UiO : Department of Informatics
University of Oslo

Towards efficient and cost-effective live migrations of virtual machines

Fredrik Meyn Ung

Master's Thesis Spring 2015

Towards efficient and cost-effective live migrations of virtual machines

Fredrik Meyn Ung

18th May 2015

Abstract

As cloud computing and the use of virtual machines (VMs) have become a widespread phenomenon, a wide variety of optimization techniques have been invented for this field. One of them is *live migration*, which enables relocation of VMs between physical hosts without shutting them down.

Since this feature has been implemented and simplified in the majority of popular virtualization platforms, IT administrators have begun migrating VMs regularly. There are many reasons for this, including load balancing, server consolidation and disaster recovery.

This thesis have used a machine learning based algorithm to partition migration marked VMs into migration groups, with the goals of minimizing network load and lower the time consumption. A new algorithm, proposed by this thesis, is used to provide additional cost-optimization.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis concludes my two-year Network and System Administration master's degree at Oslo and Akershus University College of Applied Sciences. This program has been highly rewarding, but also demanding.

I would first and foremost like to thank my supervisor, Anis Yazidi, for all the meaningful and inspiring guidance provided to me throughout this thesis period. You are a brilliant person, and you have been very patient, optimistic and kind.

I would also likt to extend my gratitude to Hårek Haugerud and Kyrre Begnum for essential input and technical support.

I would also like to thank my friends and family for the strong support and encouragement these past two years.

Lastly, I would like to thank my patient and understanding girlfriend, Ninni Cecilie Eriksen. You have been very comforting, understanding and helpful.

Chapter 1

Introduction

Modern IT infrastructures run on virtual platforms. The physical resources which power these solutions can provide a layer of abstraction which allows Virtual Vachines (VMs) to run through the technology of *virtualization*. A very prominent form of virtualization enables a complete and fully usable *operating system* (OS) to run virtualized. This is principle is commonly referred to as *OS virtaualization* [1]. Users with sufficient access to these infrastructures can provision and start their own, fully controllable "machines". Most of the state-of-the-art data centers use this technology to provide flexibility and simplicity for their customers. The principle of hosting multiple virtual machines on a virtaulized environment is commonly referred to as *cloud computing* [2], and it is thought to be one of the most essential aspect of future computing [3]. Flexibility and scalability are important aspects of this form om computing.

The virtual machines are assigned attributes, such as memory (RAM) and disk space, from resource pools in the cloud. Portions of each resource type is bound to a VM. [4] The VMs themselves see these units as physically connected, and therefore fully utilizable by the system. A runnign VM will consume the physical resources very similarly to how a traditional computer would, but all the operations go through a management process at the virtualization layer. Through the means of virtualization, a VM will shrink the amount of resources available on the physical machine (PM) on which it resides. The distribution of VMs on PMs in a cloud is therefore significant in order to reduce the possibility of a physical host being overloaded. The placement is also important for resource balancing purposes [5].

Cloud provider is a term used to describe a company or an institution which provides customers or users with a resource pool from which they can create and administer VMs. These providers tend to offer users a web based front-end interface (sometimes referred to as a dashboard)

for simple creation, manipulation and destruction of VMs. The users do not control the VMs physical attributes, such as where the memory or disk data is stored. From the user's perspective, it's all just "in the cloud". This means that the cloud operating system, which the cloud provider has configured, places the VMs automatically, based on how the environment is configured. Other underlying mechanisms, such as network communication links and virtualization engines (called *hypervisors*) are also abstracted for the users. In this case, the provider is distributing the VMs as products in accordance with the *Infrastructure-as-a-Service* model (IaaS). They administer the cloud's concrete aspects (the hardware) and the virtualization functions, while the users manage their OS instances (VMs) completely on their own [6]. The users are therefore also responsible for maintaining and configuring VMs.

The servers and virtualization software is maintained by the cloud provider. Controlling and optimizing the placement of virtual machines, with respect to the physical hosts, is one of the challenges facing the IT administrators working in cloud computing institutions.

Live migration is a very unique feature to cloud computing that introduces the possibility to move VMs between different physical locations, without having to shut the instances down. This feature allows for an extremely low service disruption time (SDT), compared to the case where the machine is shut down, moved and then powered on again. SDT refers to the time period where a VM is unavailable due to being suspended at the source, and not yet up and running at the target destination.[7]. Popular virtualization providers like Xen [8], VMware (VMotion) [9], and Hyper-V [10] all offer live migration features in their solutions. Load balancing, system maintenance, server consolidation, and electricity saving are all reasons why live migration is a key feature in modern cloud environments [11] [12].

Load balancing refers to the principle of reducing the computational load on a physical unit, by relieving it of running tasks. In a case where a node is running out of resources due to hosting to many VMs, for example, moving some of them to another host is a form of load balancing. For multiple VMs cooperating in an applications, load balancing can be achieved by splitting application requests among them. The main goal of load balancing cloud based systems are to increase performance and prevent congestions on PMs [13]. For the latter, one can use live migration to relieve the PM.

In a system maintenance case, live migrations can be used to evacuate running VMs from a PM which needs to be powered off. PMs regularly need hardware upgrades or replacement of failing components. Once the PM is not serving any VMs, it can be powered off and managed by

technicians.

Some institutions which provide cloud computing services initially places VMs on PMs with load balancing in mind, but later find that they can get by with fewer running PMs. Live migration can be used to move VMs running on a lightly used PM over to other PMs [12], [14], so the PM can be shut down. This would lead to more "tightly packed", consolidated PM. A good reason for doing this is to save energy.

Another way to save on electricity, is to consume the same amount (avoiding consolidation), but at a lower monetary cost. Live migration can be used to move VMs over the *Wide Area Network* (WAN) to another data center based in a geographical area where electricity can be bought at a lower price, or is produced in a more environment friendly way. This principle of transferring nourishment demanding objects (VMs demanding electricity) to a more cost-effective placement is referred to as the *follow-the-sun* convention [15].

There are a lot of variations when it comes to the mechanics behind the different live migration techniques proposed in the research field of computer science. It is clear that a carefully thought through approach for scheduling the VMs in a migration task can benefit the overall migration performance, and subsequently yield other positive effects. Fast migrations are important to achieve better adaptability of resource utilization and is essential for moving VMs quickly, while keeping them running for the users. [16].

This thesis aims to construct and test a framework for making live migrations more efficient and application friendly. The goals are to lower the time it takes to complete a live migration of multiple running VMs, and to reduce the network traffic amount caused by migrations. The framework will be based on *machine learning*, more specifically a *learning automata* system.

1.1 Problem statement

"How can we achieve cost-effective and efficient live migrations of virtual machines in a cloud environment?"

1.1.1 Efficiency

The migration scheme should be able to perform migration quickly, and with certainty that it will complete. If this is achieved, *efficiency* is is

accomplished.

1.1.2 Cost effectiveness

In chapter 2, the workings of different live migration techniques will be outlined. The proposed solution in this thesis will focus on minimizing performance degradation of communication intensive applications running on virtual machines. *Cost effectiveness* is considered to be archived if this impact is minimized.

Chapter 2

Background

2.1 Virtualization concepts

VMware, which is one of the world's largest companies specializing in virtualization describes it simply as *"the separation of a service request from the underlying physical delivery of that service"* [17].

What WMware points to with the term "underlying delivery" of a service is that the actual execution of the instruction initiated by a VM, for example a processor request or a memory operation, is handled by a proxy. The VM may or may not be aware or that this is happening, depending on the mode of virtualization used in the infrastructure.

One can build a virtualization platform in a number of different ways. In order to comprehend the differences and specifications behind each of them, there are some often used terms which require explanation:

- The **guest** is a virtual machine running on top of a virtualization infrastructure. The guest is usually, in itself, a fully functional operating system (OS).
- The **host** is a machine which delegates resources to a guest.
- The **hypervisor** is the abstraction layer between the physical hardware and the host. The hypervisor is sometimes referred to as a "virtual machine manager" (VMM). This is the proxy that carries out instructions on behalf of the guest.

There are essentially two widely used virtualization architectures available today - *full virtualization* and *paravirtualization*.

2.1.1 Full virtualization

This is, design wise, the simplest form of virtualization. A hypervisor is installed directly onto the physical unit, which has one or more of each of the standard computer devices installed - network card, hard drive, processor, and memory (RAM). The hypervisor type which is used here is a *bare metal hypervisor*, which is also referred to as "type 1". The OS that the guest uses is unmodified, and therefore sees the devices as "real", which means that the OS cannot determine that it is being virtualized. Behind the scenes, the guest OS is allocated resources from the PM. The fact that the hypervisor has direct access to the hardware, can make this mode of operation more flexible and efficient [17].

However, this virtualization mode relies on *binary translation* of CPU instructions between the guest and host, as it simulates all the underlying hardware to the guests, which can have negative effects on efficiency in terms of overhead [18]. Full virtualization provides guests with a complete blue print of a computer system. This means that the hypervisor must serve all components, including a virtualized BOIS and a memory shadow table [19].

If security is an important concern, it is recommended to use a full virtualization suite, as they can provide thorough isolation of running applications. The hypervisor can split resources into pools, and it is common practice to only allow one pool to a VM and only host one application per VM. If a PM is running multiple VMs which together serve many applications, the occurrence of a security breach can cause a lot of damage [20].

2.1.2 Paravirtualization

This way of configuring virtualization relies on modifying the OS of the guest [21] and adding the hypervisor layer on top of the host. The hypervisor then exists between the host and the guests. This requires a "type 2" hypervisor, which is also called a *hosted hypervisor*, and hence this type of virtualization is also known as *hosted virtualization*. It is essentially a program that runs on the host OS. In this architecture, each VM can be aware that it is being run virtualized, as modifications have to be made on the guest OS. With modern OS images, this happens automatically during installation.

The resource requests from the guest has to go through the hypervisor on the host before reaching the physical resources, which can make it more overhead-heavy. However, since the VMM is small and simple in

paravirtualization, guests can often achieve "near-native" performance [22]. This means that a VMs is very close to executing instructions as fast as a physical computer with the same specifications. The guests do not run on emulated devices, as in full virtualization, but rather access resources through special device drivers [23].

In the x86 architecture, there is the concept of *privilege rings*. An x86 OS typically runs in the most privileged level, "ring 0", while rings 1, 2 and 3 are lesser privileged modes ("user level rings"). An OS needs to perform it's executions in ring 0. When a VM initiates such an instruction (called a *hypercall*), it is captured and executed by the hypervisor running alongside the host OS, on behalf of the guest.

Figure 2.1 compares the two architectures.

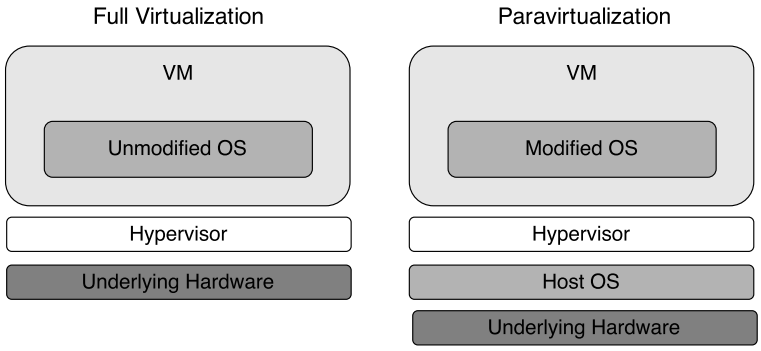

Figure 2.1: Simplified architecture of para- and full virtualization

2.1.3 QEMU and KVM

QEMU is a software which provides machine emulation and virtualization. The program operates in different modes, including "full system emulation", where it is able to host a wide variety of machine types as guests. This is achieved by a recompiler in the QEMU software which translates binary code destined for one CPU type to another. QEMU also contains many emulators for other computer components, such as network cards, hard drives and USB. In other words, QEMU is a hypervisor which uses emulation to provide virtualization capabilities [24].

KVM, which stands for Kernel Virtual Machine, is a variant of the QEMU program and hence also a hypervisor. It is built into the Linux OS, and transforms the standard Linux kernel into a hypervisor, if activated. It runs guests as if they were processes on the host, which means they can be controlled like any other program. They are assigned process IDs with which KVM can interact. The KVM program can be started from the Linux command line, which could make one think it is a type 2 hypervisor (since

it is running "on top" of an OS), but the VMs on KVM actually run on bare metal, effectively making it a type 1 (bare metal virtualization). [25]. The discrepancy of which solutions lie within the different architectures can sometimes be unclear [26]. KVM is reliant on the host having installed a processor which supports virtualization features, such as one from the "Intel VT" or the "AMD-V" series.

2.1.4 Libvirt

Libvirt is a Red Hat developed application programming interface (API) and program daemon for managing virtualization services. It allows you to connect to VM remotely through the network and to launch VMs via the *virsh* command line interface. Libvirt can set up a virtual network switch on the host and have the guests connect to it and pass their network traffic through. The virtual switch is an optional feature. By default it performs network address translation (NAT) using the masquerade options, so that all VMs which connect to external sources will look like traffic sourced from the IP of the virtual bridge (*virbr0* interface by default). The switch program can also operate in "routed mode", where it puts VMs on a separate subnet, unknown to the host. In this mode, computers from the outside can access the VMs through a static route on on the host, which forwards traffic destined to the VMs to the bridge interface. A third option in Libvirt is to use an existing virtual switch on the host where a physical interface is connected. Each VM would then get a *tap-interface* attached to the existing bridge. This is a virtual interface seen as a physical by the host, and as a switch port by the switch software.

When setting up hosted virtualization on a Linux platform, it is common to use a combination of QEMU, KVM and Libvirt.

2.2 Live migration

Current hypervisors in virtual cloud environments include different functionality for migrating virtual machines. Migration is performed either *sequentially* or in *parallel*. The sequential method migrates one VM at a time. In parallel migration, multiple MVs assigned to the same migration task are moved simultaneously [27]. The main focus of in this chapter is the parallel approach, which is where the potential for speed optimization lies. This is essentially about optimizing the available network bandwidth.

2.2.1 Pre-copy Migration

The most common way for virtual machine migration is the *pre-copy method* [28]. During such a process, the complete disk image of the VM is first copied over to the destination. If anything was written to the disk during this process, the changed disk blocks are logged. Next, the changed disk data is migrated. Disk blocks can also change during this stage, and once again the changed blocks are logged. Migration of changed disk blocks are repeated until the generation rate of changed blocks are lower than a given threshold or a certain amount of iterations have passed [29] [30]. After the virtual disk is transferred, the RAM is migrated, using the same principle of iteratively copying changed content. Next, the VM is suspended at the source machine, and resumed at the target machine. The states of the virtual processor are also copied over, ensuring that the machine is the very same in both operation and specifications, once it resumes at the destination. The rate at which disk or memory changes during the migration is referred to as *dirty rate* (DR).

It is important to note that the disk image migration phase is only needed if the VM doesn't have its image on a network location, such as an NFS share, which is quite common for data centers. [31].

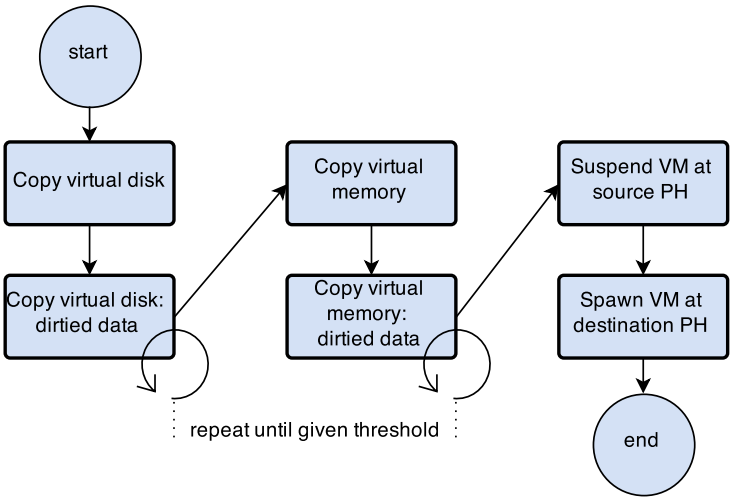

Figure 2.2: Pre-copy method for live migration

2.2.2 Post-copy Migration

This is the most primitive form of virtual machine migration [32]. The basic outline of the post-copy method is as follows. The VM is suspended at the source PM. The minimum required processor states, which allows the VM to run, is transferred to the destination PM. Once this is done, the

VM is resumed at the destination PM. This first part of the migration is common to all post-copy migration schemes. Once the VM is resumed at the destination, memory pages are copied over the network as the VM requests them, and this is where the post-copy techniques differ [33]. The main goal in this latter stage is to push the memory pages of the suspended VM to the newly spawned VM, which is running at the destination PM. In this case, the VM will have a short SDT, but along performance degradation time (PDT).

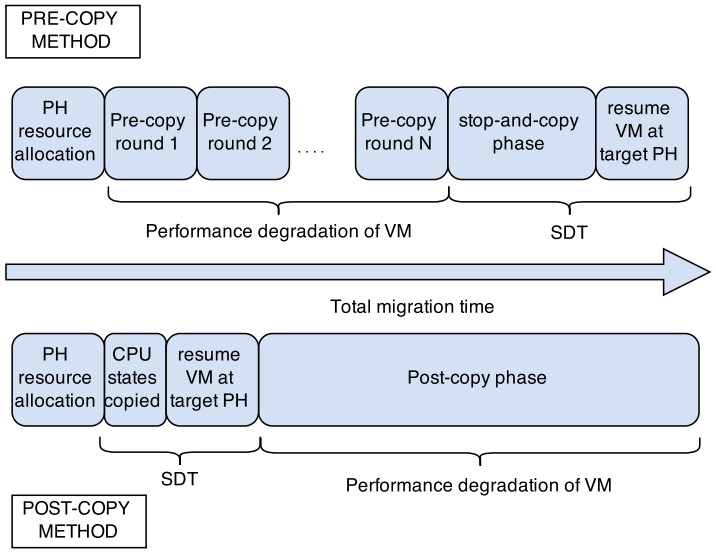

Figure 2.3: Pre- vs. Post-copy migration sequence

Figure 2.3 illustrates the difference between these two migration techniques. The diagram only depicts memory and CPU state transfers, and not the disk image of the VM. The latter is performed similarly in both the migration techniques, and does not affect the performance of the VM, and is therefore disregarded from the comparison. The "performance degradation of VM" in the pre-copy method refers to the hypervisor having to keep track of the dirty pages; the RAM which has changed since the last pre-copy round. In the post-copy scenario, the degradation is greater and lasts longer. In essence, the post-copy method activates the VMs on the destination faster, but all memory is still located at the source. When a VM migrated with post-copy requests a specific portion of memory not yet local to the VM, the relevant memory pages will have to be pushed over the network. The "stop-and-copy" phase in the pre-copy method is the period where VM is suspended at the source PM and the last dirtied memory and CPU states are transferred to the destination PM. SDT is the time where the VM is inaccessible.

2.3 Issues with live migration

Moving virtual machines between physical hosts has its challenges. Research papers propose different ways to tackle the various impact this process has on resources. The following sub sections show which concerns are commonly addressed.

Application performance degradation

A multi-tier application is an application which communicates with many VMs simultaneously. These are typically configured with the different functionality spread over multiple VMs [34]. For example might the database part of an application be stored on one set of VMs, and the web server functionality on another set. In a scenario where an entire application is to be moved to a new site which has a limited bandwidth network link to the original site, the application will deteriorate in performance during the migration period for the following reason. If one of the application's member VMs are resumed at the destination site, any traffic destined to that machine will be slower than usual due to the limited inter site bandwidth, and the fact that the rest of the application is still running at the source site. Several researchers have proposed ways of handling this problem of geographically split VMs during migration. Like Zheng et al. [27], this thesis will refer to this as the *split components problem*.

Network congestion

Live Migrations which take place within a data center, where no VMs end up at the other end of a slow WAN-link, are not as concerned about the performance of running applications. It is common to use *management links* in production cloud environments, which allow management operations like live migrations to proceed without affecting the VMs and their allocated network links. The occurrence of some amount of SDT is unavoidable. However, such an implementation could be costly. In a setting where management links are absent, live migrations would directly affect the total available bandwidth on the links it uses. One issue that could arise from this, is that several migrations could end up using the same migration paths, effectively overflowing one or more network links, and hence slow the performance of multi-tiered applications.

Migration time

In a scenario where a system administrator needs to shut down a physical machine for maintenance, all the VMs currently running on that machine will have to be moved, so that they can keep serving the customers. For such a scenario, it would be favorable if the migration took the least time possible. In a case where the migration system is only concerned about fast migration, optimal target placement of the VMs might not be attained.

2.4 Relevant research

Sequencer (CQNCR)

Bari et al. [35] have created a system called CQNCR (read "sequencer"), which goal is to make a planned migration perform as fast as possible, given a source and target organization of the VMs. The tool created for this research focuses in intra-site migrations. The researches claim to be able to increase the migration speed significantly, by reducing total migration time by up to 35%. They also introduce the concept of *virtual data centers* (VDCs) and *residual bandwidth*. In practical terms, a VDC is a logically separated group of VMs and their associated virtual network links. As each VM has a virtual link, it too needs to be moved to the target PM. When this occurs, the bandwidth available to the migration process changes. The CQNCR-system takes this continuous change into account and does extended recalculations to provide efficient bandwidth usage, in a parallel approach. The system also prevents potential bottlenecks when migrating.

The COMMA system

Another system, COMMA [27], groups VMs together and migrates one group at a time. Within a group are VMs which have a high degree of *affinity*; VMs which communicate a lot with each other. After the migration groups are decided, the system performs inter- and intra-group scheduling. The former is about deciding the order of the groups, while the latter optimizes the order of VMs within each group. The goal of COMMA is to address the issue mentioned in section 2.3. The main function of COMMA is to migrate associated VMs at the same time, in order to minimize the traffic which has to go through a slow network link. The system is therefore especially suitable for inter-site migrations. It is structured so that each

VM has a process running, which reports to a centralized controller which performs the calculations and scheduling.

The COMMA system defines the impact as the amount of inter-VM traffic which becomes separated because of migrations. In a case where a set of VMs, $\{VM_1, VM_2, ..., VM_n\}$, is to be migrated the traffic levels running between them are measured and stored in matrix TM. Let the migration completion time for vm_i be t_i. Equation 2.1 represents the impact that their system should minimize.

$$impact = \sum_{i=1}^{n} \sum_{j>i}^{n} |t_i - t_j| \cdot TM[i,j] \tag{2.1}$$

VMbuddies

The VMbuddies system [36] also address the challenges in migrating VMs which is used by multi-tier applications. The authors formulate the problem as a *correolated VM migration problem*, and is tailored towards VM hosting multi-tier applications. Correlated VMs are machines that work closely together, and therefore send a lot of data to one another. An example would be a set of VMs hosting the same application, where two or three VM subsets perform different roles in different tiers, as described in section 2.3. Their work lead to the implementation of an algorithm for optimizing network bandwidth and a mechanism for reducing the cost of a live migration. Tests the have conducted show clear improvements compared to current migration techniques, including a 36% reduced migration time, compared to Xen.

Clique Migration

A system called Clique Migration [37], also migrates VMs based on their level of interaction, and is directed at inter-site migrations. When Clique migrates a set of VMs, the first thing it does is to analyze the traffic patterns between them and try to profile their affinity. This is similar to the COMMA system. It then proceeds to create groups of VMs. All VMs within a group will be initiated for migration at the same time. The order of the groups is also calculated to minimize the cost of the process. The authors define the migration cost as the volume of inter-site traffic caused by the migration. Due to the fact that a VM will end up at a different physical location (a remote site), the VMs disk is also transferred along with the RAM.

Time bound migration

Chanchio and Thaenkaew [16] have created a *time-bound thread-based live migration* (TLM) technique. Their focus was to handle large migrations of VMs running RAM-heavy applications, by allocating additional processing power at the hypervisor level to the migration process. TLM can also slow down the operation of such instances to lower their dirty rate, which will help in lowering the total migration time. The completion of a migration in TLM is always within a given time period, proportional to the RAM size of the VMs.

All the aforementioned solutions migrate groups of VMs simultaneously, in one way or another, hence utilizing parallel migration to lower the total migration time. Lu et al. [11] have found, in very recent research, that when running parallel migrations within data centers, an optimal sequential approach is preferable. The have implemented a migration system called vHaul which does this. They argue that the application performance degradation caused by split components is caused by many VMs at a time, whereas only a single VM would cause degradation if sequential migration is used. However, the shortest possible migration time is not reached because vHaul's implementation of a *no-migration interval* between each VM migration. During this small time period, the pending requests to the moved VM is answered, which reduces the impact of queued requests during migration. vHaul is optimized for migrations within data centers which have dedicated migration links between physical hosts.

2.5 Measuring migration impact

It is common view the live migration sequence into three parts, when talking about the pre-copy method:

1. Disk image migration phase
2. Pre-copy phase
3. Stop-and-copy phase

The last two phases are shown in figure 2.3.

2.5.1 Total migration time

Bari et al. [35] and Mann et al. [38] use the following mathematical formulas to calculate the time it takes to complete the different parts of the migration. Let W be the disk image size in megabytes (MB), L the bandwidth allocated

to the VM's migration in MBps and T the predicted time in seconds. X is the amount of RAM which is transferred in each of the pre-copy iterations.

The time it takes to copy the image from the source PM to destination PM is:

$$T_i = W/L \tag{2.2}$$

Once the VM's image is copied over, the pre-copy phase is initiated. It's time duration can be calculated as follows:

$$T_{p+s} = \frac{M \cdot \frac{1-(R/L)^n}{1-(R/L)}}{L} \tag{2.3}$$

The stop-and-copy period is the last phase of a pre-copy live migration, where a VM is suspended at the source PM and resumed at the destination PM. The completion time for this final phase is given by:

$$T_s = M/L \cdot (R/L)^n \tag{2.4}$$

The n in the equations 2.3 and 2.4 is given by:

$$n = \min(\lceil \log_{R/L} \frac{T \cdot L}{M} \rceil, \lceil \log_{R/L} \frac{X \cdot R}{M \cdot (L-R)} \rceil) \tag{2.5}$$

2.5.2 VM traffic impact

Liu and He [36] provide the following formulas to describe total the total network traffic amount and total migration duration, respectively. The number of iterations on the pre-copy phase (n) is not defined here, but is calculated based on a given threshold.

Variable	Description
V	Total network traffic during migration
T	Time it takes to complete migration
N	Number of pre-copy rounds (iterations)
M	Size of VM RAM
d	Memory dirty rate during migration
r	Transmission rate during migration

Table 2.1: Variables used in formulas in the VMbuddies system

They first derive general expressions for v_i and t_i to:

$$v_i = \frac{M \cdot d^i}{r^i} \tag{2.6}$$

$$t_i = \frac{M \cdot d^i}{r^{i+1}} \tag{2.7}$$

Then the total network traffic during migration becomes:

$$V = \sum_{i=0}^{n} v_i = M \cdot \sum_{i=0}^{n} \frac{d_i}{r_i} \tag{2.8}$$

Table 2.1 denotes the variables used in equation 2.8.

Another possible metric for measuring how impactful a migration was, is to look at the total amount of data the migrating VMs have sent between the source and destination PMs during the migration process. This would vary depending on how the scheduling of the VMs is orchestrated.

2.6 Bin packing

The mathematical concept of bin packing centers around the practical optimization problem of packing a set of different sized "items" into a given number of "bins". The constraints of this problem is that all the bins are of the same size and that none of the items are larger than the size of one bin. The size of the bin can be thought of as its capacity. The optimal solution is the one which uses the smallest number of bins [39]. This problem is known to be NP-hard, which in simple terms means that finding the optimal solution is computationally heavy. There are many real-life situations which relate to this principle. For instance, using the smallest number of boxes when moving things from one location to another, by packing them as tightly as possible.

In VM migration context, one can regard the VMs to be migrated as the items and the network links between the source and destination host as bins. The capacity in such a scenario would be the amount of available bandwidth which the migration process can use. Each VM requires a certain amount of bandwidth in order to complete in a given time frame. If a VM scheduling mechanism utilized parallel migration, the bin packing problem is relevant because the start time of each migration is based on calculations of when it is likely to be finished, which in turn is based on

bandwidth estimations. A key difference between traditional bin packing of physical objects and that of virtual machines on network links, is that the VMs are infinitely flexible. This is show in figure 2.4. In this hypothetical scenario, VM1 is being migrated between time t_0 and t_4, and using three different levels of bandwidth before completion, since VM2 and VM3 are being migrated at times where VM1 is still migrating.

The main reason for performing parallel migrations, is to utilize bandwidth more efficiently, but it could also be used to schedule migration of certain VMs at the same time.

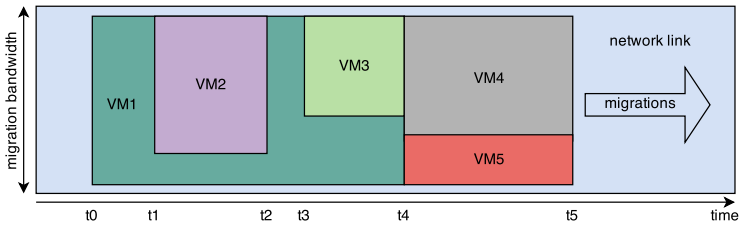

Figure 2.4: Bin packing in VM context

2.7 Graph partitioning

Graph partitioning refers to a set of techniques used for dividing a network of *virtecies* and *edges* into smaller parts. One appliance for such a technique could be to group VMs together in such a way that the VMs with a high degree of affinity are placed together. This could mean, for example, that they have a lot of network traffic running between them. In graph partitioning context, the network links between virtual machines would be the edges and the VMs vertices. Figure 2.5 shows an example of the interconnection of nodes in a network.

The "weight" in the illustration could represent the average traffic amount between two VMs in a given time interval, for example. This can be calculated for the entire network, so that every network link (edge) would have a value. The "cut" illustrates how one could devide the network into two parts, which means that the cut must go through the entire network, effectively crossing edges so that the output is two disjoint subsets of nodes.

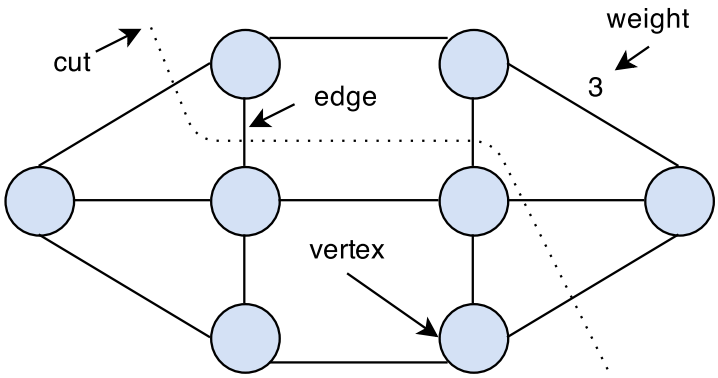

Figure 2.5: Nodes connected in a network

If these nodes were MVs marked for simultaneous migration, and the sum of the their dirty rate was greater than the bandwidth available for the migration task, the migration will not converge [27]. It is therefore imperative to divide the network into smaller groups of VMs, so that each group is valid for migration. For a migration technique which uses VM grouping, it is prudent to cut a network of nodes (which is too large to migrate all together), using a *minimum cut* algorithm, in order to minimize the traffic that goes between the subgroups during migration [37]. The goal of a minimum cut, when applied to a weighted graph, is to cut the graph across the vertices in the way that leads to the smallest sum of weights. The resulting subsets of the cut are not connected after this.

In a similar problem called the *uniform graph partitioning problem*, the number of nodes in the resulting two sets have to be equal. This is known to be NP-complete [40], which means that there is no efficient way of finding a solution to the problem, but it is takes very little time to verify if a given solution is in fact valid.

2.8 Learning Automata partitioning

Multiple algorithms have been proposed for solving the graph partitioning problem. In a small scale, as the example shown in figure 2.5, the time required to computationally discover the minimum cut is very low, as there are few possibilities (cuts over vertices) which lead to exactly four nodes in each subset. Note that the referenced figure's cut is not a uniform graph cut resulting in two equal sized subsets, nor shows the weight of all the vertices. It merely illustrates a graph cut.

To exemplify the complexity growth of graph cutting, one could regard two networks, where one has 10 nodes and the other has 100. The amount of

valid cuts and hence the solution space in the former case is 126, and 10^{29} for the latter [40]. This clearly shows that a brute force approach would use a lot of time finding the optimal solution, when there are many vertices. A number of heuristic and genetic algorithms have been proposed in order to try and find near optimal solutions to this problem.

Learning Automata is a science which divisions under the scope of adaptive control in uncertain and random environments. Adaptive control is about managing a controller so that it can adapt to changing variables using adjustment calculations. The learning aspect refers to the way the controller in the environment gradually starts to pick more desirable actions, based on feedback. The reaction from the environment is to give either a reward or a penalty for the chosen action. In general control theory, control of a process is based on the control mechanism having complete knowledge of the environment's characteristics, meaning that the probability distribution in which the environment operates is deterministic, and that the future behavior of the process is predictable. Learning automata can, over time and by querying the environment, gain knowledge about a process where the probability distribution is unknown [41].

In a *stochastic* environment, it is impossible to accurately predict a subsequent state, due to the non-deterministic nature of it. If a learning automata mechanism is initiated in such an environment, one can gradually attain more and more certain probabilities of optimal choices. This is done in a query-and-response fashion. The controller has a certain amount of available options, which initially has an equal opportunity of being a correct and optimal choice. One action is chosen, and the en environment responds with either a reward or a penalty. Subsequently, the probabilities are altered based on the response. If a selected action got rewarded, the probability of this same action should be increased before the next interaction (iteration) with the system, and lowered otherwise. This concept can be referred to as *learning automation.* [41]

The following is an example of how learning automation would work. Consider a program which expects an integer n as input, and validates it if $0 < n < 101$ and $n \bmod 4 = 0$. A valid input is a number between 1 and 100, which is divisible by 4. Now, let's say that the learning automation only knows the first constraint. Initially, all the valid options (1-100) has the probability value of 0.01 each, and the automata chooses one at random. A penalty or reward is received, and the probabilities are altered, with the constraint that $\sum_{x \in A} f_X(x) = 1$, where x is a valid option. After many iterations, all the numbers which the environment would validate should have an approximately equal probability, higher than the rest.

Oommen, Croix et al. [40] have proposed a learning automata based

algorithm for splitting any graph into equal sized subgroups, where the result is such that the sum of the edges that go between the subgroups is as small ass possible. In other words, the proposed algorithm ensures that a minimum cut has been reached between any two resulting subgroups of the input graph.

Chapter 3

Approach

This chapter will outline an overview of how the experiments for this thesis will be conducted, as well as what can be expected from them. The goal is that the experiments will give answers to the problem statement described below.

"How can we achieve cost-effective and efficient live migrations of virtual machines in a cloud environment?"

As the introduction specifies, a scheme for migrating VMs is needed. It should try to minimize the impact which migrations will have on running multi-tiered applications. Section 3.1 will clarify how an envisioned system will behave.

3.1 Migration Design

A matrix will be used to illustrate the traffic amount between VMs marked for migration. In the following example we have extracted the average traffic amount running between a selection of 10 VMs. The top-left to bottom-right diagonal is all zeros, since loopback traffic on the machines in not considered. Let VMs be VM and n the number of VMs in the group. The variables i and j are the number of columns and rows, respectively. We could then end up with a $n \cdot n$ matrix like this:

$$\begin{bmatrix} VM_{1,1} & VM_{1,2} & ... & ... & VM_{1,n} \\ VM_{2,1} & ... & ... & ... & VM_{2,n} \\ ... & ... & ... & ... & ... \\ ... & ... & ... & VM_{i,j} & ... \\ VM_{n,1} & VM_{n,2} & ... & ... & VM_{n,n} \end{bmatrix} \tag{3.1}$$

The total network traffic running between a pair of VMs is $VM_{i,j} + VM_{j,i}$. It will not be considered traffic if a VM talks to itself, meaning $VM_{j,i} = 0$, where $j = i$.

Both dedicated management network links and non-dedicated links will be used when testing migrations. For the latter, the migration traffic will have to share the bandwidth with VMs which are communicating with each other from different hosts, meaning that both the VMs dirty rates and shared traffic will affect migration performance. It is likely that the effectiveness of the migration scheme will be more evident when migration traffic and inter-VM traffic runs on the same link. This is because the grouping of the VMs are made with a minimum cut method, such that a minimum amount of inter-group traffic is present on the migration link during the process.

A migration over a dedicated link path, where no inter-VM traffic is sent, can be expected to complete faster, but is not necessary to measure to amount of separated traffic caused by migrations, as this would be the same in both cases. Separated traffic will occur in any migration scenario where VMs are communicating during the migration process, and where not all VMs finish migration at the same time.

3.1.1 The Learning automata

This section will simplistically express the nature of the algorithm found in the paper 'Graph partitioning using learning automata' [40], which will be used in this project in order to create VM groups. It should, in theory, output equal size subgroups where the sum of the weight on the vertices between them is close to minimal.

The algorithm uses the concept of similarity and dissimilarity, and moves nodes between subgroups and between *states* within a subgroup. Similar nodes belong in the same subgroup. Since the system is learning automata based, it continuously rewards or penalizes nodes by comparing them, as described in in the example in section 2.8. The nodes being similar and in different subgroups, yields a penalty (because they are not in the same subgroup) or a reward, if they are in fact residing in the same one. There are two additional cases, and all four as shown in table reftab:reward-penalty.

	Similar	Dissimilar
Same Subgroup	reward	penalty
Different Subgroup	penalty	reward

Table 3.1: The four cases of reward and penalty

The state in a subgroup is a measure of certainty, where the internal state is the most certain. The figure 3.1 shows that a node can be sent to a more secure state, a less secure state, or to another subgroup. The latter will be explained later in this section.

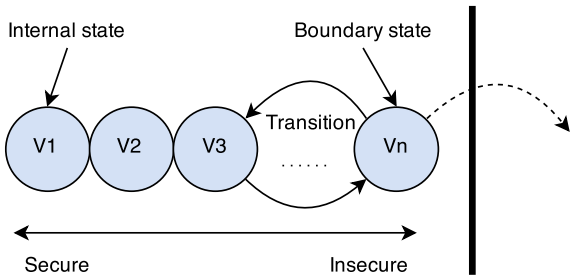

Figure 3.1: States within a subgroup

Both rewards and penalties can move nodes towards and away from the boundary states. This all depends on which of the four cases is matched when the comparison takes place. It is important to know that multiple nodes can share the same state in the same subgroup. In fact, the initialization of the algorithm places all nodes in the boundary state.

The algorithm iteratively picks two and two random nodes from the entire graph and finds the weight of the vertex connecting them. It then needs to decide if these two nodes are similar or dissimilar. This decision is based on a metric which could be user specified, but by default it is the mean weight of all the vertices in the graph. Calculating this is therefore a prerequisite for running the automata. Let C_{ij} be the weight between node v_i and v_j, z the mean weight in the graph, and p a user specified value. The nodes are then considered similar if $C_{ij} > (1 + p) \cdot z$ and dissimilar if $C_{ij} < (1 - p) \cdot z$. z will be referred to as *Mean_Edge_Cost* is the algorithm.

The goal of the algorithm is to take in a group of nodes V, connected in a graph with edges E, and output K subgroups.

Table 3.2 show the notations used in the algorithm 1, which follows underneath.

Variable	Description
$V = V_1, V_2, ..., V_{KN}$	The nodes which are to be partitioned
K	Number of subsets
E	Edges between the nodes
$(\alpha_1, \alpha_2, ..., \alpha_K)$	Set of actions a node can fall into. Each action belongs in a certain subgroup
$\Phi_1, \Phi_2, ..., \Phi_{KM}$	number of memory states in automata
$\beta 0, 1$	input set, where 0 is reward and 1 is penalty
Q	transition function which moves nodes
G	function which partitions the set of states for the subgroups

Table 3.2: Notations used in the learning automata algorithm.

Algorithm 1: Graph Partitioning Learning Automata (GPLA)

Preprocess:

Compute Mean_Edge_Cost.

Randomly partition V into $\{V_1, V_2, ..., V_K\}$

Assign all nodes to the boundary state of the actions

Data: Set of nodes to be partitioned: $V = \{V_1, V_2, ..., V_{KN}\}$

Result: The final solution to the GPLA

for *Iteration :=1 to Max_Iterations* **do**

  for *a random edge* E_{ij} **do**

    if $C_{ij} > (1 + p) \cdot$ *Mean_Edge_Cost* **then**

      if v_i *and* v_j *are in same subgroup* **then**

        RewardSimilarNodes(i,j)

      else

        PenalizeSimilarNodes(i,j)

    else

    if $C_{ij} < (1 - p) \cdot$ *Mean_Edge_Cost* **then**

      if v_i *and* v_j *are in same subgroup* **then**

        PenalizeDissimilarNodes(i,j)

      else

        RewardDissimilarNodes(i,j)

return final partitions $\{V_1, V_2, ..., V_K\}$

Procedure RewardSimilarNodes(i,j)

Data: Node indices i and j, where ω_i and ω_j are the state indices of similar nodes in the same subgroup.

if $\omega_i \mod M \neq 1$ **then**

  $\omega_i = \omega_i - 1$

if $\omega_j \mod M \neq 1$ **then**

  $\omega_j = \omega_j - 1$

Procedure RewardDissimilarNodes(i,j)

Data: Node indices i and j, where ω_i and ω_j are the state indices of dissimilar nodes in the different subgroups.

if $\omega_i \mod M \neq 1$ **then**

  $\omega_i = \omega_i - 1$

if $\omega_j \mod M \neq 1$ **then**

  $\omega_j = \omega_j - 1$

Procedure PenalizeSimilarNodes(i,j)

Data: Node indices i and j, where ω_i and ω_j are the state indices of similar nodes in the different subgroups.

if $(((w_i \mod M) \neq 0) and ((w_i \mod M) \neq 0))$ **then**
- $\omega_i = \omega_i + 1$
- $\omega_j = \omega_j + 1$ **else**
- **if** $\omega_i \mod M \neq 0$ **then**
  - $\omega_i = \omega_i + 1$
  - temp = ω_j
  - $\omega_j = (\omega_i divM) \cdot M$
  - t = index of a node in v_i subgroup with $v_t \neq v_i$ and v_t closest to the boundary state of ω_i
  - ω_t = temp
  - **else**
  - **if** $\omega_j \mod M \neq 0$ **then**
    - $\omega_j = \omega_j + 1$
    - temp = ω_i
    - $\omega_i = (\omega_j divM) \cdot M$
    - t = index of a node in v_j's subgroup with $v_t \neq v_j$ and v_t closest to the boundary state of ω_j
    - ω_t = temp

Procedure PenalizeDissimilarNodes(i,j)

Data: Node indices i and j where ω_i and ω_j are the state indices of dissimilar nodes in the same subgroup

if $((w_i \mod M) \neq 0) and (w_i \mod M) \neq 0))$ **then**
- $\omega_i = \omega_i + 1$
- $\omega_j = \omega_j + 1$

else

if $\omega_i \mod M \neq 0$ **then**
- $\omega_i = \omega_i + 1$
- TempState1 = ω_j
- Pres_Cost = EvaluateCost of current partitioning **for** *all remaining* $K-1$ *partitions* **do**
  - ω_p = state of node closest to boundary in this current subgroup
  - TempState2 = ω_p
  - $\omega_j = (\omega_p divM + 1) \cdot M$
  - $\omega_p = TempState1$
  - New_Cost = EvaluateCost of current partitioning
  - **if** $New_Cost > Pres_Cost$ **then**
    - $\omega_p = TempState2$
    - $\omega_j = TempState1$
  - **else**
    - Pres_Cost = New_Cost

else

$\omega_j = \omega_j + 1$

TempState1 = ω_i

Pres_Cost = EvaluateCost of current partitioning

for *all remaining* $K-1$ *partitions* **do**
- ω_p = state of node closest to boundary in this current subgroup, α_Z
- TempState2 = ω_p
- $\omega_i = (\omega_p divM + 1) \cdot M$
- ω_p = TempState1
- New_Cost = EvaluateCost of current partitioning
- **if** $New_Cost > Pres_Cost$ **then**
  - $\omega_p = TempState2$
  - $\omega_i = TempState1$
- **else**
  - Pres_Cost = New_Cost

Algorithm procedures

The sub routines "RewardSimilarNodes", "PenalizeSimilarNodes", "PenalizeDissimilarNodes", and "RewardDissimilarNodes" are explained in this sub section. A clarification of each case (all possible circumstances) in each routine is attempted here, for the sake of informing the reader about the inner workings of the algorithm.

Reward Similar Nodes The two compared VMs are in the same subgroup and similar, which means they will be rewarded. They are pushed one step towards the most internal (secure) state. If a node is already at state M, which is the most secure, it is not moved. This could also be the same for both nodes.

Penalize Similar Nodes The nodes are in different subgroups, but should be residing in the same one.

Case 1: This is the simplest case, since they are both in internal states (neither in a boundary state) and moved towards boundary state. If this is not the case, then one or both nodes are at the boundary state of their respective subgroups.

Case 2: Node v_i is in internal state. This means that node v_j is in the boundary state, since Case 1 is not true. v_j can not advance to a more insecure state, and has to jump over to another subgroup - the subgroup where v_i is currently residing. After all, v_i and v_j should be together. v_i is advanced one hop towards the boundary state, and v_j takes the place of a node v_k in the subgroup of v_i which is closest to it's boundary state. v_k is then moved to the old position of v_j.

Case 3: Node v_j is in internal state and v_i is in a boundary state. This case is the same as in case 2, except for the fact that v_i is moved to the subgroup of v_j and v_j is advanced towards the boundary state within it's own subgroup.

Penalize Dissimilar Nodes The nodes are in the same subgroup, but should be in different ones, as they don't belong together because of their dissimilarity.

Case 1: Both the compared nodes v_i and v_j are in internal states and are advanced one step towards the boundary state. One or both of them could reach the boundary state with this movement.

Case 2: One or both of the nodes are in the boundary state of their current subgroup. A node in this state, say v_i is repeatedly moved to all the other subgroups (of which there are $K - 1$) and finally stationed in the subgroup which is most suited, which is the one which causes the minimum amount of inter subgroup cost. One of the nodes in the subgroup which acquired v_i is then moved to the original subgroup of v_i, and it will be the one closest to the boundary state.

Reward Dissimilar Nodes The nodes are in different subgroups, which they should be, as they are dissimilar. Here, the nodes will be moved towards the most internal (secure) state. No movement is done to a node if it currently occupies this state, which means that nothing at all happens if they are both in that state.

Figure 3.2 shows how the nodes "move" within subgroups in the different cases. The cases where a node switch to another subgroup is not illustrated.

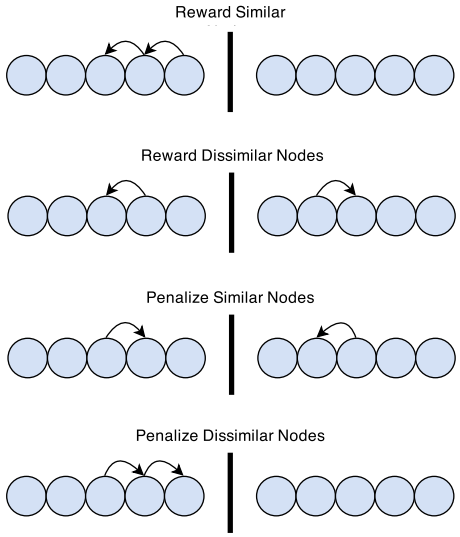

Figure 3.2: Simple cases of GPLA transitions

Once the subgroups are created by the learning automata algorithm, it is known which VMs have a high degree of communication with each other. A second algorithm is needed to optimize the sequence of them. One can assume that the movement of a group with high affinity to VMs residing at the destination (already migrated) would be beneficial for lowering the amount of separated traffic the migration process causes.

Conclusively, the migration system will be based on two algorithms. One is the implementation of the learning automata algorithm created by

Oommen, Croix et al. [40] and the other is developed for this thesis. The latter is found in section 4.3.

3.2 Design of experiment

A matrix will be generated with the dimensions $n \cdot n$, where n is the number of nodes to be migrated. The matrix will be used to model the network traffic between all the VMs, and should have higher values where VMs communicate intensively. As an example, let's consider a scenario with 100 VMs, and that we want to have a lot of traffic running between VMs VM_1 through VM_{10}, VM_{11} through VM_{20}, and so on (ten groups of ten). Within one group, there is a lot of traffic running between the individual VMs. Elsewhere, the communication pattern is scattered randomly, with small values, so that small amounts of inter-subgroup traffic is also accounted for.

The learning automata partitioning scheme should cut the graph into approximately these groups by looking at the values in each position in the matrix. In other words, we have a clear impression of what the "solution" (the subgroups the algorithm outputs), and can easily check if it works in our scenario. We consider the traffic amount on link VM_i, VM_j to be $\frac{(VM_i,VM_j)+(VM_j,VM_i)}{2}$, because the solution will not take the flow direction of traffic into account.

Once a set of VMs are up and running, and have also started transmitting traffic to one another, they will be migrated based on the two algorithms. The metrics that the system should be able to produce values in relation to, are:

- Total migration time
- Total amount of separated traffic

A script for calculating the separated traffic is needed. It should read the a matrix representing the traffic, and also get information about the distribution of VMs in subgroups and the total migration time for each subgroup. Based on this, the script can know where all VMs are (source or destination PM) at any time, and from there calculate the amount of traffic going between VMs currently located in different PMs.

Through the two mentioned metrics, we can tell if the migration scheme is successful in terms of the attributes defined in the problem statement in section 1.1. *Total migration time* corresponds to efficiency and *total amount of seperated traffic* to cost effectiveness.

3.2.1 Bulk migration

Migrations will be done in a sub-grouped fashion, using the pre-copy live migration method. The reason for dividing the group of migration marked VMS into subgroups, is to be sure that the whole process can converge. If one were to instantiate parallel migration of all the VMs, one could in theory achieve a fast migration with no separated traffic, if the migrations finish at the same time. Intuitively, this makes sense in that all VMs would be restarted at the target at the same time and then resume their communications. Inter-VM traffic would in such a case not cross the migration link, and thus not become separated. However, this approach is not possible if one or more of the VMs are generating new memory pages at a higher rate than the amount of available migration bandwidth, at the time that particular VM is migrating. Also, if a subgroup's collective dirty rate is higher than this residual bandwidth, the group can not be migrated with the pre-copy method. Therefore, based on the dirty rate conditions of the VMs, a suitable group size needs to be found.

In figure 3.3, B represents the total bandwidth, S the separated traffic and R the residual bandwidth. From this, we can understand the concept of residual bandwidth and that it changes as S increases or decreases.

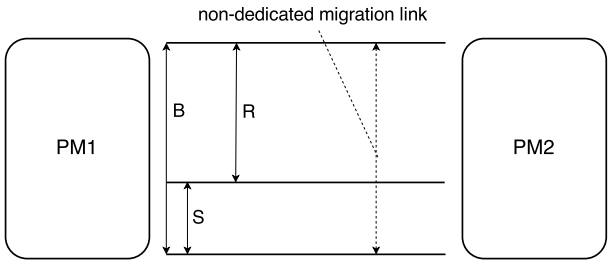

Figure 3.3: Bandwidth notations in a non-dedicated migration link

A valid migration subgroup has to satisfy 3.2.

$$\sum_{VM_i \in subgroup} \{DirtyRate_i\} \leq R \tag{3.2}$$

3.3 Lab setup

We will implement a series of test in a physical virtualization environment. This paravirtualized setup consists of two PMs, each with the following specifications:

Hardware / attribute	Details
Processing	Intel Core 2 Dual Core CPU @ 2.93 GHz
Memory	4 x 2048 MB DIMM @ 1066 MHz
Operating System	Ubuntu 14.04.2 (x64) LTS
Virtualization Solution	Libvirt, version 1.2.2
Storage	2 x 250 GB 7200 RPM hard drives
Networking	Interface 1 (eth0): up to 1000 Mb/s Interface 2 (eth1): up to 1000 Mb/s

Table 3.3: Physical lab hardware specifications

The PMs have the Linux system Ubuntu, which allows for easy accommodation with the KVM virtualization environment. KVM will modify the Linux kernel automatically during installation and add features for hardware management. An altered version of QEMU is used for the actual virtualization. This package is called the KVM/QEMU driver.

3.3.1 Libvirt live migration

Libvirt supports live migrations, and uses the pre-copy method to conduct them. This form of migration copies the entire disk (if any) and memory iteratively, while keeping track of data that has been altered during the migration process. Section 2.2.1 describes this method in detail. Libvirt does not migrate VMs in parallel, but sequentially. However, parallel migrations can be achieved by using *threads* in a program which initiated the migration calls. Libvirt migrations is typically done over a secure channel (SSH) which uses TCP.

3.3.2 Test VMs

The virtual machines which will be orchestrated to create the inter-VM traffic is called "TinyCore" [42]. This is a very small and open-source Linux distribution. It can run with as little as 46 MB RAM, and using a disk with it is storage is optional, which means that it can run directly from memory. One can simply boot it from an image file, and it will run with the image's preconfigured settings and packages installed. Since inspection of live migrations and how they affect the network utilization is the core of this thesis, disk storage is not required. It was therefore chosen to take advantage of TinyCore's disk-less capabilities, and to modify the standard TinyCore-image to make it run a perform some actions each time it boots. When a VM boots, it will:

- Start an SSH-server

- Create a user with root privileges
- Create script files for client and server socket
- Start the server socket and wait for incoming connections
- Install and configure the Python programming language

3.4 Traffic generation

The test environment needs to be able to set some different levels of traffic between the test VMs. This is because we need to distinguish between the sending rates, so that we are able to tell "a lot" from "little" traffic. The learning automata program, which will take a traffic matrix as input, will need these differences in order to perform it's optimizations. Since the iterative comparison of traffic amount between two VMs is based on the mean edge cost amongst all migrating VMs, and a user specified value (see section 3.1.1), no optimization can be expected to be achieved where the traffic pattern is flat in the migration graph. Experimentation on how the physical lab reacts to varying traffic levels are needed to find suitable transmission rates. They should be high enough to actually impact the migration performance.

The transmission and propagation of the traffic requires a CPU friendly and simple sending protocol with little overhead, which is why the User Datagram Protocol (UDP) will be incorporated. The test VMs should be able to be called with instructions on which other VMs to target, and with which traffic levels.

3.4.1 Bandwidth altering

The network link between the two PMs used in the testing can support up to 100 Megabit (mbit) per second transfer speed. That is 12.5 Megabytes (MB) per second, which theoretically means that a 100 MB VM can be transferred between the PMs in 8 seconds. This assumption is disregarding the fact that the sending protocol (TCP) needs to transmit overhead information back and forth to control the transmissions in the migrations. SSH, which is the application level protocol for Libvirt migrations, also has some overhead. This includes Public Key Infrastructure (PKI) related calculations and key transmissions. In practical terms, this means that the actual transmission rate (throughput) of a migrating VM will be lower than 12.5 Megabytes per second.

The implementation of the proposed algorithms, which regulates and schedules the migrations, will likely have a more profound effect if the migration link is more congested. As the residual bandwidth shrinks, each migration will take longer. The expected effect of a slower migration is a greater amount of separated traffic, in addition to a longer total migration time.

3.5 Requirements for solution

Ultimately, a proposed solution should accomplish the following

1. Load the predefined traffic matrix
2. Loop though the matrix
3. Obtain traffic metric between each VM (if any)
4. Instantiate traffic metric (data transmissions) between VMs
5. Migrate the VMs corresponding to the imported matrix to another PM, using the grouping and sequencing from the algorithms

3.6 Revised Approach

The results gathered in the initial testing, called for altercations to certain parts of the test environment. This section contains information about the changes which was done to the initial approach described above. These were necessary in order to get rational data out of the experiments. The individual corrections are outlined in the following sub sections.

3.6.1 Host system load

Initially, the plan was to instantiate network traffic between VMs, and have varying levels of communication in *both* directions between all communicating peers. This meant that for any traffic instantiated pair of VMs, VM_i and VM_j, there would be non-zero values in the traffic matrix in positions i, j as well as j, i. This impacted the load on the VMs and hence the physical machines hosting them quite significantly. When running many VMs, and spawning high levels of traffic in large groups, the processors on the hosts were processing at near maximum capacity.

It was therefore decided to only initiate traffic in one direction for all the communicating peers. By doing it this way, the amount of

connections which the hosts and VMs had to keep track of was reduced considerably.

This was solved by utilizing the concept of triangular matrices. If VM_i and VM_j are selected as communication partners, only one of them would send traffic to the listening server process on the other end. An example of such a matrix is shown in figure 3.4. T is a given traffic value set of VMs.

$$\begin{array}{c} \\ VM_i \end{array} \begin{array}{c} \begin{array}{cccc} & & VM_j & \end{array} \\ \begin{pmatrix} 0 & \mathrm{T} & \mathrm{T} & \mathrm{T} \\ 0 & 0 & T_{ij} & \mathrm{T} \\ 0 & 0 & 0 & \mathrm{T} \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 3.4: Triangular matrix for traffic generation

3.6.2 Dirty rate

It was decided to test migrations with different dirty rates and group sizes, over a dedicated migration link. The reason for this is the following hypothesis. Let L be the expected completion time (latency) for the migration of a subgroup, M the memory size of the VMs, R the residual bandwidth, N the number of VMs in the subgroup and D the collective dirty rate of the subgroup. It is then suspected that L will increase rapidly when N is also increased. Equation 3.3 shows the suspected correlation between the attributes in a group based migration.

$$L = \frac{M}{\frac{R}{N} - D} \tag{3.3}$$

When many dirty rate dominated VMs must share the same residual bandwidth, migrations become very slow as the number of simultaneously migrating VMs increase.

3.7 Simulating migration cost

Preliminary migration testing in the physical lab created some uncertainties related to the different test types which was planned. A simulation program was therefore created, based on the Learning Automata implementation. It can calculate the amounts of separated traffic caused by a migration, by reading the traffic matrix and each subgroup's completion time.

Adjustable parameters in the simulation program include dirty rate, migration link speed and VM memory size. With this feature, it becomes easy to gather information about traffic separation.

The calculation in the script sums of the traffic between all communication between VMs located in different PMs, in both directions. It is based on the cost-formula (equation 2.1) used by the COMMA system [27].

Chapter 4

Results and analysis

This chapter contains the measurements from various experiments derived from the approach. The analyses of the results are present underneath each result. It also describes the physical setup of the test bed used to conduct them.

4.1 Testbed configuration

A physical lab has been set up with the hardware described in section 3.3. Figure 4.1 shows the interconnection between the VMs and the hosts. As we can see, the VMs which are spawned in this habitat are attached to a *virtual bridge*, where a physical interface is connected to provide the migration capabilities. The dotted unidirectional arrows show the possible migration path of a node VM3 from PM1 to PM2. The non-dedicated migration path is also the path that VMs communicate with each other over, if they are located on different PMs.

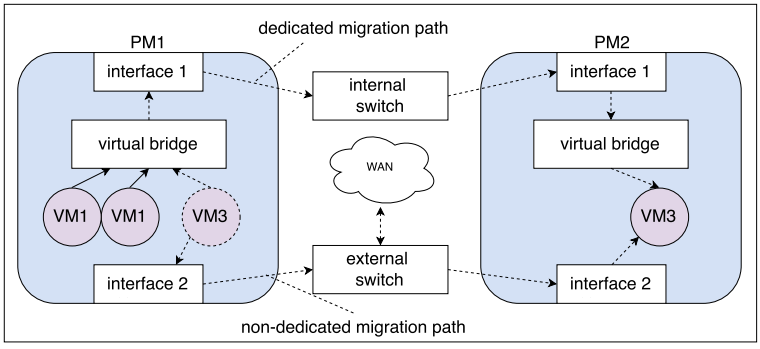

Figure 4.1: Physical lab

4.1.1 Containment of VMs

An organized system for keeping control of multiple VMs had to be created. Since the VMS all boot from the same image and request an IP address during the boot process, the use of an addressing protocol was needed. The environment with the two PMs was therefore set up two identical DHCP server instances (one on each PM) with static host configurations based on the MAC-addresses of the VMs. The environment was set up to be able to create up to a hundred VMs. The script which starts the hosts with Libvirt reads the MAC-addresses and names the hosts so that we can be sure that the host-part of the assigned IP matches the host name. The VMs are spawned on the network 192.168.1.0/24. The first VM gets the IP 192.168.1.101 and the last of the VMs will get 192.168.1.200. The snippet underneath shows that the host "vm25" gets an address ending in 25:

```
1 host vm25 {
2         hardware ethernet 52:54:00:4F:DD:8A;
3         fixed-address 192.168.1.125;
4 }
```

Since the VMs are continuously requesting the local DHCP server for an IP-address, and will do so once migrated to the other host, an identical DHCP instance needs to be running on the destination. Before the lease time of the assigned IP has expired, the DHCP client (the VM) will request to keep it's existing IP, and because the host definitions are the same on both PMs, this IP will be kept. Using this technique, one can be sure of which virtual machine is actually being targeted by the tests defined later.

4.2 VM-to-VM traffic

The program which generates the traffic on the hosts, continuously sends a 1350 Bytes message to the target. This is included the IP and UDP header lengths of the 20 and 8 Bytes, respectively. The raw data sent is therefore a string of 1322 Bytes, since the message is a 104 character long word. This word never changes, so that the clients avoid the processing it takes to create a new one before each datagram is created and sent.

Four different traffic rates have been set up. They are based on different *sleep timers* in the program which sends the data. The code snippet underneath illustrates how this works. The variable *sleep_lenght* is based on input from the script which initiates the traffic on the VMs, which in turn is based on the traffic matrix sent as an input parameter to the script. *Message* is the variable containing the string which is transmitted on the

network by UDP. *Level* is the parameter which indirectly determines the transmission rate, as the loop which sends traffic sleeps a certain amount of time based on it.

```
while True:
    sock.sendto(str.encode(message), server_address)
    sleep(sleep_len)
```

The lowest sleep timer which can be set by the script is 0.05 seconds. This equals a bit rate of 0.216 mbit per second, which is given by $1350 \cdot (1 \div 0.05) * 8/1000000$, or $PacketSize \cdot (1 \div level) * 8 \div 1000000$. This rate will be the fastest any VM can communicate in the test environment, per connection. This is not a particularly fast data rate, compared to today's standards, but each VM will have multiple connections with the others. The most important thing is, in any case, that the test environment can run different traffic levels, and that it is possible to distinguish them. With multiple VMs in a test, the amount of consumed network bandwidth used will quickly become significant. The other three levels are 0.1, 0.072, and 0.054 mbit per second. The different levels in kbit/s are 216, 108, 72 and 54. These traffic levels decrease linearly.

4.2.1 Matrix Usage

Underneath, some example matrices from a 6 node network are shown. The matrix generation script creates these. The asymmetrical matrix is used by the traffic initiation script and, while symmetrical one is fed to the learning automata for partitioning.

Asymmetrical (left) and corresponding symmetrical matrix (right)

```
([[ 0.,  3.,  2.,  1.,  0.,  0.],  |  ([[  0.,  54.,  36.,  27.,   0.,   0.],
  [ 0.,  0.,  0.,  2.,  0.,  0.],  |    [ 54.,   0.,   0.,  36.,   0.,   0.],
  [ 0.,  0.,  0.,  4.,  2.,  0.],  |    [ 36.,   0.,   0., 108.,  36.,   0.],
  [ 0.,  0.,  0.,  0.,  0.,  0.],  |    [ 27.,  36., 108.,   0.,   0.,   0.],
  [ 0.,  0.,  0.,  0.,  0.,  4.],  |    [  0.,   0.,  36.,   0.,   0., 108.],
  [ 0.,  0.,  0.,  0.,  0.,  0.]])  |    [  0.,   0.,   0.,   0., 108.,   0.]])
```

Notice that the matrix on the right is mirrored around one of the diagonals. Position (0,1) (the first "3" in the left matrix) represents traffic level 3, which is 108 kbit/s. Since this is then made symmetrical, it becomes 54 kbit/s, as seen in position (0,1) and (1,0) in the right matrix.

Once a VM is has it's traffic level instantiated for a given test, it will send data at the corresponding rate as long as the client UDP script is running.

4.3 Subgroup scheduling

The first and most important part of the migration scheduling is to apply the minimum cut principle on the network traffic graph using the learning automata algorithm. The idea for using graph partitioning with minimum cut is to decrease the impact the migration will have on inter-VM network traffic. As indicated in section 2.3, the performance of a multi-tiered application will decrease because of the *split components problem*. The solution proposed in this thesis will therefore migrate VMs from one group at a time, in order to keep the traffic running between VMs located at the source- and destination PM minimized, and is hence optimized for minimal amount of traffic between VMs during the migration.

The order of which the groups are migrated can affect the total amount of separated traffic during the migration. This thesis proposes a loop based algorithm which decides which subgroup is to be migrated next at any time, until all subgroups (and hence all VM) are running at the target PM. Coupled with the learning automata algorithm, it should result in an efficient migrations scheme, which produces a low network impact.

Let G be subgroups, S and D be source and destination PM respectively, and T the amount of traffic between subgroups. For any two subgroups G_i and G_j, the exchanged traffic between these groups is the sum of the exchanged traffic between the VMs belonging to these two groups. In formal terms, this is defined as.

$$T(G_i \rightarrow G_j) = \sum_{VM_i \in G_i, VM_j \in G_j} VM_{ij} \tag{4.1}$$

The algorithm 2 migrates groups of VMS based on "affinity", which is a term that has been used to some degree by previous research (i.e. [43], [27]). If a VM has a high degree of affinity to another, it means that they have a lot in common. In our case, this translates to a pair of VMs communicating intensively over the network. The algorithm works in the following way. Initially, the list of already migrated VMs is empty, and all the VMs marked for migration are in the S list. The algorithm then enters a loop where, in each iteration, the subgroup with the highest affinity to the destination PM is migrated. The group is then removed from the list S and appended to list D. This continues until all groups have been migrated, which is until S is empty. Migrating the group with the highest affinity to the destination will intuitively lower the amount of separated traffic. If a group with a high degree of affinity to the destination is not migrated for a long time, then VMs residing in this group will populate the data network link between the source and destination PM, during this period. Consequences of this

Algorithm 2: Affinity algorithm

Data: List of groups to be migrated: $S = \{G_1, G_2, ..., G_N\}$
Result: All VMs from each subgroup is migrated
$D = \emptyset$
while $S \neq \emptyset$ **do**
  for $G_i \in S$ **do**
    $T(G_i \rightarrow D) = \sum_{G_k \in D} T(G_i \rightarrow G_k)$
    $T(G_i \rightarrow S \setminus G_i) = \sum_{G_k \in S \setminus G_i} T(G_i \rightarrow G_k)$
    $\Delta_i = T(G_i \rightarrow D) - T(G_i \rightarrow S \setminus G_i)$
  i_0 = get first elemet of list S
  $i_{max} = i_0$
  for i *in* $S \setminus i_0$ **do**
    if $\Delta_i > \Delta_{max}$ **then**
      $\Delta_{max} = \Delta_i$
      $i_{max} = i$
  $G_{i_{max}} = \arg\max_{G_i} \Delta_i$
  Migrate $G_{i_{max}}$
  $D = D \cup G_{i_{max}}$
  $S = S \setminus G_{i_{max}}$

will be lower residual bandwidth and a larger amount of separated traffic, which are the two metrics the proposed solution seeks to lower.

Algorithm 2 will hereafter be referred to as the "affinity algorithm".

The inter-site traffic amount changes after each group of VMs arrives at the destination. In diagram 4.2, one can imagine that subgroups $\{G3, G4, G5\}$ have been migrated and $\{G1, G2, G6\}$ are still running at the source PM. In this thought scenario, all subgroups G are being transferred live to PM2, which means that either G1, G2 or G6 is next to migrate. The figure show the full mesh nature of possible split traffic relations. In other words, the implementation of algorithm 2 need to know which VMs are located in each groups, to have a complete picture.

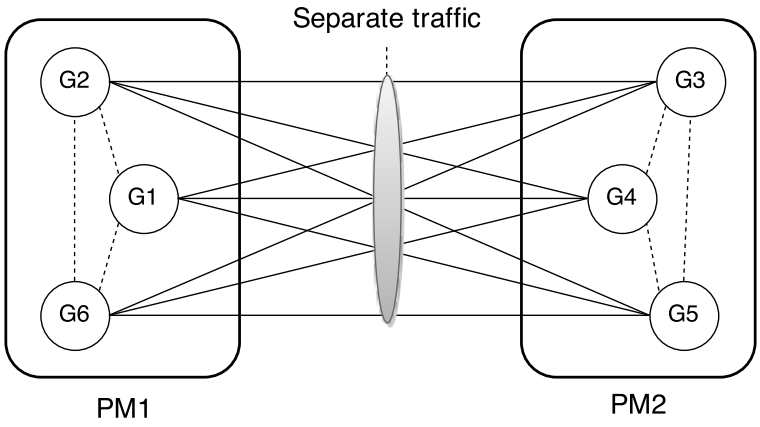

Figure 4.2: Separate inter-site traffic occurring during migrations

4.4 Workflow of testing

The following diagram presents the work flow and architecture behind the testing environment.

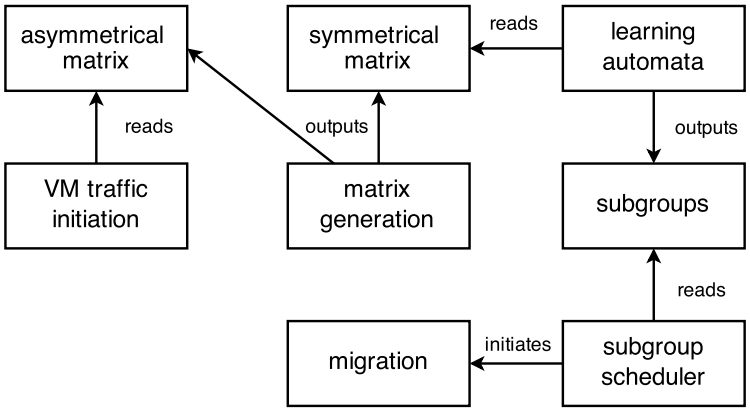

Figure 4.3: Workflow of testing environment

All migration tests will be conducted following this sequence of events:

1. A script generates traffic matrices
2. Traffic is initiated on the running VMs, based on the asymmetrical matrix
3. The learning automata also reads this matrix and outputs subgroups for migration

4. Migration program reads the set of subgroups and decides the order of their migration, and subsequently migrates them based on resulting order

It is important to note that, when testing in this manner, one could simply migrate the subgroups where the matrix generation creates a lot of traffic, and likely get good results. For example, if the matrix script is told to create groups of five, then it will create dense traffic between the first five hosts, and these nodes could simply be grouped for migration. One would only need the learning automata if the distribution of traffic levels is unknown, which would be a more realistic case. It therefore interesting to see how similar the suggested migration subgroups will be to the ones the matrix generation creates.

As the revised approach points out, the test bed could not handle dense traffic in both directions for communicating VMs. It was therefore decided to only let one of any two communicating nodes send traffic. This reduced the overall system load, and enabled simultaneous testing of more nodes than previously.

The following figures graphically reveal how the traffic is distributed when the matrix generator is asked to group 4 and 4 VMs in a 12 by 12 matrix.

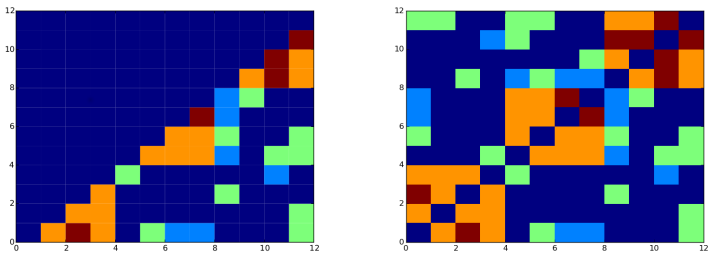

Figure 4.4: Asymmetrical (left) and symmetrical traffic matrices (right)

The blue areas is where there is absence of traffic, such as in the diagonal from the bottom left to the top right corner. The orange and brown areas are where VMs communicate heavily. The asymmetrical figure portrays the pattern of groups most evidently. The symmetrical matrix is needed to get the average traffic between nodes into the learning automata, since it's calculations are based on the average traffic running on a given link between any two VMs.

4.5 Example test

This section will demonstrate the solution using 12 VMs. No measurements will be made here, but the example will serve as an introduction to how the solution can be used on a Linux command line.

The following command will be used to create the VMs from the TinyCore image. Each VM will get 100 MB of RAM. It can be called without the *–thread* option, but using it speeds up the spawning process, as it will make the host receive all the tasks at the same time and schedule processing between them. Running without the option will sequentialize the process, so that the next VM spawning will have to wait for the previous task to complete.

```
python spawn_vms.py -n 12 -r 100 --thread
```

After a while, the VMs are available and can be listed with:

```
virsh list
 Id    Name                           State
----------------------------------------------------
 29    vm1                            running
 27    vm2                            running
 (...)
 35    vm11                           running
 37    vm12                           running
```

The VMs will be referenced by their name, or the number in the name, as the listing of VMs seldom will be sequential. This is because the completion of the installations are dependent on which thread will be completed first.

```
python generate_matrix_general.py -n 12 -g 4 -d
```

The option *-d* dumps binary files on the disk, so that other scripts can read them. After this, the traffic is initiated with:

```
python set_traffic.py -m asym.p
```

The *set_traffic.py* script uses an SSH module to connect to and execute the script *udp_client.py* on the clients, one process per connection to the other hosts. This initiation takes the asymmetrical matrix as input (the file *asym.p*). Next, the learning automata is given the corresponding symmetrical matrix for group scheduling. This program is a Python implementation of algorithms 1 (in section 3.1.1) and 2.

```
python partition.py -a <number of actions in learning
    automata> -n <nodes in each action>
```

This outputs the calculated subgroups to a file, which is later read as input by the migration program. The file looks like this:

```
vm9,vm11,vm12
vm5,vm6,vm7
vm4,vm8,vm10
vm1,vm2,vm3
```

This file is the rank of the migration tasks, where the first line represents the first subgroup to be migrated. This means that VMs 9, 11, and 12 will be the first group to be sent to the destination. The migration itself is started witht the *migrate_vms.py* script, where *-t* is the target host and *-f* is the sequence file which contains the order of the migration groups listed above.

```
python migrate_vms.py -t 192.168.1.2 -f sequence.txt
```

The output of this last command is how long each group took to migrate. Adding these together will give the total migration time, which can be used conclude whether the migration was efficient or not.

4.6 Experiments

There are a number of parameters who's values can be altered which likely will affect either the total migration time or separated traffic amount. Preliminarily, this is thought to include group size, traffic levels, migration link capacity and obviously amount of VMs. If any of these are changed before a test, the changes will be explained in the appropriate parts of this section.

4.6.1 Non-dedicated link

Here, the results from migrations done over a non-dedicated migration link is outlined.

Like in the example test scenario in section 4.5, the same scripts were executed in order to get the decided migration subgroups and the calculated sequence of them.

Experiment 1: Timing parallel migrations

This experiment is done over a non-dedicated link, and the purpose is to see how the effects of running migrations in parallel affects the total migration time. The grouped migration tasks will be the subgroups which

the Learning Automata has produced. It will also be observed how the affinity algorithm's scheduling will affect the migration time.

Attribute	Value
Amount of RAM	200 MB
Number of nodes	16
Group size	4
Highest traffic level	216 kbit/s
Migration link type	Non-dedicated
Migration link speed	1000 mbit/s

Table 4.1: Baseline test configuration

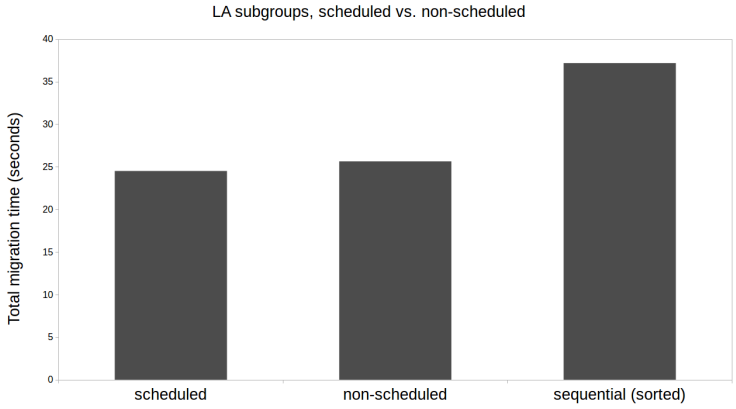

Figure 4.5: Parallel migration performance

Analysis

This results show the effectiveness of migrating multiple VMs in parallel. The *scheduled* column shows the performance of the migration method decided by the Learning Automata (LA) and the affinity algorithm. The *non-scheduled* column shows the performance of the same migration, but with random orders of the subgroups. It is clear the scheduling the LA subgroups with the affinity algorithm is has very little effect on total migration time. It only produces a 4.3% faster migration time, compared to the non-scheduled migrations; 24.5 seconds vs. 25.6 seconds. The reason for this is that there is very little inter-subgroup traffic found in the matrix, and very few groups, which renders the optimization by the affinity scheduling trivial, compared to the graph partitioning.

Lastly, it can be observed that the sequential approach is outperformed greatly by parallel migration. The keyword "sorted", as displayed in the graph, means that migrations were done in the order $VM_1, VM_2, ...VM_N$.

The reason this noteworthy is that VMs are initiated with high levels of traffic sequentially through the VM names. If α is the subgroup size, then the first subgroup would be the VMs in set $\{VM_1, VM_2, ..., VM_\alpha\}$, and the second $\{VM_{\alpha+1}, VM_{\alpha+2}, ..., VM_{2\alpha}\}$. This makes the sequential approach comparable to the grouping mechanism (the Learning Automata), which is supposed to find subgroups similar to these optimal sets. The reason why this is so much slower than the other two approaches is that during sequential migration, much more split traffic occurs, which lowers the residual bandwidth.

Conclusively, this experiment can disclose that if the Affinity algorithm is used, there is very little to save in terms of total migration time, although a slight advantage can be ovserved.

Experiment 2: Amounts of separated traffic

The focus of this experiment is to observe the different amounts of separated traffic which our migration approach generates, versus those of random groups of the same sizes. The attributes of the VMs is kept the same, as well as the group size and number of VMs. Graph 4.6 represents the sums of traffic, based on migration data from 11 tests. The first migration test was for our "calculated subgroup", which is the same migration sequence as "scheduled" in the previous experiment. When this is run multiple times, the level of separate traffic will be identical each time, as the same VMs always will spawn on the destination PM at regular time intervals. The ten other tests are for the random subgroups. The value presented is the average of these tests.

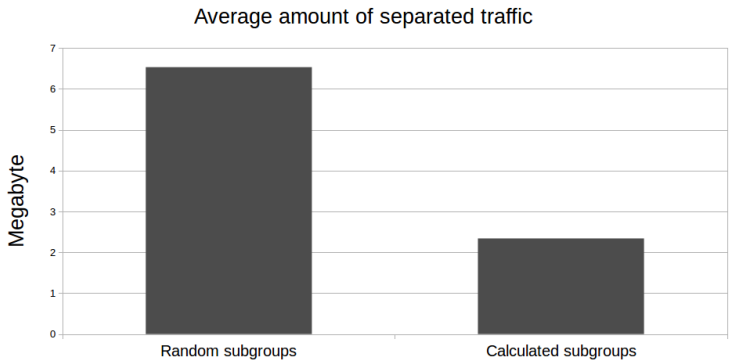

Figure 4.6: Impact of migration random groups

Analysis

This clearly shows that there is a clear benefit, in terms of separated traffic, to cutting a traffic graph with a minimum cut method. The random subgroups generated an average of 6.53 MB (52.24 mbit) of inter-site communication during migration, against 2.34 MB (18.72 mbit) with the approach this thesis proposes. Random grouping creates 179% more traffic. It can be argued that the proposed solution is 64.16% more cost-effective in this scenario, since this is the percentage decrease.

Experiment 3: Speed comparison with random groups

It has been observed that separated traffic can be lowered quite a lot by using the graph partitioning algorithm, and even a bit more by adding scheduling to the groups. Despite this, the problem statement asks for an *efficient* way of performing migrations, meaning that the solution should be fast when migrating multiple talkative VMs. A good way to investigate whether it is efficient or not is to time the migrations of random groups and compare the results with timing of random groups. The VM attributes and subgroup size is still the same. The transmission speed of the migrations link has been set to the lowest speed that the network cards support, 10 mbit/s. Presumably, migration completion times should be different with any link speed, if there are varying levels of split traffic on the link. Yet, the distinctions should be more clear on a slower link.

It was found that the migration times were indistinguishable when the traffic matrix was applied to the 16 VMs. Applying the same traffic matrix two times would double the amount of traffic running between the them. It was only after tripling the traffic amount that there was any notable difference in migration time. Table 4.2 shows the average total migration times from this comparison. The numbers are from 10 experiments each.

	Scheduled LA subgroups	Random subgroups
Total migration time	29.53	30.62

Table 4.2: Comparing migration time with random groups

Analysis

Using the subgroup sequence outputted by the algorithms does actually produce faster migration times, but not to the extent which was hoped for. However, one can assume this small advantage be more significant if the migration link would have less capacity. It this scenario, one can see that the scheduled migration is only 3.55% faster.

4.6.2 Dedicated link

The following experiments have been conducted on a dedicated migration link. This means that any separated inter-VM traffic would not affect the migration, as it would not inhabit the migration link. The purpose of this test is to find the optimal number of VMs to put in each migration subgroup in order to achieve the best possible network throughput and a low amount of separated traffic. If the group size is to large, the sum of the dirty rate in the group could exceed the available migration bandwidth. This is the optimal for reducing the separated VM traffic, but would result in a failed migration, or a so-called "deadlock"[44].

A script has been made to create dirty rate on the VMs. It works in a similar manner as in the script which starts UDP traffic, which is based on a sleep timer. When started, it copies pseudo random bits to a file of a certain size, sleeps for a given time period and repeats. It can be initiated using the following script and syntax:

```
1 python set_dirty.py -s 192.168.1.0 -S <sleep time>
2 -b <block size> -C <block count>
```

The variables *block size* and *block count* can be multiplied together to get the number of bytes which will be written to the file. The *sleep time* parameter is used to calculate the dirty rate per second. The subnet address is given to option *-s*, so the script knows where to look for VMs. In an example where the script is called with block size and block count 100, and a sleep time of 1, 10.000 Bytes would be written in 1 second. From this we can calculate what the dirty rate will be, based on subtracting the time it takes to complete the write operation from this one second, and sleeping for the reminding period.

Experiment 1: Introducing dirty rate

In this experiment, the purpose was to observe the correlation between the subgroup size and migration time, with varying dirty rates. A group of 16 VMs, each with 200 MB RAM, was migrated using parallel migration with two different subgroup sizes, 2 and 4, and four different dirty rate levels. The results are shown in figure 4.7.

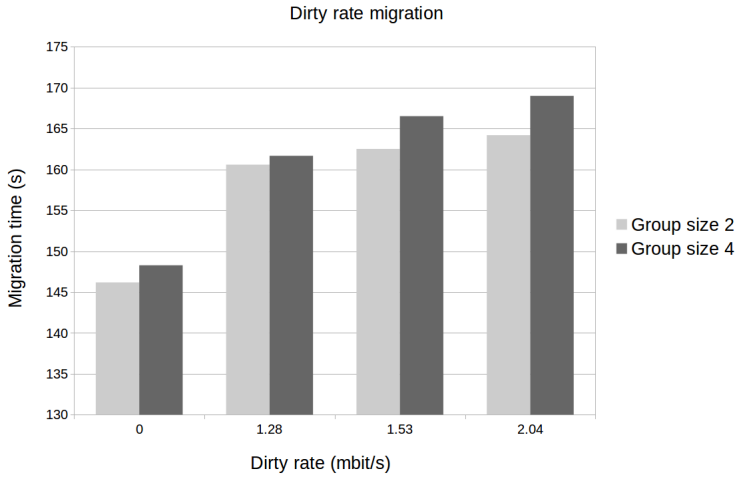

Figure 4.7: The effects which increasing dirty rates has on migration time

These VMs were also sending traffic between each other, in addition to dirty rate. The same matrix file as in the previous experiment was applied once. The following amounts of separated traffic has been calculated from the matrix, and are based on estimations on when each subgroup was finished migrating.

Dirty rate (mbit/s)	Subgroup size 2	Group size 4
0	135626.4	84556.8
1.28	149040.0	92851.2
1.53	149040.0	96076.8
2.04	151275.6	96307.2

Table 4.3: Separated traffic during different dirty rates

Analysis

Graph 4.7 could indicate that a higher amount of simultaneously migrating VMs results in a longer migration time, and that the time difference is more significant on higher dirty rates. Since the number of running VMs was 16, only factors of 16 could be used as group sizes, such that all VMs would reside in equal sized subgroup. The same migration tests were therefore also attempted with subgroup size 8 (2 groups). None of the migration involving any of the previously tested dirty rate resulted in successful migrations when the subgroup size was this big. In other words, these migrations resulted in a deadlock situation. However, when migrating groups of 8 VMs without any dirty rate, the whole process completed within 140 seconds, which is 15.8% faster than subgroup size 2 with no

dirty rate. The effectiveness of threading migration of many VMs at the same time is therefore quickly suppressed when dirty rate is introduced as a factor.

When it comes to separated traffic, the best result is obviously subgroup size 4 with zero dirty rate. This size is nonetheless preferable with all the presented dirty rates. With migrations with subgroup size 8 not converging on any dirty rate, 4 is the optimal subgroup size for minimizing traffic separation, and therefore the impact any running multi-tiered application hosted by the VMs. However, it takes longer to complete compared to migrating two and two.

4.6.3 Simulation results

Since it is time consuming and impractical to manually perform physical tests, a simulation script was used to facilitate all the variables needed to run virtual migrations, and observe effects related to separated traffic. It takes dirty rate, available migration bandwidth and VM memory size as input.

The simulation can yield credible results when it comes to random subgroups, as it can create as many of them as needed. Results from a small number of random subgroups could be polluted by the optimal grouping being chosen by random. The number of permutations of few groups can be low, making it probable that this sequence will be picked.

For all these simulations, a similar traffic matrix has been created, of 16 by 16 nodes, with high traffic levels between groups of four. The simulation will demonstrate the effects of adjusting the parameters dirty rate and group size. In order to be able to compare these results with the actual physical migration, the memory size of each VM is set to 1600 mbit (200 MB) and the migration link to 100 mbit/s, as in the real test scenarios. The simulations have gathered data for three cases:

- No dirty rate
- Dirt rate of 5 mbit per VM
- Dirt rate of 10 mbit per VM

All dirty rate cases are tested with subgroup sizes 2, 4 and 8.

Abbreviations used in the graphs are:

Abbreviation	Explanation
LA	The Learning Automata has been used to decide the subgroups
AF	The affinity based algorithm has been used to decide subgroup migration order
RG	Random subgroups have been used

Simulation 1: no dirty rate

This first simulation is used as a baseline to only identify the effects of changing the group size. Figure 4.8 can reveal that the optimal subgroup size for achieving low amounts of separate traffic is 4. It also show that the best performance, by the same metric, is achieved by a combination of the two implemented algorithms (LA+AF). When the subgroup size is 8, there are only two of them, and hence random grouping would result in the same amount of separate traffic as in the sequenced method. This column has therefore been omitted. This is explained by the fact that the separated traffic is the sum of traffic running between the migrated subgroups. If there are only two, then this traffic occurs after one of them has been migrated, and is absent when the last is migrated. Therefore it will be the same in both cases.

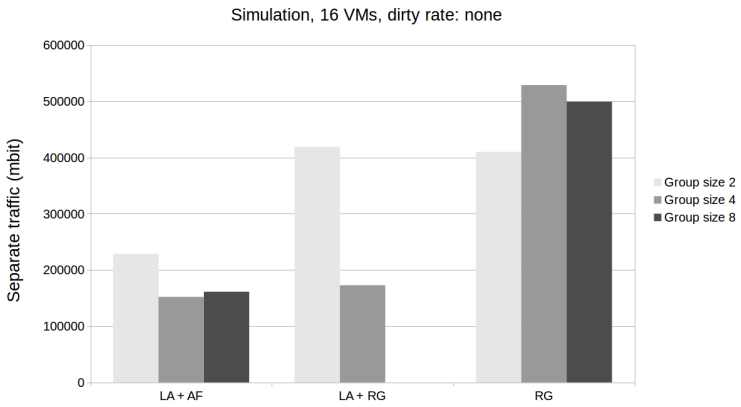

Figure 4.8: Migration simulation without any dirty rate

Analysis

One can notice that the is an advantage to using the LA in combination with the affinity algorithm. When both are used to schedule the migrations (the three leftmost columns), the optimal group size is 4. Sizes 4 and 8 are comparable, but it can be concluded that a grouping of 2 is two small as it leads to more traffic separation. Also, when migrating only two nodes at a

time *without* applying the affinity scheduling (hence random grouping), the separated traffic is more than doubled, meaning that the affinity algorithm can double the cost effectiveness in this case. Lastly, it can be observed that random grouping is substantially more ineffective, when the optimal subgroup size is applied.

Simulation 2: with dirty rates

Figure 4.9 shows the results of setting a consistent 5 mbit/s dirty rate on the VMs in the simulation. Still, the proposed solution gives the best result with all the different subgroup sizes.

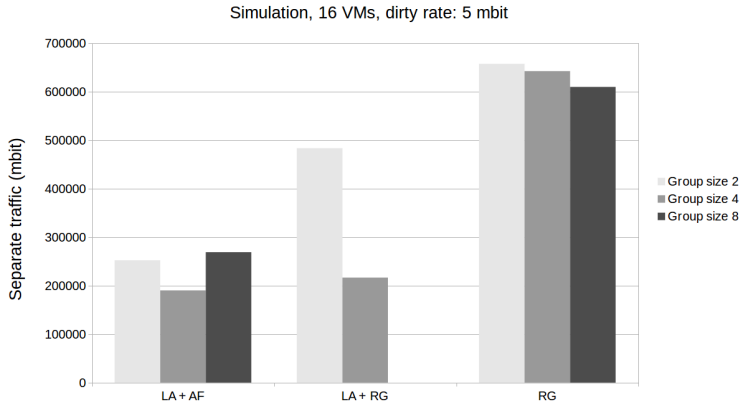

Figure 4.9: The effects of adding constantly changing memory to the VMs

Figure 4.10 shows what happens when the dirty rate is doubled, to 10 mbit/s.

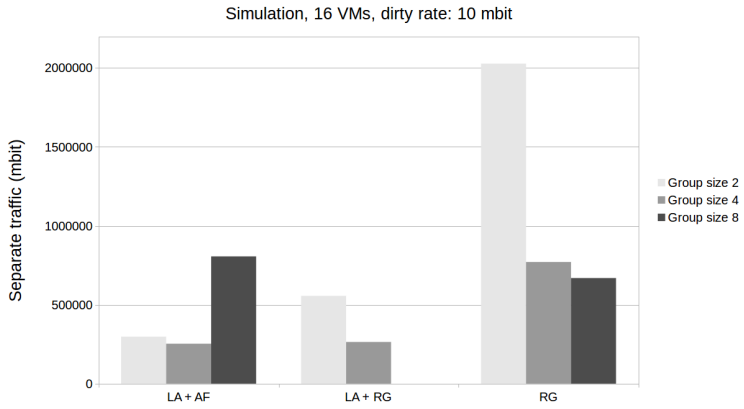

Figure 4.10: Increasing the dirty rate

Analysis

When comparing the two previous graphs, 4.9 and 4.10, with the non-dirty rate scenario, it becomes clear the optimal group size is 4, when both algorithms are used. Focusing on this size, we also see that it is only a marginally favorable to include the affinity algorithm. The LA+AF reduces cost by 4%, with the measurements being 253440 mbit for LA+AF and 264192 with random grouping.

When random groups of two VMs are migrated with the highest dirty rate, a massive amount of separated traffic occurs, which can be seen in figure 4.10. This is because separation happens very fast after the migration starts. It takes very little time to move the two first VMs to the destination, and it exists from then on, until the migration completes. When 8 VMs are beginning to migrate, all VMs still run at the source PM, and traffic separation does not occur until the first group is migrated. Thus, traffic separation has less time to occur.

4.6.4 Affinity algorithm

In cases where VMs have high dirty rates, parallel migrations could be rendered impossible, as since grouping is out of the question due to violation of the following constraint.

$$\sum_{VM_i \in subgroup} DirtyRate_i \leq Bandwidth \tag{4.2}$$

Such a scenario could occur when planning migrations between data centers located in different geographical areas, where the migration link is significantly slower than local management links.

So far, only the LA algorithm has been tested with or without the Affinity algorithm for deciding sequence. The simulation script was altered so that the affinity method, which is an implementation of the pseudo code in algorithm 2, could be used in the simulation program by itself. It would be interesting to examine the performance of this algorithm in isolation. In this experiment, a list of migration marked VMs is fed to the affinity algorithm, which should output the recommended sequential migration sequence.

The traffic matrix the calculation is based on is the same as in the previous experiment. The following is the decided sequence, which should result in a modest amount of separated traffic.

```
1 ('vm14', 'vm15', 'vm13', 'vm16', 'vm9', 'vm10','vm11', 'vm1
    ', 'vm12', 'vm7', 'vm6', 'vm5', 'vm3', 'vm4', 'vm2', '
    vm8')
```

Figure 4.11 shows that the proposed solution will produce less separate traffic during a sequential migration. The right column represents the average of 30 random sequences.

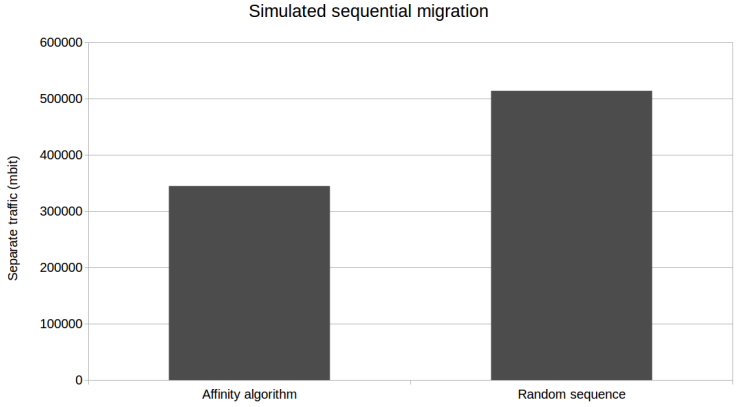

Figure 4.11: Less separate traffic when using the affinity algorithm

When comparing the calculated separated traffic amounts for a thousand random sequences, and removing any duplicate values, we are left with 462 unique values. The affinity algorithm is among the best results from this set. Only three sequences produces less separated traffic, which means that only 0.64% were better. It can therefore be said that the affinity algorithm improves sequencing for over 99% of the cases, compared to choosing a sequence at random.

The following histogram shows the distribution of these samples.

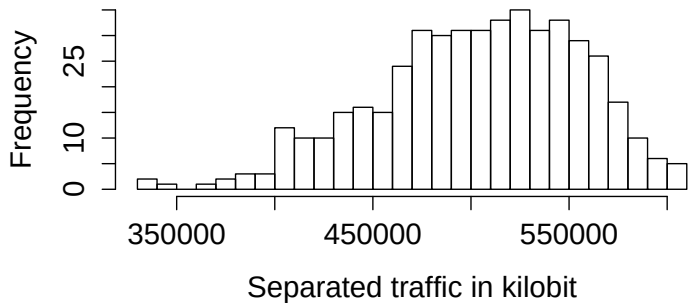

Figure 4.12: Histogram showing separate traffic amounts for random sequences

Analysis

It is noticeable from the output sequence above that high levels of traffic have been initiated in groups and based on VM names. For example high traffic between VMs *vm1* to *vm4*. Intuitively, the VMs belonging to a given subgroup should be listed after one another in some permutation. Nevertheless, a few VMs are displaced, outside their respective subgroups, namely *vm1* and *vm8*. This might be caused by the low level traffic which is present in about 30% elsewhere in the traffic matrix. The result shows that while using the Affinity algorithm alone, low separate traffic can still be achieved. In this experiment, the reduction was 32.9%.

Sequential migration time

After observing the affinity algorithm's positive effect with sequential migration, it was thought that the total migration time for a group of VMs would be somewhat lower when this sequence is used, compared to random sequences. The reason being that less separated traffic would leave a greater amount of available migration bandwidth. The following table shows the resulting migration times (in seconds) after applying different levels of traffic between the VMs. The traffic matrix was applied to the VMs up to four times. The experiment was done over a non-dedicated link, so that the split VM traffic would impact the completion time. Intuitively, the gap between the results should widen the more times the traffic matrix is applied. However, very similar values were observed on these different stages.

	Times applied traffic matrix			
Scheduling	**1**	**2**	**3**	**4**
Using Affinity algorithm	36.51	39.08	41.87	43.76
Random	36.59	39.05	41.94	43.81

Table 4.4: Similar migration times using only the affinity algorithm

Analysis

This results of this experiment shows that migration time is not significantly lowered when applying only the affinity algorithm to the migration plan. This is somewhat surprising. Logically, there should be more separated traffic occupying the non-dedicated migration link when a random sequence is chosen. During this testing, it was discovered that the processor utilizations on the PMs were nearing the maximum limit when the

quadruple of the original matrix was applied. Applying it any more times could possibly impair them, and hence the traffic application only went up to four times.

Chapter 5

Discussion and conclusion

It was discovered, during the course of this thesis, that optimizing live migrations is a popular computer research field which has lead to lots of interesting contributions and discoveries. In this chapter, the various aspects of the thesis will be reflected upon.

The main idea in this project was primarily that applying a minimum cut algorithm to a traffic graph and migrating the resulting groups of VMs, would be beneficial for lowering migration time. By grouping all the talkative VMs and migrating them in parallel, one will reduce the amount of traffic running between groups. Following this principle, one can be sure that less VM-traffic would "pollute" a migration link. This initial idea was what started the research and lead to the application of the learning automata (LA) algorithm. The paper it was published in claimed it was both accurate and efficient, and was therefore chosen for this assignment. As other proposed live migration systems have already used minimum-cut methods for dividing networks, such as Clique Migration [37], another algorithm was developed with the hopes that further optimization could be achieved.

5.1 Evaluation

Results from the proposed solution in this thesis have disclosed some interesting findings. The most noteworthy is that the LA partitioning reduces the amount of separated traffic to a substantial degree. The contribution of this thesis, on the other hand, is the affinity algorithm. When testing this without the use of LA partitioned subgroups, the separated traffic caused by the migration was lowered substantially, compared to random groupings. Results in section 4.6.4 show that optimization is achieved in over 99% of the cases with sequential migration.

This algorithm is therefore effective for lessening inter-subgroup traffic. However, the experiments applying only the affinity algorithm are only done in simulated environments, and should also be tested in a real testbed. Measuring the actual separated traffic occurring during migration is the only way to be completely sure if any optimization is actually happening. This could, for example, have been done by setting up a firewall, such as *iptables*, between the PMs and configure rules to match inter-site traffic. One could then count the number of bytes matched by the rules after the migration has completed.

The time difference between migrating LA-suggested subgroups and random subgroups is real, although not compelling. It is feasible to believe that further lowering the migration link's capacity would lead to a greater difference in migration completion time, because a larger portion of the total bandwidth would be used by split traffic.

As discovered, memory dirty rate can stall migrations due to the nature of the iterative page fetching from destination to source. The goal of the tests involving dirty rate, was to explore how the different subgroup sizes were affected by it, in terms of separated traffic and migration time. Cost-effective migrations is defined in the approach as network-friendly migrations. Accordingly, dirty rate was not was not that important for the experiments.

5.1.1 Problem statement

The following was the problem statement which this thesis set out to solve.

"How can we achieve cost-effective and efficient live migrations of virtual machines in a cloud environment?"

All in all, by implementing a combination of minimum cut graph partitioning and affinity aware scheduling, it was possible to reduce the amount of separated traffic. Test case 4.6.1 resulted in a 64% improvement over random parallel migrations. By this, we can say that *cost-effective* migrations can be performed, and thus this part of the problem statement is answered. On the other hand, the performance of the solution has not been measured against any existing migration systems, so it can obviously not be concluded that it is better other solutions.

When it comes to the *efficiency*, we can only conclude that the algorithms used results in marginally faster migrations, compared to random parallel migrations of equal sized subgroups. Since the migrations were only tested on a high capacity link, and with few VMs, only a 3.5% time reduction

was found, as shown in section 4.6.1. Again, no other existing works were compared against, which means the current solution is only a little faster than choosing random sequences.

The most prominent result is how the algorithms can reduce the amount of separated traffic. The algorithm introduced in this thesis helps migrations to accomplish more cost-effective relocations of virtual machines.

5.2 Future work

Using the physical lab had it's challenges, even though it provided the necessary platform needed to run migrations. It did not, however, allow much more than 20 simultaneously running VMs on one PM, when they all had traffic running between them. The main issue with the test bed was processing power. The CPU usage neared the maximum of 200% (2 cores) when running 20 such VMs, and therefore the most used amount was 16.

The first goal of forthcoming work is to upgrade the physical test bed to more capable and robust hardware, capable of running hundreds of memory- and network-intensive VMs. Subsequently, large scale migrations can be conducted in order to see if the algorithms produce efficient and cost-effective migrations in such cases. Migrating larger subgroups should also be tested. Since the affinity algorithm optimizes scheduling based on the inter-subgroup traffic, it would be interesting to see how it performs in a large scale environment, where more such traffic would be present. By increasing the amounts of VMs in each subgroup, the probability for more inter-subgroup traffic increases. It is suspected that the affinity algorithm can perform better, in terms of cost-reduction, in environments with high degrees of inter-subgroup traffic.

5.2.1 Traffic patterns

The traffic generation script, which outputs the matrix the traffic initiation is based on, was made because a clear traffic pattern was needed. An alternative approach would be to acquire some real traffic data, and create a matrix from this. This would likely not produce a matrix where groups containing the same amounts of nodes are communicating intensely, as in a constructed case. The most flexible alternative approach would perhaps be to create a program which can scan network activity automatically, and output a traffic graph based on average values. But if this is incorporated, the time spent analyzing the network would become a factor in the overall completion time for the migrations, and would have to be taken

into consideration. Using a limited time network analysis, it could be interesting to see if a longer time produces a more credible pattern, and if a more "correct" migration decision is made because of it.

The proposed solution has been tested with the pre-copy live migration method on the Libvirt virtualization platform. It is encouraged to try other platforms, to verify that migration performance is enhanced on those, as well as in Libvirt.

5.2.2 Latency

This thesis did not focus on maintaining low response times of VMs, which is a major goal for cloud computing companies today. Demanding services like online gaming and video streaming are particularly vulnerable if server response times go up. Latency optimization should be included as a metric in the future.

If the system were to accommodate this functionality, one could test response times on all migrating VMS, by pinging (gathering timed responses) all VMs during a migration, and compare the results to the current scheduling system proposed in this thesis. One could also compare response times of VMs located in the same migration subgroup with those of distinct subgroups. Intuitively, there would be less performance delay on the VMs migrated together, but this hypothesis would have to be tested.

5.2.3 More sophisticated scheduling

In order to make the migration planning more refined, one could create features for timing migrations, so that all VMs in a given subgroup are resumed at the destination at the same time. This is where bin packing could be used to "pack" migration tasks (individual VMs) on a migration link. The COMMA system has support for this. The idea is that different sized VMs need different bandwidth amounts to complete migration within a given time. As an example, when COMMA (see section 2.4) migrates two VMs together, they are started at different times and spawn simultaneously at the destination. Doing so enables maximum network utilizations, while eliminating split traffic between the VMs. This approach is very attractive and will be included in future work.

5.2.4 Conclusion

This thesis conveys a proof-of-concept like approach to tackling multiple-VM live migrations, using a combination of two algorithms. Through the proposed solution, we were able to observe the cost-reducing effects when performing migration experiments, as well as a slight shortening of total migration time.

Using both real and simulated VM migrations, the memory- and network related properties were examined and understood, and the main obstacle was defined as a "split components problem". The algorithms were tested on a KVM based virtualization using Libvirt as the management tool.

The thesis' main contribution is the affinity algorithm, which iteratively migrates VMs with strong connections to the target host and little affiliation to VMs at the source. Experiments show that, on average, more than 30% of this traffic can be eliminated by using it.

Bibliography

[1] Oren Laadan and Jason Nieh. 'Operating system virtualization: practice and experience'. In: *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. ACM. 2010, p. 17.

[2] Charles David Graziano. 'A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project'. In: (2011).

[3] Liang Liu et al. 'GreenCloud: a new architecture for green data center'. In: *Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session*. ACM. 2009, pp. 29–38.

[4] Ching-Huang Lin et al. 'Resource allocation in cloud virtual machines based on empirical service traces'. In: *International Journal of Communication Systems* 27.12 (2014), pp. 4210–4225. ISSN: 1099-1131. DOI: 10.1002/dac.2607. URL: http://dx.doi.org/10.1002/dac.2607.

[5] Timothy Wood et al. 'Black-box and Gray-box Strategies for Virtual Machine Migration.' In: *NSDI*. Vol. 7. 2007, pp. 17–17.

[6] Wentao Liu. 'Research on cloud computing security problem and strategy'. In: *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*. IEEE. 2012, pp. 1216–1219.

[7] William Voorsluys et al. 'Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation'. English. In: *Cloud Computing*. Ed. by MartinGilje Jaatun, Gansen Zhao and Chunming Rong. Vol. 5931. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 254–265. ISBN: 978-3-642-10664-4. DOI: 10.1007/978-3-642-10665-1_23. URL: http://dx.doi.org/10.1007/978-3-642-10665-1_23.

[8] Masamitsu Honjo, Atsushi Kubota and Toshiaki Kitamura. 'Parallel programming framework for heterogeneous computing environment with Xen virtualization'. In: *TENCON 2010-2010 IEEE Region 10 Conference*. IEEE. 2010, pp. 1100–1105.

[9] Christopher Clark et al. 'Live migration of virtual machines'. In: *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association. 2005, pp. 273–286.

[10] Wenjin Hu et al. 'A quantitative study of virtual machine live migration'. In: *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. ACM. 2013, p. 11.

[11] Hui Lu et al. 'vHaul: Towards Optimal Scheduling of Live Multi-VM Migration for Multi-tier Applications'. In: (2015).

[12] Kejiang Ye et al. 'Live migration of multiple virtual machines with resource reservation in cloud computing environments'. In: *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE. 2011, pp. 267–274.

[13] Yi Zhao and Wenlong Huang. 'Adaptive distributed load balancing algorithm based on live migration of virtual machines in cloud'. In: *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*. IEEE. 2009, pp. 170–175.

[14] Kejiang Ye et al. 'Two optimization mechanisms to improve the isolation property of server consolidation in virtualized multi-core server'. In: *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*. IEEE. 2010, pp. 281–288.

[15] Fereydoun Farrahi Moghaddam, Mohamed Cheriet and Kim Khoa Nguyen. 'Low carbon virtual private clouds'. In: *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE. 2011, pp. 259–266.

[16] Kasidit Chanchio and Phithak Thaenkaew. 'Time-bound, thread-based live migration of virtual machines'. In: *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE. 2014, pp. 364–373.

[17] Vmware. *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*. Feb. 2015. URL: http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf.

[18] Wei Chen et al. 'A novel hardware assisted full virtualization technique'. In: *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*. IEEE. 2008, pp. 1292–1297.

[19] Tim Abels, Puneet Dhawan and Chandrasekaran Balasubramanian. *An Overview of Xen Virtualization*. May 2015. URL: http://courses.cs.vt.edu/~cs5204/fall07-kafura/Papers/Virtualization/Xen-ShortOverview.pdf.

[20] Karen Scarfone. *Guide to security for full virtualization technologies*. DIANE Publishing, 2011.

[21] Yuan-Cheng Lee, Chih-Wen Hsueh and Rong-Guey Chang. 'Inline emulation for paravirtualization environment on embedded systems'. In: *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*. Vol. 1. IEEE. 2011, pp. 388–392.

[22] Jyotiprakash Sahoo, Subasish Mohapatra and Radha Lath. 'Virtualization: A survey on concepts, taxonomy and associated security issues'. In: *Computer and Network Technology (ICCNT), 2010 Second International Conference on*. IEEE. 2010, pp. 222–226.

[23] Lucas Nussbaum et al. 'Linux-based virtualization for HPC clusters'. In: *Montreal Linux Symposium*. 2009.

[24] Juan Carlos Chaves. 'Enabling high productivity computing through virtualization'. In: *DoD HPCMP Users Group Conference, 2008. DOD HPCMP UGC*. IEEE. 2008, pp. 403–408.

[25] Open Virtualization Alliance (OVA). *Kvm: The Rise Of Open Enterprise-Class Virtualization*. Feb. 2015. URL: https://software.intel.com/sites/default/files/OVM_KVM_wp_Final7.pdf.

[26] Wikipedia. *Hypervisor*. Apr. 2015. URL: https://en.wikipedia.org/wiki/Hypervisor.

[27] Jie Zheng et al. 'COMMA: Coordinating the Migration of Multi-tier Applications'. In: *SIGPLAN Not.* 49.7 (Mar. 2014), pp. 153–164. ISSN: 0362-1340. DOI: 10.1145/2674025.2576200. URL: http://doi.acm.org/10.1145/2674025.2576200.

[28] Avi Kivity et al. 'kvm: the Linux virtual machine monitor'. In: *Proceedings of the Linux Symposium*. Vol. 1. 2007, pp. 225–230.

[29] Robert Bradford et al. 'Live wide-area migration of virtual machines including local persistent state'. In: *Proceedings of the 3rd international conference on Virtual execution environments*. ACM. 2007, pp. 169–179.

[30] Timothy Wood et al. 'CloudNet: dynamic pooling of cloud resources by live WAN migration of virtual machines'. In: *ACM SIGPLAN Notices*. Vol. 46. 7. ACM. 2011, pp. 121–132.

[31] Fabio Checconi, Tommaso Cucinotta and Manuel Stein. 'Real-time issues in live migration of virtual machines'. In: *Euro-Par 2009–Parallel Processing Workshops*. Springer. 2010, pp. 454–466.

[32] Peng Lu, Antonio Barbalace and Binoy Ravindran. 'HSG-LM: hybrid-copy speculative guest OS live migration without hypervisor'. In: *Proceedings of the 6th International Systems and Storage Conference*. ACM. 2013, p. 2.

[33] Michael R Hines, Umesh Deshpande and Kartik Gopalan. 'Post-copy live migration of virtual machines'. In: *ACM SIGOPS operating systems review* 43.3 (2009), pp. 14–26.

[34] Adnan Ashraf et al. 'Feedback control algorithms to deploy and scale multiple web applications per virtual machine'. In: *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. IEEE. 2012, pp. 431–438.

[35] M.F. Bari et al. 'CQNCR: Optimal VM migration planning in cloud data centers'. In: *Networking Conference, 2014 IFIP*. June 2014, pp. 1–9. DOI: 10.1109/IFIPNetworking.2014.6857120.

[36] Haikun Liu and Bingsheng He. 'VMbuddies: coordinating live migration of multi-tier applications in cloud environments'. In: (2013).

[37] Tao Lu et al. 'Clique Migration: Affinity Grouping of Virtual Machines for Inter-cloud Live Migration'. In: *Networking, Architecture, and Storage (NAS), 2014 9th IEEE International Conference on*. Aug. 2014, pp. 216–225. DOI: 10.1109/NAS.2014.40.

[38] Vijay Mann et al. 'Remedy: Network-aware steady state VM management for data centers'. In: *NETWORKING 2012*. Springer, 2012, pp. 190–204.

[39] Marcela Quiroz-Castellanos et al. 'A grouping genetic algorithm with controlled gene transmission for the bin packing problem'. In: *Computers & Operations Research* 55 (2015), pp. 52–64.

[40] B. John Oommen, De St Croix et al. 'Graph partitioning using learning automata'. In: *Computers, IEEE Transactions on* 45.2 (1996), pp. 195–208.

[41] Kumpati S Narendra and MLAA Thathachar. 'Learning automata-a survey'. In: *Systems, Man and Cybernetics, IEEE Transactions on* 4 (1974), pp. 323–334.

[42] Team Tiny Core. *Welcome to The Core Project - Tiny Core Linux*. Apr. 2015. URL: http://distro.ibiblio.org/tinycorelinux/.

[43] Jianhai Chen et al. 'Aaga: Affinity-aware grouping for allocation of virtual machines'. In: *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*. IEEE. 2013, pp. 235–242.

[44] Tusher Kumer Sarker and Maolin Tang. 'Performance-driven live migration of multiple virtual machines in datacenters'. In: *Proceedings of the 2013 IEEE International Conference on Granular Computing (GrC)*. IEEE. 2013, pp. 253–258.

Chapter 6

Appendix

VM migration script

```
1
2  #!/usr/bin/python
3
4  import pprint
5  import os
6  import re
7  import time
8  import sys
9  from threading import Thread
10 import argparse
11 import subprocess
12 import math
13
14 def read_batch(filename):
15     """ Gets the structure for vm migration
16     filename: name of the file
17     """
18
19     vms = {}
20     with open(filename, 'r') as f:
21         for id, line in enumerate(f.readlines()):
22             l = line.strip().split(',')
23             vms[id] = l
24     return vms
25
26 def read_vm_names(filename):
27     vms = []
28     with open(filename, 'r') as f:
29         for line in f.readlines():
30             vms.append(line)
31     return vms
32
```

```
33
34 def migrate_batch(vms,target):
35     """ Migrates selected vms to desired host
36     vms: [vms]
37     target: ip of desired host
38     """
39     threads = []
40     for vm in vms:
41         thread = Thread(target=migrate_vm, args=[vm, target
               ])
42         thread.start()
43         threads.append(thread)
44
45     for thread in threads:
46         thread.join()
47
48 def migrate_vm(vm, target,single=False):
49     """ Migrates a vm from a host to specific target
50     host: ip of host
51     vm: specific vm
52     target: ip of target
53     """
54     cmd = "virsh migrate --live %s qemu+ssh://%s/system" %
          (vm, target)
55     if single:
56         output = subprocess.check_call(cmd, shell=True)
57         if output:
58             print "single vm done: %s" % time.time()
59         else:
60             print "single vm done: %s" % time.time()
61
62     else:
63         migrate = os.popen(cmd)
64
65 def run_the_batches(batch, target):
66     timedone = {}
67     for batch, vms in batch.iteritems():
68         curr_time = time.time()
69         migrate_batch(vms, target)
70         timedone[batch] = (time.time()-curr_time)
71
72     return timedone
73
74 def main(args):
75     print "start time,%s" % time.time()
76     if args.single:
77         migrate_vm(args.single,args.target,True)
78     else:
79         batch = read_batch(args.filename)
```

```
80
81      timedone = run_the_batches(batch, args.target)
82
83      with open("seq_completion_time_groups.txt", "w+") as f:
84
85          vm_names = read_vm_names(args.filename)
86          for batch, times in timedone.iteritems():
87              timestring = str(times)
88              vm_numbers = []
89              print vm_names[batch] # gives: vm13,vm14,vm15,
                    vm16
90              for item in vm_names[batch].split(','):
91                  vm_numbers.append(str(int(item[2:])-1))
92              print vm_numbers
93              log_string = "%s,%s\n" % (timestring, ",".join(
                    vm_numbers))
94              f.write(log_string)
95
96
97  if __name__ == '__main__':
98      parser = argparse.ArgumentParser(
99          description='threaded script to allow concurrent /
                parallel migrations (libvirt)')
100     parser.add_argument(
101         '-f', '--filename', type=str, help='Filename of
                batchfile', required=False)
102     parser.add_argument(
103         '-t', '--target', type=str, help='Target Host IP',
                required=True)
104     parser.add_argument('-s', '--single', type=str, help='
            migrate a single vm: name')
105     args = parser.parse_args()
106
107     main(args)
```

Traffic matrix generator

```
1
2   #!/usr/bin/env python
3   # -*- coding: utf-8 -*-
4
5   import random
6   import pprint
7   import numpy
8   import argparse
9   import sys
10  import pickle
11
12  LOW_LEVELS = [1,2]
```

```
13 HIGH_LEVELS = [3,4]
14
15 def dump_matrix(filename, matrix):
16     pickle.dump(matrix, open(filename, "wb"))
17
18
19 def set_dense_traffic(matrix, arms, groupsize, nodes):
20     for counter in range(1,(arms+1)):
21         d = (groupsize*counter)
22         for i in range((d-groupsize),(d+1)):
23             for j in range((d-groupsize),(d+groupsize)):
24                 if ((i/d) == 0 and (j/d) == 0 and i!=j and
                       j>i):
25                     matrix[i][j] = random.choice(
                           HIGH_LEVELS)
26     return matrix
27
28 def set_sparse_traffic(matrix, nodes):
29     for i in range(0,nodes):
30         for j in range(0,nodes):
31             if matrix[i][j] == 0 and i!=j and j>i:
32                 if random.randint(0,100) < 30:
33                     matrix[i][j] = random.choice(LOW_LEVELS
                           )
34     return matrix
35
36 def symmetrical(matrix, nodes):
37     for i in range(0,nodes):
38         for j in range(0,nodes):
39             if i < j:
40                 a = (matrix[i][j] + matrix[j][i])*1.0/2
41                 matrix[j][i] = a
42                 matrix[i][j] = a
43     return matrix
44
45 def real_traffic_values(matrix,nodes):
46     """ setting real mbit values in the matrix for sep.
          traffic
47     calculation, which is based on time * sep traffic """
48     matrix_real = numpy.zeros(shape=(nodes, nodes))
49     for i in range(0,nodes):
50         for j in range (0,nodes):
51             if matrix[i][j] == 4:
52                 matrix_real[i][j] = 216 # kilobits here
53             elif matrix[i][j] == 3:
54                 matrix_real[i][j] = 108
55             elif matrix[i][j] == 2:
56                 matrix_real[i][j] = 72
57             elif matrix[i][j] == 1:
```

```
58                  matrix_real[i][j] = 54
59      return matrix_real
60
61  def main(args):
62      matrix = numpy.zeros(shape=(args.nodes, args.nodes))
63
64      arms = divmod(int(args.nodes),int(args.groupsize))
65      if arms[1] != 0:
66          print "nodes not divisible by groupsize"
67          sys.exit(1)
68      else:
69          arms = int(arms[0])
70      matrix = set_dense_traffic(matrix, arms, int(args.
            groupsize), int(args.nodes))
71      if not args.zeros:
72          matrix = set_sparse_traffic(matrix, args.nodes)
73
74      matrix_real_asym = real_traffic_values(matrix,args.
            nodes)
75
76      if args.dump:
77          dump_matrix("asym.p", matrix)
78          dump_matrix("asym_real.p",matrix_real_asym)
79      else:
80          print "asym matrix not saved"
81          pprint.pprint(matrix)
82          pprint.pprint(matrix_real)
83
84      matrix = symmetrical(matrix, args.nodes)
85      matrix_real_sym = symmetrical(matrix_real_asym,args.
            nodes)
86
87      if args.dump:
88          dump_matrix("sym.p", matrix)
89          dump_matrix("sym_real.p",matrix_real_sym)
90      else:
91          print "sym matrix not saved"
92          pprint.pprint(matrix_asym)
93
94
95  if __name__ == "__main__":
96      parser = argparse.ArgumentParser(
97          description='Script makes a symmetric matrix')
98      parser.add_argument(
99          '-n', '--nodes', type=int, help='Number of nodes in
                LA', required=True)
100     parser.add_argument(
101         '-g', '--groupsize', type=int, help='number of
                nodes in each arm', required=True)
```

```
102     parser.add_argument(
103         '-d', '--dump', action='store_true', default=False,
                help='if true, dump bin files of matrices')
104     parser.add_argument(
105         '-z', '--zeros', action='store_true', default=False
                , help='if true, all zeros outside grous')
106     args = parser.parse_args()
107     main(args)
```

Script for spawning VMs

```
1   #!/usr/bin/env python
2   # -*- coding: utf-8 -*-
3
4   import random
5   import subprocess
6   import re
7   import pprint
8   import shlex
9   import sys
10  import argparse
11  from time import sleep
12  from threading import Thread
13
14  def remove_vm(nr):
15      try:
16          cmd = "/usr/bin/virsh undefine vm%s && /usr/bin/
                virsh destroy vm%s" % (nr, nr)
17          output = subprocess.check_call(cmd, shell=True)
18          return True
19      except:
20          return False
21
22  def spawn_vm(mac, nr, ram):
23      print "Spawning %s nr" % str(nr)
24      removed = remove_vm(nr)
25      subprocess.call(["virt-install",
26          "--virt-type","kvm",
27          "--name","vm%s" % str(nr),
28          "--ram=%s" % str(ram),
29          "--cdrom","my.iso",
30          "--network","bridge=br0,mac=%s" % str(mac),
31          "--nodisk",
32          "--vnc",
33          "--noautoconsole"])
34
35  def spawn_vms(macs, ram, sleeptime=17, use_thread=False):
36      threads = []
37      for nr, mac in enumerate(macs):
```

```
38          if use_thread:
39              thread = Thread(target=spawn_vm, args=[mac, nr
                    +1, ram])
40              print thread
41              thread.start()
42              threads.append(thread)
43          else:
44              spawn_vm(mac, nr+1, ram)
45              if not nr == len(macs)-1:
46                  sleep(sleeptime)
47      if threads:
48          print threads
49          for thread in threads:
50              thread.join()
51
52  def check_env(create_all=False):
53
54      cmd_get_nets = ("virsh net-list --all | egrep -v '"
55                  "^\s+?#|^\s+?$|^-Name' | awk {'print $1'}")
56      nets=subprocess.check_output(cmd_get_nets,shell=True)
57
58      if not nets:
59          print "No networks defined in libvirt!"
60          print "Need a network to continue. exiting..."
61          sys.exit()
62
63  def get_macs(dhcp_file, nodes):
64
65      with open(dhcp_file) as f:
66          content = f.readlines()
67      mac_regex = "([0-9a-fA-F]{2}:){5}[0-9a-fA-F]{2}"
68      macs = []
69      for line in content:
70          m = re.search(mac_regex,line)
71          if m:
72              if len(macs) < nodes: # num here is num of VMs
                    created
73                  macs.append(m.group())
74      return macs
75
76
77  def main(args):
78
79      if args.nodes:
80          check_env(create_all="True")
81      else:
82          check_env()
83
84      if args.nodes:
```

```
85          macs = get_macs('isc_dhcp_hosts', args.nodes)
86      elif args.vm_nr:
87          macs = get_macs('isc_dhcp_hosts', 100)
88
89      if args.vm_nr and macs:
90          spawn_vm(macs[args.vm_nr-1], args.vm_nr, args.ram)
91      elif args.nodes:
92          if args.thread:
93              spawn_vms(macs, args.ram, use_thread=True)
94          else:
95              spawn_vms(macs, args.ram)
96      elif not macs:
97          print "No mac addresses"
98
99  if __name__ == '__main__':
100
101     parser = argparse.ArgumentParser(
102         description='script for initiating traffic levels
                on VMs based on matrix')
103     parser.add_argument('-n', '--nodes', type=int,
104             help='number of nodes to spawn', required=False
                )
105     parser.add_argument('-r', '--ram', type=int,
106             help='amount of RAM for VMs', required=True)
107     parser.add_argument('-v', '--vm_nr', type=int,
108             help='case: spawn one vm, nr of it', required=
                False)
109     parser.add_argument('-t', '--thread', default=False,
            action='store_true',
110             help='Threads the spawning of vms')
111
112     args = parser.parse_args()
113
114     main(args)
```

Script for instantiating traffic among VMs

```
1
2   #!/usr/bin/env python
3   # -*- coding: utf-8 -*-
4
5   import subprocess
6   import argparse
7   import paramiko
8   import pickle
9   import pprint
10  import logging
11  from os import path
12  from sys import exit, exc_info
```

```
13 from threading import Thread
14
15 UDP_COMMAND = "sleep 30;/home/tc/udp_client.py -t %s -l %s
       -s %s &"  # ip, trafficlevel,strlen
16 USERNAME = "tc"
17 PASSWORD = "Rekesalat123"
18
19 logging.basicConfig(filename='set_traffic.log', filemode='a
       ', level=logging.DEBUG)
20
21
22 def get_matrix(matrix_file):
23     with open(matrix_file, 'rb') as input:
24         matrix = pickle.load(input)
25     return matrix
26
27
28 def ip_fix(nr):
29     """ Fixes relation between place nr and ip octet """
30     if nr < 9:
31         return "10%s" % str(int(nr)+1)
32     else:
33         return "1%s" % str(int(nr)+1)
34
35
36 def convert_levels(level):
37     return 5-int(level)
38
39
40 def host_connections(traffic_connection, ip_prefix
       ="192.168.1.%s"):
41     """ Iterates through the hosts list and finds out which
           servers to connect to
42     Input: traffic_connection: list of connectionrate,
          where host octet is item placement
43     Returns: Dict of host: connectionrate
44     """
45     hosts = {}
46     for i, traffic in enumerate(traffic_connection):
47         if traffic > 0:
48             hosts[ip_prefix % str(ip_fix(i))] = traffic
49
50     return hosts
51
52
53 def get_host_traffic(matrix):
54     """ Iterates over the matrix to get a dictionary with
55     Returns: Dictionary: {from: {to: trafficlevel}}
56     """
```

```
57      ip_prefix = "192.168.1.%s"
58      h_matrix = {}
59
60      for i in range(0, len(matrix)):
61          h_matrix[ip_prefix % str(ip_fix(i))] =
                host_connections(matrix[i])
62      return h_matrix
63
64
65  def run_cmd_on_hosts(hosts, cmd, username=USERNAME,
        password=PASSWORD):
66      """ runs command on remote linux system
67      hosts: should be list: [ip1,ip2,...,ipN]
68      cmd: string: any linux command
69      """
70      ssh = paramiko.SSHClient()
71      ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy
            ())
72      for host in hosts:
73          print host
74          try:
75              ssh.connect(host, username=username, password=
                    password, timeout=5)
76              stdin, stdout, stderr = ssh.exec_command(cmd)
77              ssh.close()
78          except:
79              print "Unexpected error:", exc_info()[0]
80              print "Command not run at host %s" % host
81
82
83
84  def instantiate(hostip, hostdict, strlen):
85      """ Runs the commands on the remote host (loops through
            external host and traffic levels)
86      Input: hostip: IP to run commands from
87          hostdict: externalip: trafficvalue
88      """
89      print "Connecting to %s" % hostip
90      ssh = paramiko.SSHClient()
91      ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy
            ())
92      ssh.connect(hostip, username=USERNAME, password=
            PASSWORD)
93
94      for external, level in hostdict.iteritems():
95          stdin, stdout, stderr = ssh.exec_command(
                UDP_COMMAND % (external, convert_levels(level),
                strlen))  # ip, traflev
96      ssh.close()
```

```
97
98
99  def main(matrix_file, do_thread, strlen):
100     matrix = get_matrix(matrix_file)
101
102     host_traffic = get_host_traffic(matrix)
103     threads = []
104
105     for host, hostdict in host_traffic.iteritems():
106         if do_thread:
107             thread = Thread(target=instantiate, args=[host,
                    hostdict])
108             thread.start()
109             threads.append(thread)
110         else:
111             instantiate(host, hostdict, strlen)
112
113     if do_thread and threads:
114         for thread in threads:
115             thread.join()
116
117
118 if __name__ == '__main__':
119     parser = argparse.ArgumentParser(
120         description='script for initiating traffic levels
               on VMs based on matrix')
121     parser.add_argument(
122         '-m', '--matrix_file', type=str, help='matrix file
               to read from',
123         required=True)
124     parser.add_argument(
125         '-s', '--strlen', type=int, help='length of string
               in UDP packet',
126         required=True)
127     parser.add_argument('-t', '--thread', default=False,
           action='store_true',
128             help='Threads the instantiating of traffic')
129
130     args = parser.parse_args()
131     print args
132     if not path.exists(args.matrix_file):
133         print "File %s does not exist" % matrix_file
134         sys.exit(1)
135
136     main(args.matrix_file, args.thread, args.strlen)
```