"  23 hours ago  friendly_kowalevski
```

After the migration was completed, the container looper is also displayed:

```
docker@docker-vm2:# docker ps
CONTAINER ID  IMAGE  COMMAND      CREATED        NAMES
008a33fef905  stress "./bin/bash" 4 minutes ago  looper
9c7a2c0b9cb4  iperf  "./bin/bash" 23 hours ago   friendly_kowalevski
```

Monitor Module

The Monitor module displays the CPU and memory for the host and all running containers. The output generated by this module is presented at Appendix A, Section A.6.

If we check the output for the looper container, we see that the loop is still running:

```
docker@docker-vm2:# docker logs looper
1194
1195
1196
```

Another interesting aspect to be analyzed is the total transfer time. Figure 5, presented in Chapter 2, detailed the stop and copy mechanism for container migration. There is a frozen time in which the container is out of service. In our test, the Migration process took 9 seconds at source (sub-module Migrate) and 1 second at destination (sub-module Restore). In the source host, the image was saved and the checkpoint created. Then the files were transferred to the destination. This part of the process rely on the network quality, and is critical, due to the period of time that can takes to be completed. Therefore, the total migration time for a Ubuntu image of 190MB size was 10 seconds.

Elsaid & Meinel (ELSAID; MEINEL, 2014) analyzed live migration impact of VMs in a data center. They have defined transmission time in an IP network as:

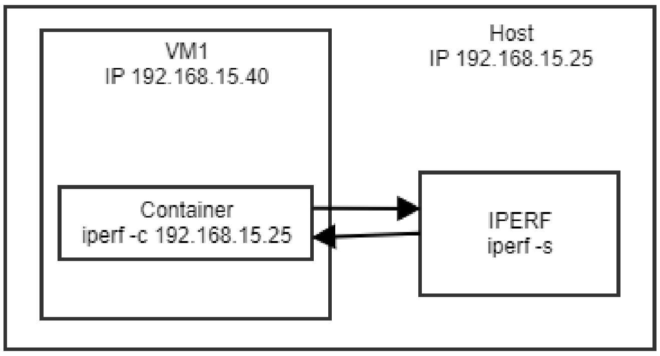
$$T_{Tran} = V_{pkt}/R \tag{5.4}$$

Where T_{Tran} is the transmission time, V_{pkt} is the packet volume size in bits and R is the migration transmission rate in bits/sec. The authors found during their tests, that a VM of size of 4GB took 113 seconds to migrate, one of 2GB took 105 seconds and 1GB took 92 seconds. VMs were running VMWare (VMWARE, 2018) in a fiber network. As we have discussed previously, the size of a VM image is bigger compared to containers, and, therefore, the transmission time is longer. In case of VM migration, it is important to have mechanisms that optimize the process due to the fact that the migration itself consumes lots of computational resources. As our test has shown, container live migration has advantages as they are lightweight compared to VMs and the total migration time took only 10 seconds.

5.2.2 Iperf command - External Data Traffic

Let us analyze how the module behaves when we generate traffic using iperf. Figure 28 details the test environment.

Figure 28: Iperf external traffic.



In order to test data traffic from a VE outside the VM environment, we run iperf server in a host with IP 192.168.15.25. Then, inside the container running on VM1, we execute iperf client, directing the traffic to the host.

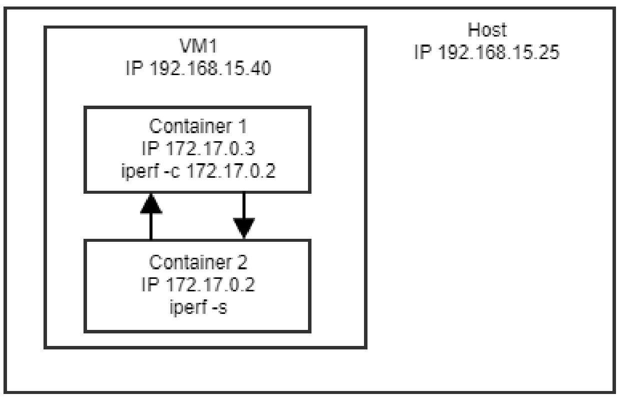
```
root@vm1:# iperf -c 192.168.15.25
```

The Decision module will allow the migration, as the network traffic is outside VM1. If we migrate this container to another host, it will not affect the overall network performance, because the external traffic is already happening. Moving the container to another host will not create additional traffic. The output generated by the Decision module is presented at Appendix A, Section A.7. The result of the iperf server command can be seen at Appendix A, Section A.8.

5.2.3 Iperf command - Internal Data Traffic

Another simulation is to run the iperf command from one container to another container in the same VM. Figure 29 details the test environment.

Figure 29: Iperf internal traffic.



In this scenario, we run iperf server on a container C2 with IP 172.17.0.2, and the iperf client at container C1 with IP 172.17.0.3. All inside the same VM.

The Monitor module detects the CPU overload at 81%, and also calculates the data traffic to be sent to the Decision module. The output generated by the Monitor module is presented at Appendix A, Section A.9.

The Decision module detects that there is a host overloaded, and start the decision process. Through the application of the heuristic and the score calculation, the Decision module detects that there is traffic between containers inside the same host and abort the migration process. The output generated by the Decision module is presented at Appendix A, Section A.10. In this scenario, if we migrate one of these containers to another host, additional traffic would be added to the overall network as external traffic will be created.

The results of the iperf commands can be seen at Appendix A, Section A.11.

The testbed presented describes how the modules behave in a minimalist environment. Three type of scenarios were explained and the end results discussed.

Conclusion

Virtualization is one of the foundations of cloud computing as it allows better utilization of computing resources in a data center. Live migration is a functionality that solves a load balancing problem, allowing better distribution of resources in case of overloaded hosts, providing a more energy efficient environment. Furthermore, the possibility of migrating VEs between different cloud providers, is an empowerment to users that can choose the provider based on better SLAs and reduced costs.

In this dissertation, a heuristic, that applies Pearson correlation coefficient to CPU utilization and internal network traffic of VEs, was proposed. The contribution of this work is to take into consideration the network traffic during the migration decision process, and, to apply a heuristic using the Pearson correlation coefficient. By definition a heuristic is designed to find an approximate solution, making the decision process quicker compared to complex methods that look for an exact solution. The adjustment of the heuristic threshold $T_{heuristic}$ can make the migration decision more flexible or more conservative, increasing or reducing the number of migrations respectively. A testbed using Docker containers was developed to evaluate the application of the heuristic. A comparison was done against the model that also uses the Pearson correlation coefficient but that only considers CPU consumption. Results shown that when we apply the heuristic, migration of containers that have network traffic with other containers in the same host, was avoided. In average, with $T_{heuristic}$ equals to 1.30, the application of the heuristic has provided a migration reduction in the order of 18% compared to the CPU only correlation model. Load balancing has improved with less number of overloaded hosts after the heuristic was applied. The proposed heuristic reduced the number of overloaded hosts in at least 26%. The fact that the application of the heuristic provided less migrations, implies that less computational resources were consumed. Not only during migration time, but also, if we consider that network traffic between hosts would increase.

Results showed the importance of taking into consideration network traffic in the migration decision process. The use of the Pearson correlation coefficient model was efficient not only with CPU, but also with network traffic as well. The applied heuristic showed a reduction on the number of migrations of VEs that had relationship with host CPU and other VEs internal data traffic and also an improvement on the CPU distribution.

Future Work

Our simulation has evaluated load balancing of CPU usage after applying the heuristic. The results have shown a better distribution of the work load, displaying less servers overloaded. The analysis detailed that less migrations were done due to the network traffic between VEs. A next step would be to monitor the network itself, taking snapshots of the data traffic and evaluating how the traffic is behaving after each interaction of the proposed model.

Another aspect that could be considered as future work is analyzing the network topology of servers in the data center. As an example, Mysore et al. (MYSORE et al., 2009) defined a topology called Portland, which employs a lightweight protocol to enable switches to discover their position in the topology. Knowing the position of each host in the data center, would allow the inclusion of this factor in the decision process. Therefore, the decision process could allow migration of VEs that are part of the same Top of Rack (ToR), and deny migrations of VEs outside ToR.

Likewise, a module could be developed to start or stop VEs automatically. Suppose that reading current logs, it is identified that it is required to instantiate more hosts or VMs in order to allow more migration and a better distribution of the work load. Or that there are VEs that could be put in power safe mode to reduce power consumption. This new module would improve the overall performance of the data center.

Bibliography

AMARAL, M. et al. Performance evaluation of microservices architectures using containers. In: IEEE. *Network Computing and Applications (NCA), 2015 IEEE 14th International Symposium on*. [S.l.], 2015. p. 27–34. Cited 5 times in pages 33, 34, 45, 53, and 75.

ASHINO, Y.; NAKAE, M. Virtual machine migration method between different hypervisor implementations and its evaluation. In: IEEE. *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*. [S.l.], 2012. p. 1089–1094. Cited 2 times in pages 27 and 36.

BERNSTEIN, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, IEEE, v. 1, n. 3, p. 81–84, 2014. Cited 2 times in pages 34 and 58.

BUYYA, R. et al. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, Elsevier, v. 25, n. 6, p. 599–616, 2009. Cited 2 times in pages 25 and 31.

CELESTI, A. et al. Improving virtual machine migration in federated cloud environments. In: IEEE. *Evolving Internet (INTERNET), 2010 Second International Conference on*. [S.l.], 2010. p. 61–67. Cited 3 times in pages 27, 36, and 38.

CHEN, P. Y.; POPOVICH, P. M. *Correlation: Parametric and nonparametric measures*. [S.l.]: Sage, 2002. Cited in page 44.

CHRISTINE, P.; JOHN, R. Statistics without maths for psychology: using spss for windows. *Harlow, UK: Pearson Prentice Hall*, 2004. Cited 5 times in pages 44, 47, 49, 55, and 56.

CLARK, C. et al. Live migration of virtual machines. In: USENIX ASSOCIATION. *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. [S.l.], 2005. p. 273–286. Cited 3 times in pages 33, 38, and 39.

CRIU. *Checkpoint/Restore In Userspace*. 2018. <https://criu.org/Main_Page>. Cited 2 times in pages 37 and 41.

DOCKER. *Docker API*. 2017. <https://docs.docker.com/engine/api/v1.25/#>. Cited in page 60.

DOCKER. *Docker*. 2018. <https://www.docker.com/>. Cited 3 times in pages 25, 34, and 64.

DOCKERCRIU. *DOCKERCRIU*. 2018. <https://criu.org/Docker>. Cited in page 64.

DOCKERSTATS. *Docker Stats*. 2016. <https://github.com/moby/moby/blob/eb131c5383db8cac633919f82abad86c99bffbe5/cli/command/container/stats_helpers.go#L175-L188>. Cited in page 61.

DOCKERSWARM. *Docker Swarm*. 2018. <https://docs.docker.com/engine/swarm/>. Cited in page 34.

ELSAID, M. E.; MEINEL, C. Live migration impact on virtual datacenter performance: Vmware vmotion based study. In: IEEE. *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*. [S.l.], 2014. p. 216–221. Cited 5 times in pages 27, 33, 36, 41, and 80.

FELTER, W. et al. An updated performance comparison of virtual machines and linux containers. In: IEEE. *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. [S.l.], 2015. p. 171–172. Cited 2 times in pages 25 and 33.

FILHO, D. B. F.; JÚNIOR, J. A. d. S. Desvendando os mistérios do coeficiente de correlação de pearson (r). Universidade Federal de Pernambuco, 2009. Cited 4 times in pages 33, 43, 44, and 46.

GROZEV, N.; BUYYA, R. Dynamic selection of virtual machines for application servers in cloud environments. *arXiv preprint arXiv:1602.02339*, 2016. Cited in page 41.

HYPER-V. *Microsoft Hyper-V*. 2018. <https://www.microsoft.com/en-us/cloud-platform/server-virtualization>. Cited in page 25.

IPERF. *IPERF*. 2018. <https://iperf.fr/>. Cited 3 times in pages 35, 53, and 76.

JELASTIC. *JELASTIC*. 2018. <https://jelastic.com/>. Cited in page 37.

JOY, A. M. Performance comparison between linux containers and virtual machines. In: IEEE. *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*. [S.l.], 2015. p. 342–346. Cited in page 25.

KALIM, U. et al. Seamless migration of virtual machines across networks. In: IEEE. *2013 22nd International Conference on Computer Communication and Networks (ICCCN)*. [S.l.], 2013. p. 1–7. Cited in page 40.

KANSAL, N. J.; CHANA, I. Cloud load balancing techniques: A step towards green computing. *IJCSI International Journal of Computer Science Issues*, sn, v. 9, n. 1, p. 238–246, 2012. Cited 2 times in pages 27 and 35.

KANSO, A.; HUANG, H.; GHERBI, A. Can linux containers clustering solutions offer high availability? 2016. Cited in page 37.

KERNEL, L. *Proc Kernel Page*. 2018. <https://www.kernel.org/doc/Documentation/filesystems/proc.txt>. Cited in page 60.

KUBERNETES. *Kubernetes*. 2018. <http://kubernetes.io/>. Cited 2 times in pages 34 and 37.

KVM. *KVM*. 2018. <http://www.linux-kvm.org/page/Main_Page>. Cited 2 times in pages 25 and 34.

LI, W.; KANSO, A. Comparing containers versus virtual machines for achieving high availability. In: IEEE. *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. [S.l.], 2015. p. 353–358. Cited 2 times in pages 25 and 37.

LI, W.; KANSO, A.; GHERBI, A. Leveraging linux containers to achieve high availability for cloud services. In: IEEE. *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. [S.l.], 2015. p. 76–83. Cited 3 times in pages 25, 26, and 37.

LINTHICUM, D. S. Moving to autonomous and self-migrating containers for cloud applications. *IEEE Cloud Computing*, IEEE, v. 3, n. 6, p. 6–9, 2016. Cited 2 times in pages 27 and 36.

LXC. *Linux Containers - LXC*. 2018. <https://linuxcontainers.org/>. Cited 3 times in pages 25, 34, and 38.

LXD. *Linux Containers - LXD*. 2018. <https://linuxcontainers.org/lxd/>. Cited in page 34.

MANPAGE, L. P. *Proc Man Page*. 2018. <http://man7.org/linux/man-pages/man5/proc.5.html>. Cited in page 59.

MELL, P.; GRANCE, T. et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 2011. Cited 3 times in pages 26, 31, and 32.

MORABITO, R. Power consumption of virtualization technologies: an empirical investigation. In: IEEE. *Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on*. [S.l.], 2015. p. 522–527. Cited 6 times in pages 27, 35, 45, 53, 69, and 74.

MYSORE, R. N. et al. Portland: a scalable fault-tolerant layer 2 data center network fabric. In: ACM. *ACM SIGCOMM Computer Communication Review*. [S.l.], 2009. v. 39, n. 4, p. 39–50. Cited 2 times in pages 41 and 84.

OPENVZ. *OpenVZ*. 2018. <https://openvz.org/Main_Page>. Cited 3 times in pages 25, 37, and 38.

PEARSON, K. Royal society proceedings. 1895. Cited 5 times in pages 27, 28, 33, 35, and 43.

PIRAGHAJ, S. et al. A framework and algorithm for energy efficient container consolidation in cloud data centers. In: IEEE. *Data Science and Data Intensive Systems (DSDIS), 2015 IEEE International Conference on*. [S.l.], 2015. p. 368–375. Cited 11 times in pages 26, 27, 28, 35, 41, 43, 47, 49, 54, 55, and 67.

PROXMOX. *Proxmox Virtual Environment*. 2018. <https://www.proxmox.com/en/proxmox-ve>. Cited in page 34.

RIGHTSCALE. *State of the Cloud Report, 2016*. 2016. <https://www.rightscale.com/lp/state-of-the-cloud>. Cited 4 times in pages 25, 34, 41, and 58.

RKT. *rkt*. 2018. <https://coreos.com/rkt/>. Cited in page 25.

ROMERO, F.; HACKER, T. J. Live migration of parallel applications with openvz. In: IEEE. *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*. [S.l.], 2011. p. 526–531. Cited in page 37.

STANTON, J. M. Galton, pearson, and the peas: A brief history of linear regression for statistics instructors. *Journal of Statistics Education*, Taylor & Francis, v. 9, n. 3, 2001. Cited 2 times in pages 33 and 43.

STEVENS, W. R.; FENNER, B.; RUDOFF, A. M. *UNIX network programming*. [S.l.]: Addison-Wesley Professional, 2004. Cited in page 62.

SYNYTSKY, R. *Containers Live Migration: Behind the Scenes*. 2016. <https://www.infoq.com/articles/container-live-migration>. Cited 4 times in pages 38, 39, 40, and 63.

TCPDUMP. *TCPDUMP*. 2018. <https://www.tcpdump.org/>. Cited in page 62.

VAQUERO, L. M. et al. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, ACM, v. 39, n. 1, p. 50–55, 2008. Cited 2 times in pages 25 and 32.

VIGLIOTTI, A. P. de la F.; BATISTA, D. M. Energy-efficient virtual machines placement. In: IEEE. *Computer Networks and Distributed Systems (SBRC), 2014 Brazilian Symposium on*. [S.l.], 2014. p. 1–8. Cited in page 35.

VMWARE. *VMWare*. 2018. <https://www.vmware.com/>. Cited 4 times in pages 25, 36, 41, and 80.

VOORSLUYS, W. et al. Cost of virtual machine live migration in clouds: A performance evaluation. In: SPRINGER. *IEEE International Conference on Cloud Computing*. [S.l.], 2009. p. 254–265. Cited in page 38.

XAVIER, M. G. et al. Performance evaluation of container-based virtualization for high performance computing environments. In: IEEE. *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. [S.l.], 2013. p. 233–240. Cited in page 38.

XEN. *Xen*. 2018. <https://xenserver.org/>. Cited 2 times in pages 25 and 41.

ZHANG, X. et al. A relationship-based vm placement framework of cloud environment. In: IEEE. *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. [S.l.], 2013. p. 124–133. Cited in page 35.

APPENDIX A – Command Outputs

A.1 Docker API JSON Response - call GET

ContainerID:

```
"Id": "8a4e2131b563c707de984837c0b20b12c1f38f5f0c2d14
4089298ae565db910a",
  "Names":
  [
      "/my-ubuntu"
  ],
  "Image": "ubuntu",
  "ImageID": "sha256:c73a085dc3782b3fd4c032971c76d6afb
  45fa3728a048175c8c77d7403de5f21",
  "Command": "/bin/bash",
  "Created": 1477068841,
  "Ports":
```

Container State:

```
  "State": "running",
  "Status": "Up 27 minutes",
```

Container Network:

```
      "Networks":
      {
          "bridge":
          {
              "IPAMConfig": null,
              "Links": null,
              "Aliases": null,
              "NetworkID": "97517875ff10f080d32353a97d312ead0f
              904b3a38a97b219744cbd2776fc957",
              "EndpointID": "7b2c768797dad4eb710405b049d13fe84
              beca9979889fe990fb74a106daa4feb",
                "Gateway": "172.17.0.1",
                "IPAddress": "172.17.0.2",
                "IPPrefixLen": 16,
                "IPv6Gateway": "",
                "GlobalIPv6Address": "",
                "GlobalIPv6PrefixLen": 0,
                "MacAddress": "02:42:ac:11:00:02"
            }
        }
```

A.2 Docker API JSON Response - container statistical data

CPU:

```
"cpu_stats":
{
  "cpu_usage":
  {
      "total_usage": 31529013,
      "percpu_usage":
      [
          2202068,
          27023982,
          2302963,
          0
      ],
      "usage_in_kernelmode": 0,
      "usage_in_usermode": 20000000
  },
  "system_cpu_usage": 960790000000,
  "throttling_data":
  {
      "periods": 0,
      "throttled_periods": 0,
      "throttled_time": 0
  }
}
```

The $SystemCPUusage$ above is equal to the sum of all fields from the /proc/stat CPU line. It represents the host CPU Usage in nanoseconds. The $CPUusage$ represents the container CPU usage itself.

Memory:

```
"memory_stats":
{
  "usage": 4177920,
  "max_usage": 4337664,
  "stats":
  {
      "active_anon": 495616,
      "active_file": 2711552,
      "cache": 3694592,
      "dirty": 0,
      "hierarchical_memory_limit": 9223372036854772000,
      "inactive_anon": 12288,
      "inactive_file": 958464,
      "mapped_file": 2785280,
      "pgfault": 919,
      "pgmajfault": 35,
      "pgpgin": 1400,
      "pgpgout": 380,
      "rss": 483328,
      "rss_huge": 0,
      "total_active_anon": 495616,
      "total_active_file": 2711552,
      "total_cache": 3694592,
      "total_dirty": 0,
      "total_inactive_anon": 12288,
      "total_inactive_file": 958464,
      "total_mapped_file": 2785280,
      "total_pgfault": 919,
      "total_pgmajfault": 35,
      "total_pgpgin": 1400,
      "total_pgpgout": 380,
      "total_rss": 483328,
      "total_rss_huge": 0,
      "total_unevictable": 0,
      "total_writeback": 0,
      "unevictable": 0,
      "writeback": 0
  },
  "failcnt": 0,
  "limit": 16742645760
}
```

Most of the memory information is also available in the /proc/meminfo file.

Network:

```
"networks":
{
  "eth0":
  {
      "rx_bytes": 11569,
      "rx_packets": 84,
      "rx_errors": 0,
      "rx_dropped": 0,
      "tx_bytes": 648,
      "tx_packets": 8,
      "tx_errors": 0,
      "tx_dropped": 0
  }
}
```

A.3 Decision Module Output - CPU Stress Test

```
00:00:32-Host: 192.168.1.106 CPU%: 2
00:00:48-Host: 192.168.1.107 CPU%: 0
00:00:51-Host: 192.168.1.106 CPU%: 2
00:01:10-Host: 192.168.1.106 CPU%: 2
00:01:11-Host: 192.168.1.107 CPU%: 100
00:01:11-Host: 192.168.1.107 CPU Threshold reached at 100%
00:01:11-Host: 192.168.1.107 Verify Container fa1a543429bf IP 172.17.0.3
00:01:11-Host: 192.168.1.107 Host Destination for migration 192.168.1.106
00:01:11-Host: 192.168.1.107 Checking Network containers of the same host
 CID: 975d7028f733 rCPU 0.0 rTX 0.0 rRX 0.0 hTot 2.0
 CID: fa1a543429bf rCPU 0.99 rTX 0.0 rRX 0.0 hTot 0.667
00:01:11-Host: 192.168.1.107 Container looper Image stress lowest hTot
00:01:11-Host: 192.168.1.107 Migrating CID fa1a543429bf to 192.168.1.106
00:01:11-Host: 192.168.1.107 Pearson rRX 0.000 Pearson rTX 0.000
00:01:11-Host: 192.168.1.107 Pearson CPU and container TX - hTX: 0.333
00:01:11-Host: 192.168.1.107 Pearson CPU and container RX - hRX: 0.333
00:01:11-Host: 192.168.1.107 Pearson CPU and Network hTot is 0.667
00:01:11-Host: 192.168.1.107 Migrating container!
00:01:11-Host: 192.168.1.107 Sending migration CMD to 192.168.1.107
00:01:11-Host: 192.168.1.107 Migration executed successfully
```

A.4 Monitor Migration Module - Sub-Module Migrate Output - CPU Stress Test

```
root@docker-vm1:# ./MonitorMigration.sh
Creating Migration Execution socket server ...

00:01:11 Migration process started ...
00:01:11 Migrating container looper to IP 192.168.1.106
#####################################################
Start Migration Process
Finding Image Name
stress
Saving Docker image
Transfering Docker Image
Getting Container ID
fa1a543429bffe3f1571ddd5cb27b46adfca89969bc51472699ef709abd6fc02
```

```
Checkpointing Container
checkpoint1
Transferring Checkpoint
End Migration Process
Time to complete migration task (s): 9
#####################################################
```

A.5 Monitor Migration Module - Sub-Module Restore Output - CPU Stress Test

```
root@docker-vm2:# ./monitor.sh
Monitor Script
Monitoring for container images for live migration
Setting up watches.
Watches established.
****************************************************
New files moved in stress at 00:01:17
Start Loading Process at 00:01:17
Container Name
Loading the Image
open /home/docker/Images/stress/stress.tar
Creating container
Copying checkpoint
Starting container from checkpoint
Finish Loading Process at 00:01:18
Total time for loading Docker Container (s):  1
****************************************************
```

A.6 Monitor Module Output - CPU Stress Test

```
00:00:11 - Host 192.168.1.106 CPU% 6 Memory% 65
00:00:11 - Image iperf IP 172.17.0.2 CPU% 0 Container friendly_kowalevski
00:00:30 - Host 192.168.1.106 CPU% 2 Memory% 65
00:00:30 - Image iperf IP 172.17.0.2 CPU% 0 Container friendly_kowalevski
00:00:49 - Host 192.168.1.106 CPU% 2 Memory% 65
00:00:49 - Image iperf IP 172.17.0.2 CPU% 0 Container friendly_kowalevski
00:01:08 - Host 192.168.1.106 CPU% 2 Memory% 65
00:01:08 - Image iperf IP 172.17.0.2 CPU% 0 Container friendly_kowalevski
```

```
00:01:29 - Host 192.168.1.106 CPU% 4 Memory% 65
00:01:29 - Image stress IP 172.17.0.3 CPU% 0 Container looper
00:01:29 - Image iperf IP 172.17.0.2 CPU% 0 Container friendly_kowalevski
00:01:52 - Host 192.168.1.106 CPU% 0 Memory% 65
00:01:52 - Image stress IP 172.17.0.3 CPU% 0 Container looper
00:01:52 - Image iperf IP 172.17.0.2 CPU% 0 Container friendly_kowalevski
00:02:15 - Host 192.168.1.106 CPU% 4 Memory% 65
00:02:15 - Image stress IP 172.17.0.3 CPU% 0 Container looper
00:02:15 - Image iperf IP 172.17.0.2 CPU% 0 Container friendly_kowalevski
```

A.7 Decision Module Output - External IPERF Test

```
20:16:11- Host: 192.168.15.40 CPU%: 0
20:16:38- Host: 192.168.15.40 CPU%: 1
20:17:05- Host: 192.168.15.40 CPU%: 0
20:17:32- Host: 192.168.15.40 CPU%: 100
20:17:32- Host: 192.168.15.40 CPU Threshold reached at 100%
20:17:32- Host: 192.168.15.40 Verify Container dea1de565b72 IP 172.17.0.2
20:17:32- Host: 192.168.15.40 Checking Network container of the same host
 CID: 7973931d3cb5 rCPU 0.0 rTX 0.0 rRX 0.0 hTot 2.0
 CID: dea1de565b72 rCPU 0.99 rTX 0.0 rRX 0.0 hTot 0.667
 CID: c2993ac4b336 rCPU 0.0 rTX 0.0 rRX 0.0 hTot 2.0
20:17:32- Host: 192.168.15.40 Container dea1de565b72 with lowest hTot
20:17:32- Host: 192.168.15.40 Pearson rRX 0.000 Pearson rTX 0.000
20:17:32- Host: 192.168.15.40 Pearson CPU and container TX - hTx: 0.334
20:17:32- Host: 192.168.15.40 Pearson CPU and container RX - hRX: 0.334
20:17:32- Host: 192.168.15.40 Pearson CPU and Network Relation is 0.667
20:17:32- Host: 192.168.15.40 Migrating container!
20:17:59- Host: 192.168.15.40 CPU%: 2
```

A.8 IPERF Server Output - External IPERF Test

```
root@nilson-Ufscar:# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 192.168.15.25 port 5001 connected
         with 192.168.15.40 port 56592
```

```
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-15.0 sec  8.32 GBytes  4.76 Gbits/sec
```

A.9 Monitor Module Output - Internal IPERF Test

```
21:55:10 - Host 192.168.15.37 CPU% 0 Memory% 93
21:55:10 - Container iperf IP 172.17.0.3 CPU% 0
21:55:10 - Container iperf IP 172.17.0.2 CPU% 0
21:55:37 - Host 192.168.15.37 CPU% 1 Memory% 92
21:55:37 - Container iperf IP 172.17.0.3 CPU% 1
21:55:37 - Container iperf IP 172.17.0.2 CPU% 0
21:56:04 - Host 192.168.15.37 CPU% 81 Memory% 93
21:56:04 - Container iperf IP 172.17.0.3 CPU% 10.44
21:56:04 - Container iperf IP 172.17.0.2 CPU% 71.17
21:56:31 - Host 192.168.15.37 CPU% 0 Memory% 93
21:56:31 - Container iperf IP 172.17.0.3 CPU% 0
21:56:31 - Container iperf IP 172.17.0.2 CPU% 0
21:56:58 - Host 192.168.15.37 CPU% 1 Memory% 94
21:56:58 - Container iperf IP 172.17.0.3 CPU% 0
21:56:58 - Container iperf IP 172.17.0.2 CPU% 0
```

A.10 Decision Module Output - Internal IPERF Test

```
21:55:12-Host: 192.168.15.37 CPU%: 0
21:55:39-Host: 192.168.15.37 CPU%: 1
21:56:06-Host: 192.168.15.37 CPU%: 81
21:56:06-Host: 192.168.15.37 CPU Threshold reached at 81%
21:56:06-Host: 192.168.15.37 Verify Container dea1de565b72 IP 172.17.0.2
21:56:06-Host: 192.168.15.37 Checking Network container of the same host
 CID: 7973931d3cb5 rCPU 0.0 rTX 0.0 rRX 0.0 hTot 2.0
 CID: dea1de565b72 rCPU 0.79 rTX 0.95 rRX 0.95 hTot 1.315
21:56:06-Host: 192.168.15.37 Container dea1de565b72 with lowest hTot
21:56:06-Host: 192.168.15.37 Pearson rRX 0.95 Pearson rTX 0.95
21:56:06-Host: 192.168.15.37 Pearson CPU and container TX - hTX: 0.658
21:56:06-Host: 192.168.15.37 Pearson CPU and container RX - hRX: 0.658
21:56:06-Host: 192.168.15.37 Pearson CPU and Network hTot is 1.315
21:56:06-Host: 192.168.15.37 Container has network activity in the host.
21:56:06-Host: 192.168.15.37 Migration cancelled!
21:56:33-Host: 192.168.15.37 CPU%: 0
```

```
21:57:00-Host: 192.168.15.37 CPU%: 1
21:57:27-Host: 192.168.15.37 CPU%: 1
```

A.11 IPERF Server and Client Output - Internal IPERF Test

In the client side we have:

```
root@c2993ac4b336:/# iperf -c 172.17.0.2 -t 15
------------------------------------------------------------
Client connecting to 172.17.0.2, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 172.17.0.3 port 56688 connected with 172.17.0.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-15.0 sec  20.2 GBytes  11.6 Gbits/sec
```

And in the server side:

```
root@dea1de565b72:/# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 172.17.0.2 port 5001 connected with 172.17.0.3 port 56688
[  4]  0.0-15.0 sec  20.2 GBytes  11.6 Gbits/sec
```

APPENDIX B – Container Migration Performance

For each Figure below, there are 3 bar charts with the following states:

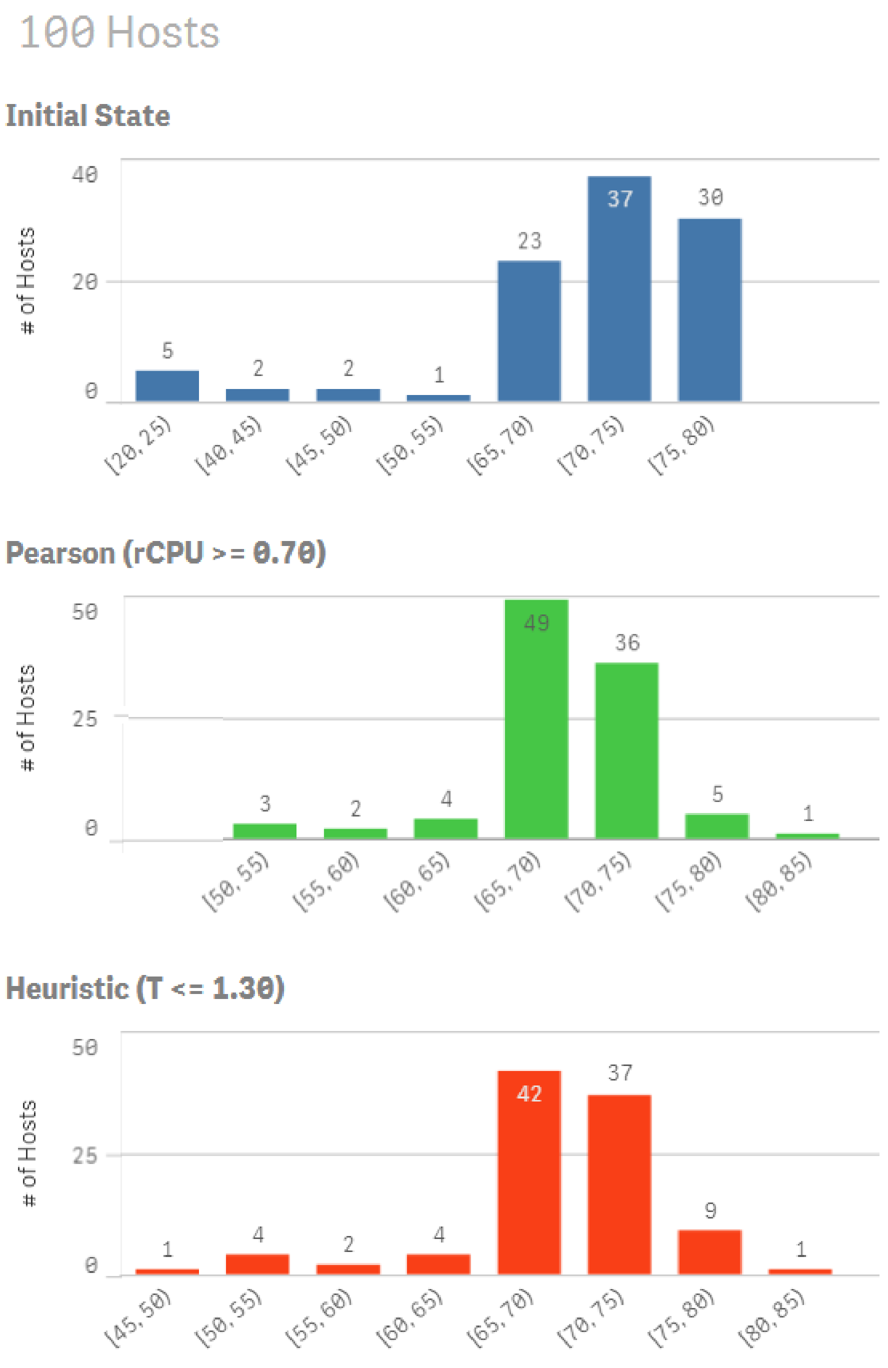
1. Initial State;
2. After applying heuristic h_{CPU}, with Pearson correlation coefficient for CPU (r_{CPU}) >= to 0.70;
3. After applying heuristic h_{Tot} with threshold $T_{heuristic} = 1.30$. Migration will occur when h_{Tot} is <= 1.30.

Tests were done for 100, 300, 500, 1000, 2500 and 5000 Hosts. All tests were repeated 30 times.

The y axis represents the number of hosts and the x axis represents the CPU consumption interval. For example, if in the x axis we have [70,75), it represents the number of hosts that have CPU consumption above or equal to 70% and less than 75%.

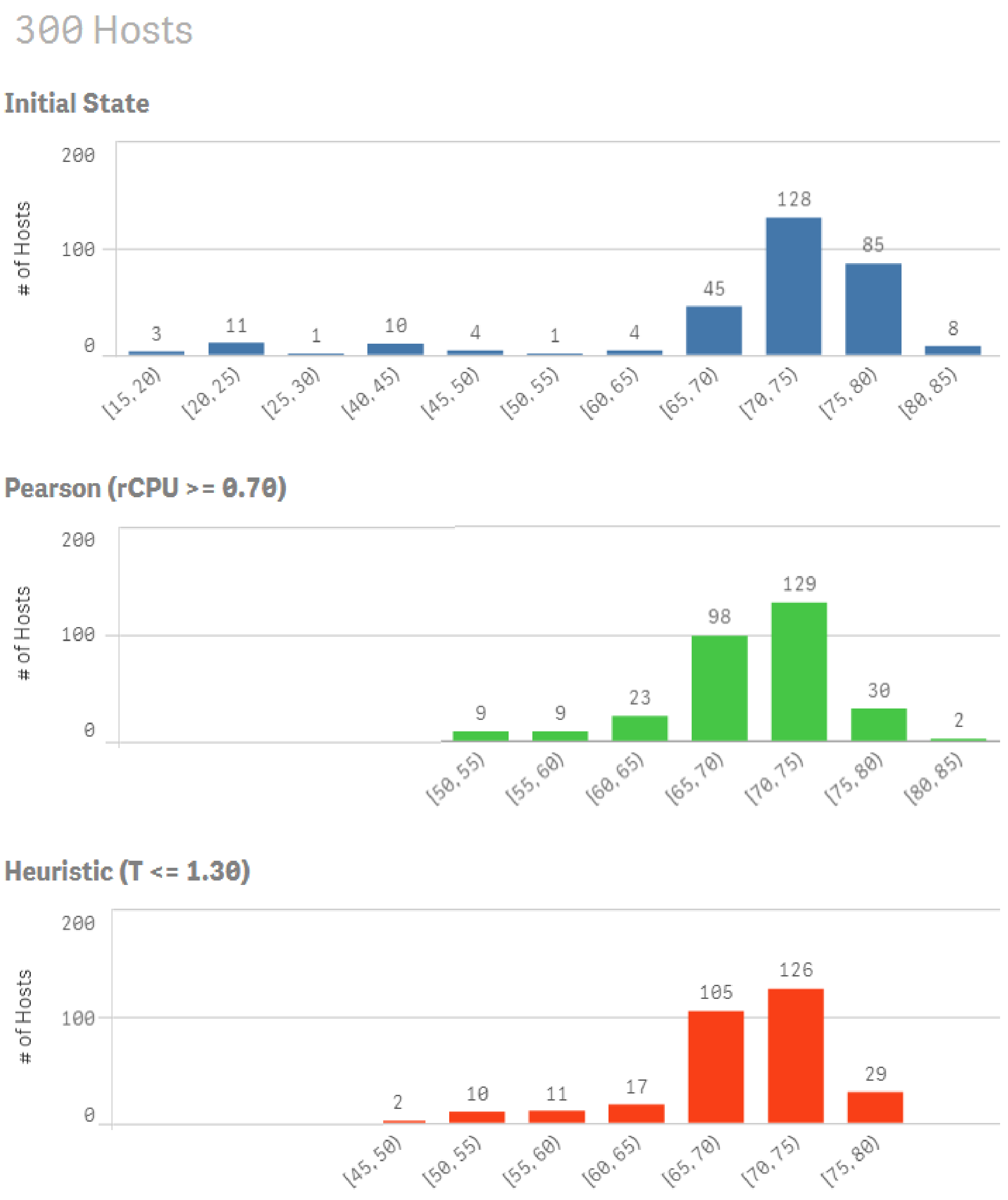
B.1 100 Hosts Simulation Results

Figure 30: Container Migration Results - 100 Hosts



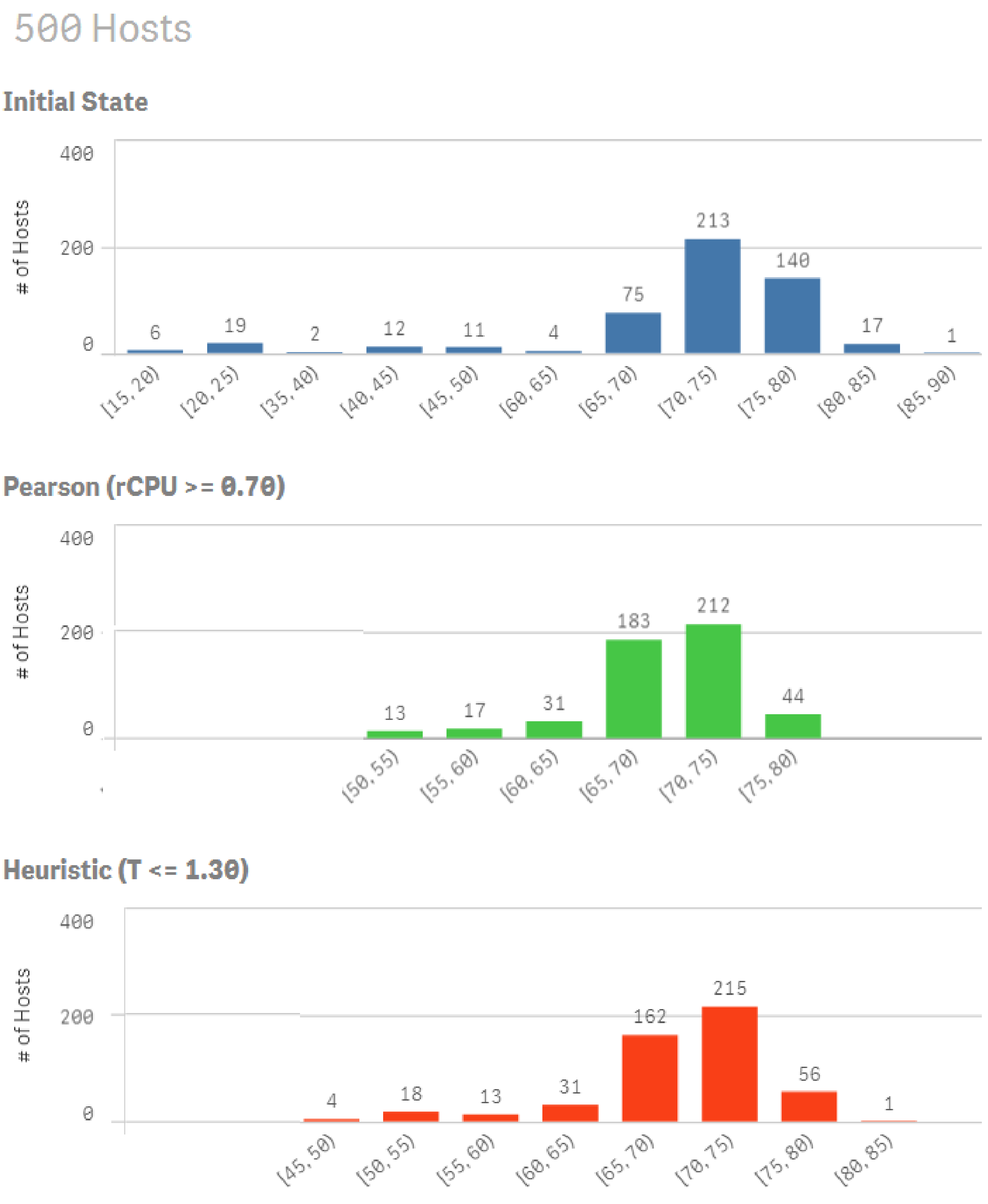
B.2 200 Hosts Simulation Results

Figure 31: Container Migration Results - 300 Hosts



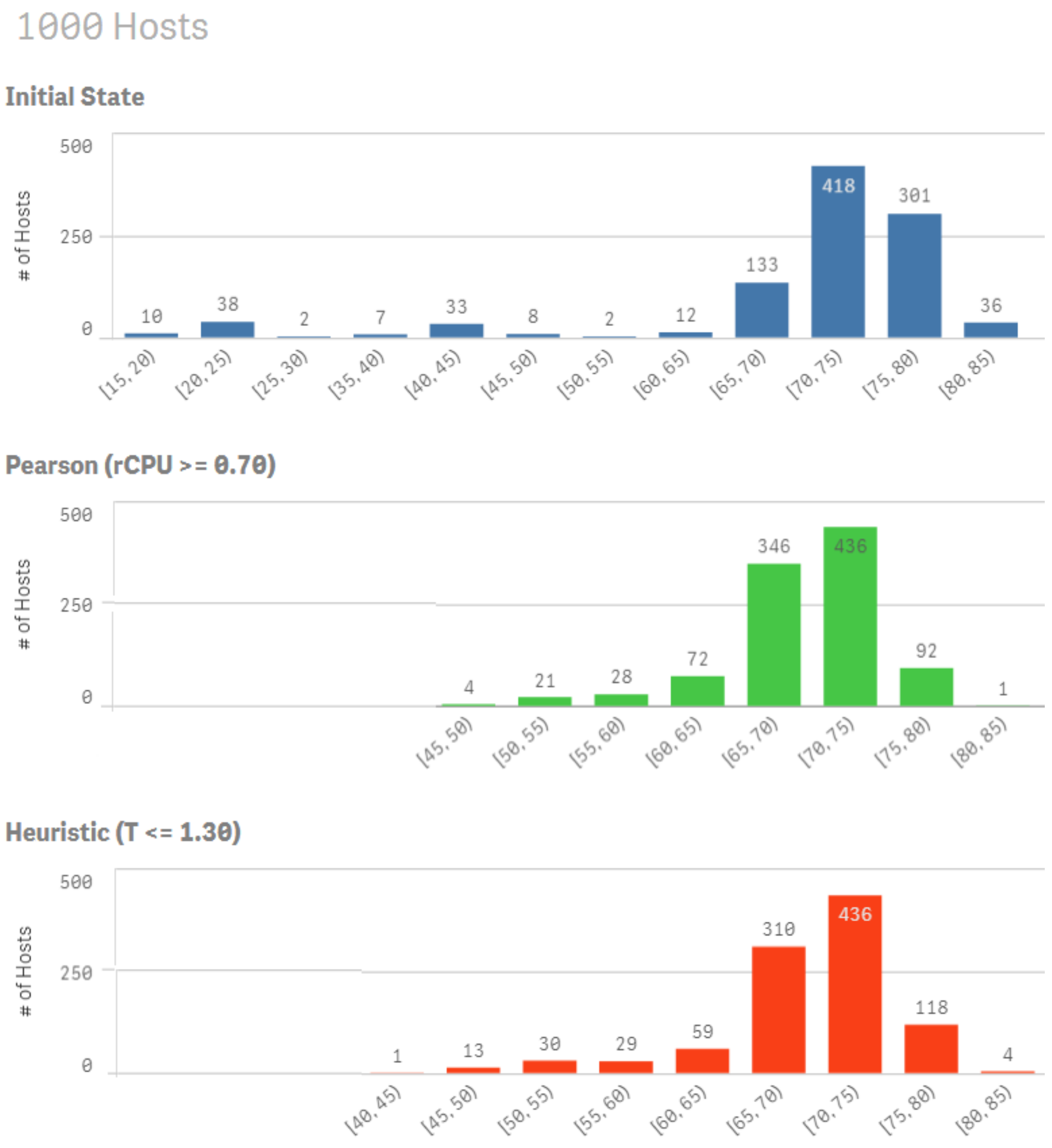
B.3 500 Hosts Simulation Results

Figure 32: Container Migration Results - 500 Hosts



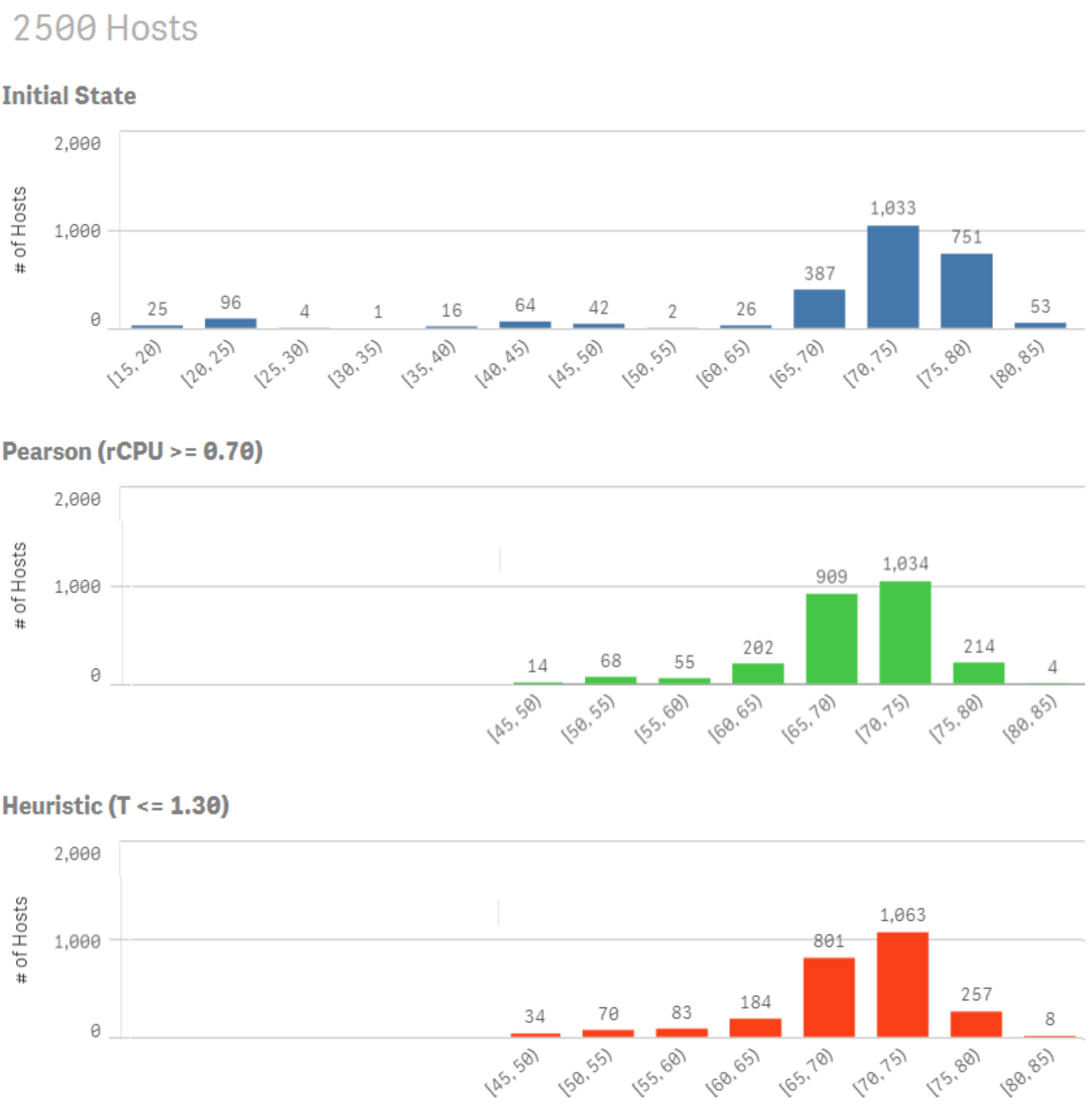
B.4 1000 Hosts Simulation Results

Figure 33: Container Migration Results - 1000 Hosts



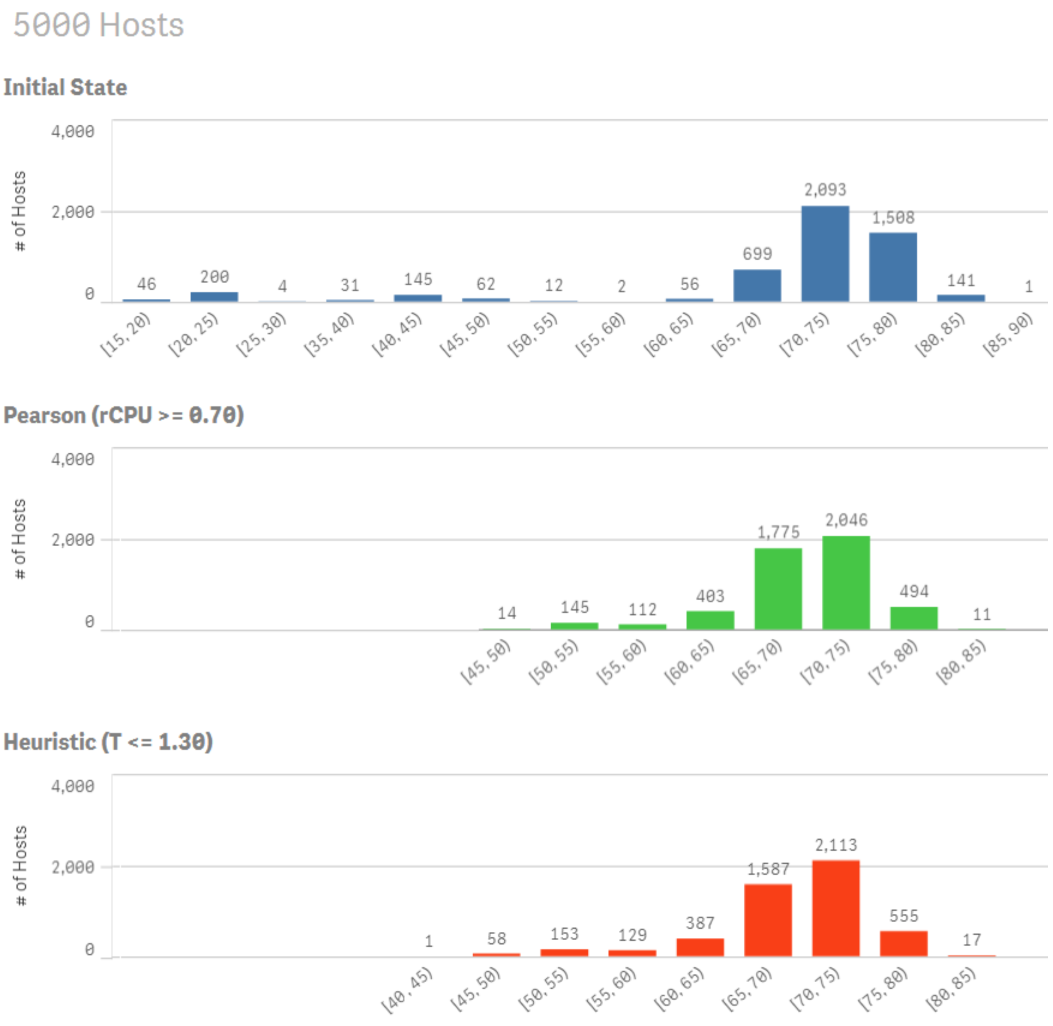
B.5 2500 Hosts Simulation Results

Figure 34: Container Migration Results - 2500 Hosts



B.6 5000 Hosts Simulation Results

Figure 35: Container Migration Results - 5000 Hosts



APPENDIX C – Supporting Tables

C.1 Initial State - CPU Table Statistics

Table 7: Initial CPU Status.

NumHosts	σ	$\bar{x}$	$\bar{x}+1\sigma$	$\bar{x}-1\sigma$
100	12.91	69.08	81.99	56.17
300	13.02	69.40	82.42	56.38
500	13.29	69.46	82.75	56.17
1000	13.39	69.53	82.92	56.14
2500	13.23	69.32	82.55	56.09
5000	13.30	69.43	82.73	56.13

C.2 After applying hCPU - CPU Table Statistics

Table 8: Status After Pearson CPU only.

NumHosts	σ	$\bar{x}$	$\bar{x}+1\sigma$	$\bar{x}-1\sigma$
100	4.66	69.08	73.74	64.42
300	5.23	69.40	74.63	64.17
500	4.99	69.46	74.45	64.47
1000	5.01	69.53	74.54	64.52
2500	5.13	69.32	74.45	64.19
5000	5.19	69.43	74.62	64.24

C.3 After applying hTot - CPU Table Statistics

Table 9: Status After Heuristic.

NumHosts	σ	$\bar{x}$	$\bar{x}+1\sigma$	$\bar{x}-1\sigma$
100	5.71	69.08	74.79	63.37
300	5.55	69.40	74.95	63.85
500	5.64	69.46	75.10	63.82
1000	5.80	69.53	75.33	63.73
2500	5.72	69.32	75.04	63.60
5000	5.75	69.43	75.18	63.68

C.4 Percentage reduction in the number of overloaded hosts

Table 10: Number of Hosts with CPU >= 70%.

NumHosts	**Initial**	**CPU**	**Heur**	**CPU%**	**Heur%**
100	67	42	47	37.31%	29.85%
300	221	161	155	27.15%	29.86%
500	371	256	272	30.99%	26.68%
1000	755	529	558	29.93%	26.09%
2500	1837	1252	1328	31.84%	27.71%
5000	3743	2551	2685	31.84%	28.26%