

Agent-based management of Virtual Machines for Cloud infrastructure

Alexandros Vichos



Master of Science
Computer Science
School of Informatics
University of Edinburgh
2011

Abstract

Cloud computing has become increasingly popular over the past few years. With the majority of its services relying on virtualisation, the number of virtual machines hosted by cloud providers is growing at a fast pace. As the number of virtual machines grows, the need for automated virtual machine management solutions becomes apparent. Currently the industry uses centralised management solutions. This project experiments towards an alternative system design using a decentralised approach. In this approach every server becomes an agent and is assigned a policy. All agents then interact in a peer-to-peer fashion, exchanging virtual machines according to their policies. In this way, the management problem is tackled by the individual servers of the system without the need for centralised control. Finally, the project reaches a feasible system design which is in turn implemented in a working solution.

Acknowledgements

I would like to especially thank my supervisor, Paul Anderson, for his help on all technical issues and for his guidance throughout this project.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Alexandros Vichos)

Table of Contents

1	Introduction	1
2	Background information	3
2.1	Virtualisation	3
2.1.1	libvirt	4
2.2	Cloud computing	4
2.3	Agent-based systems	5
2.4	OpenKnowledge and LCC	6
3	Related work	9
3.1	Sandpiper	9
3.2	A Framework for Autonomic Performance Management of Virtual Machine- Based Services	10
3.3	VIOLIN	11
3.4	Non-automated management tools	12
4	Description of work done	15
4.1	Setup and Configuration	15
4.1.1	Installing KVM	16
4.1.2	Installing Libvirt	16
4.1.3	Libvirt Java API	18
4.1.4	Applying the configuration	19
4.2	First steps	20
4.2.1	Working with libvirt	20
4.2.2	Working with the libvirt Java API	22
4.2.3	Working with OpenKnowledge and LCC	22
4.3	Design approaches	24
4.3.1	First approach	25

4.3.2	Second approach	26
4.3.3	Third approach	29
4.4	Implementing the system	30
4.4.1	Implementing policies	33
5	Usage Scenarios	35
5.1	Scenario 1	35
5.2	Scenario 2	36
5.3	Scenario 3	37
5.4	Scenario 4	39
6	Conclusions	41
6.1	Ideas for future work	41
6.2	Conclusion	42
A	Policy source code	45
A.1	Scenario1Peer.java	45
A.2	Scenario2Peer.java	46
A.3	Scenario3Peer.java	47
A.4	Scenario3ShutdownPeer.java	48
A.5	Scenario4HighLoadPeer.java	49
A.6	Scenario4HighMemoryPeer.java	50
A.7	Scenario4HighShutdownPeer.java	52
	Bibliography	53

Chapter 1

Introduction

Cloud computing is a new revolutionary concept for providing computational services and resources. Its main advantage is that the services are provided on demand, without the need for commitment, and are tailored to each customer's requirements. This allows the customer to adjust the resource supply according to his needs, while getting charged based on resource usage. Cloud computing offers an attractive solution for both businesses and individual customers and thus cloud services are becoming increasingly popular.

Virtualization technology plays a key role in the development of cloud services. The use of virtual machines has enabled the creation of custom independent systems in a timely manner, regardless of the underlying hardware. Consequently, cloud service providers greatly base their services on virtual machines, as they enable them to provide resources with great flexibility.

In order to cope with the increasing demand for cloud services, cloud service providers have to create and maintain a large number of virtual machines. Since virtual machines reside in physical machines, a substantial issue arises as to where each virtual machine should be placed. Virtual machine placement depends on many factors including the system's current state, each virtual machine's resource requirements as well as the provider's policy. The vast variety of system states combined with possibly competing restrictions make virtual machine management a truly complex problem.

In an effort to tackle the virtual machine management problem, cloud service providers are currently using a centralised approach. In this approach a central entity gathers information about the state of all the physical machines in the system, and subsequently makes decisions about the virtual machine placement. While this approach works efficiently for smaller systems, it can prove inadequate as these systems

scale to millions of virtual machines.

This project tackles the virtual machine management problem from a different angle. Instead of using the standard centralised approach, it investigates the combination of agents and peer-to-peer systems as an alternative solution to this problem. The experimental work done in this project is based on the idea of a system that would be able to manage itself through the use of policies specified on each physical machine. In this system, each physical machine would be governed by an agent whose goal would be to make the machine conform to its policy. Consequently, the agent would have to interact with other agents and exchange virtual machines in order to achieve its goal.

Following this approach, this project experiments with the OpenKnowledge [26] peer-to-peer system towards a design for an automated virtual machine management system. The experiments also include practical work on a cluster of four servers. Finally, the project results in a feasible system design which is in turn implemented in a working solution.

The rest of this document is organised as follows:

- Chapter 2 includes background information on virtualisation technology, cloud computing, agent-based systems as well as on the OpenKnowledge peer-to-peer system.
- Chapter 3 presents related work on automated virtual machine management systems.
- Chapter 4 describes the work done on configuring the cluster, designing the system, and implementing it.
- Chapter 5 demonstrates the functionality of the resulting system through usage scenarios.
- Chapter 6 presents ideas for future work and conclusions from this project.

Chapter 2

Background information

2.1 Virtualisation

Virtualisation is a technology that allows multiple instances of operating systems to run in parallel on a single physical machine. This functionality is achieved through the creation of multiple *Virtual Machines* that host the different operating systems. Virtual machines provide an isolated environment for the operating systems which is equivalent to that of a physical machine. Moreover, while running in parallel, virtual machines share the resources of the same underlying physical machine. Although they utilise the same hardware, virtual machines run separately and do not interfere with each other. Consequently, virtualisation technology allows single application servers to be divided into multiple virtual servers and take advantage of the idle server resources.

Virtualisation is made possible by adding a new layer between the hardware and the operating system layer. The *virtualisation layer* (also called *hypervisor*) hides the underlying physical architecture by presenting the operating system layer with a set of generic hardware components. The operating systems then gain access to the physical hardware through the use of appropriate drivers for these generic components. In this way, architecture compatibility issues are dealt in the virtualisation layer, enabling virtual machines to run regardless of the physical hardware. In addition, the virtualisation layer is responsible for managing the system's hardware by controlling the operating systems' accesses to resources. Accesses to the virtual hardware components are transferred to the physical hardware, while the results are then passed on to the requesting operating system. This project uses the KVM [1] hypervisor as part of its virtualisation solution.

Configuring the virtualisation layer as well as performing virtual machine related

tasks is done through management interfaces called *Virtual Machine Managers*. Virtual machine managers enable the user to create, destroy and configure virtual machines, control the physical resource allocation and configure the way in which the physical hardware is virtualised (e.g. create virtual network interfaces). Additionally, they provide a virtual machine monitoring service as well as a set of actions to start and shutdown individual virtual machines.

Another important feature provided by virtual machine managers is *live migration*. Since virtual machines depend solely on the hardware components provided by the virtualisation layer, it is possible to migrate virtual machines from one physical machine to another. Live migration though is restricted, and can occur only between hosts that run the same hypervisor. In order to achieve live migration, both the hard disk content and the memory content of the virtual machine must be transferred to the new host. Currently, memory content is transferred via network while hard disk content must be available on shared storage. Once all the required data has been transferred, the virtual machine manager switches the execution to the new host, while the virtual machine continues functioning uninterrupted.

2.1.1 libvirt

As previously mentioned, Virtual Machine Managers are an integral part of every virtualisation solution and thus are also required for this project. The virtual machine manager chosen for this project is libvirt [2]. Libvirt offers features similar to those of the majority of virtual machine managers. Its features also include live migration which was a requirement for this project. Moreover, it works on top of a number of hypervisors including the KVM hypervisor which is used in this project. However, the main reason for choosing libvirt was that it additionally provided an API. The presence of an API facilitates libvirt's usage in a programming environment and offers better integration compared to using it as an external application.

2.2 Cloud computing

Cloud computing has brought a new era in IT industry by offering a flexible way for creating IT infrastructures. Cloud service providers are able to provide computing resources on demand, thus enabling the quick deployment of IT infrastructures without the need of constructing them. Moreover, the solutions offered are fully customisable

in order to fit any organisation's needs while changes in the configurations are applied very quickly.

Cloud services are organised in three categories: Infrastructure as a Service, Platform as a Service and Software as a Service [19]. Infrastructure as a Service, is the provision of computer resources equivalent to those of an IT infrastructure. Cloud service providers create customised systems on top of their hardware infrastructures in the form of virtual machines. The customer is then given access to these machines and is able to control them and use them as application servers. Platform as a Service is used to provide computational resources in a different manner. In this case, the cloud service provider has developed a platform on which the customer can run custom applications. Usually the platform can provide large amounts of computational power which would otherwise be unobtainable for the customer. Finally, Software as a Service refers to the provision of software products on demand, for use within the cloud environment. In this manner, the software is licensed to the customer for as long as it is actually used and consequently the licensing fee is lower.

Virtualisation technology play a key role in the cloud computing industry. One of the main services in the cloud is virtual machine provisioning, given as an alternative to physical infrastructures. As a result, cloud service providers employ multiple virtual machines on their servers. Cloud service providers greatly benefit from the flexibility virtualisation offers in terms of customisation, as well as from the security issues that it resolves. Moreover, the ease of virtual machine creation and the consequential efficiency in hardware utilisation, is what makes cloud computing possible.

Due to its beneficial properties, cloud computing is becoming an increasingly popular solution for both small and larger organisations. Nevertheless, there are still issues in the cloud industry mainly regarding trust and security [19]. As the confidentiality and security of information is critical for most organisations, there is also great hesitance in migrating IT services to the cloud.

2.3 Agent-based systems

Agent-based systems are software systems that use agents to perform problem solving or other computational tasks. In agent-based systems, the system's task is assigned to autonomous software entities called *agents* which in turn cooperate with each other in order to complete it.

Agents are a special category of computer programs that in contrast to conventional

programs have the ability to act autonomously. An agent is programmed as to be able to perceive a specific environment and additionally act upon it. Moreover, each agent has a specific objective, a goal, and thus it must take action upon its environment in order to achieve it. An agent's autonomous behaviour derives from the ability of being both proactive and reactive. By being proactive, an agent adjusts its behaviour and plans its actions as to achieve its initial goal, while by being reactive it is able to respond to changes in its environment in a timely manner. It is also possible for agents to learn by gathering information from their environment and their previous actions. This information is then stored internally in the agent in the form of *beliefs* which can ultimately affect the agent's behaviour.

In addition to being autonomous, agents have the ability to engage in social interactions with other agents. Their social abilities enable them to exchange information, to cooperate in order to achieve their goals and to coordinate their actions. Agents can also have different roles in a system and may influence the behaviour of other agents or even control them by requesting specific actions.

Their unique design and its provided features along with their social abilities make agents suitable for creating complex systems [21]. Using agents can simplify the design and implementation of such systems, as not all possible links, interactions and states will have to be considered. Instead, agents can be programmed with specific behaviours that will enable them to deal with unknown states and interactions as they occur. Furthermore, agents can either be used individually, by assigning each one of them to work on a specific aspect of the problem or together, by letting them cooperate as to solve a problem in a distributed fashion.

2.4 OpenKnowledge and LCC

Since its development, the web has been used by individuals as a means to share ideas and exchange information. Today a great amount of knowledge exists in the web, both from individuals and from systems, yet finding and using that knowledge still remains difficult. The main reason behind this problem is that accessing and understanding knowledge in order to use it often requires expertise, either due to the information's nature and context, or because of increased complexity in the process of acquiring it.

OpenKnowledge tries to address this issue through a peer-to-peer solution, where both the knowledge and the ways it can be obtained are specified within the system. In this manner, peers in the OpenKnowledge network can “interact productively with one

another without any global agreements or pre-run-time knowledge of who to interact with or how interactions will proceed” [3]. Peers connecting to the OpenKnowledge network can be either individual users or services.

Coordination is facilitated in the OpenKnowledge network through the use of interaction models. Interaction models are similar to protocols as they describe all the steps required for a given interaction to complete successfully. In addition, interaction models specify roles that imply different responsibilities and actions within an interaction. Moreover, they are written using the Lightweight Coordination Calculus (LCC) [23] syntax in order to be simple and understandable. OpenKnowledge also includes a complementary discovery service that peers can use to discover interactions in which they can participate.

Another important aspect of the OpenKnowledge system is the use of components. OpenKnowledge components provide a way to extend an interaction’s functionality to include more sophisticated tasks. The use of components is integrated in the interaction models in the form of constraints that return a boolean value. Thus, when a specific component’s functionality is required, a constraint is added to the corresponding step of the interaction, requiring the component’s output before the interaction can continue. However, the component may also return a value of “false” in which case the respective step of the interaction fails.

```
a(requester, A) ::
  ask(X1) => a(informer, p2) <-- query_from(X1, p2) then
  tell(X1) <= a(informer, p2) then
  ask(X2) => a(informer, p3) <-- query_from(X2, p3) then
  tell(X2) <= a(informer, p3)

a(informer, B) ::
  ask(X) <= a(requester, B) then
  tell(X) => a(requester, B) <-- know(X)
```

Figure 2.1: Interaction model example (taken from [3])

Figure 2.1 shows an example interaction model demonstrating the use of LCC and OpenKnowledge components. In this case the interaction includes two roles, the “requester” and the “informer”. Additionally, the interaction uses two OpenKnowledge components namely “query_from()” and “know()”. The example describes a simple

scenario where a requester asks an informer for information on behalf of some other peer. In turn, the informer replies to the respective peer, but only in case it has the requested information. Communication among the peers is achieved through “ask” and “tell” messages while OpenKnowledge components extend the interaction’s functionality by performing additional computations and providing the required information.

Every time a peer wishes to use the OpenKnowledge network, it creates a new interaction according to the task it wants to accomplish. Once the new interaction is created, the peer subscribes to it by assuming the role that it wishes and publishes the interaction. By making the interaction public, other peers can discover it and decide whether they want to participate in it. The interaction can only begin once all the defined roles are assigned to OpenKnowledge peers. Eventually the peers cooperate according to the interaction model until all the steps are successfully completed.

Chapter 3

Related work

This chapter presents work done on automated virtual machine management systems by the scientific community. The systems presented below are closely related to this project as they tackle the same problem. However, the approach used in this project is a decentralised one as opposed to the popular centralised approach. Furthermore, only centralised systems were reviewed as a purely decentralised approach has not been published yet.

3.1 Sandpiper

Sandpiper [27] is an automated virtual machine management system that focuses on load balancing. Its main objective is to try and resolve “hotspots” by placing the virtual machines in such a way so that they can perform smoothly under stress. The system is designed explicitly to work with the Xen [4] hypervisor.

In order to achieve its purpose, Sandpiper follows two approaches namely the “black-box” approach and the “gray-box” approach. In the “black-box” approach, the system performs the management task based solely on the information reported by the Xen hypervisor running on every physical machine. Consequently, this first approach takes a view of the managed infrastructure on the physical machine level. The “gray-box” approach enhances the accuracy of the first approach by also considering additional information regarding the status of the individual virtual machines. In this case Sandpiper employs daemon processes that execute and report from within each virtual machine.

In terms of architecture, Sandpiper is considered a centralised system, as the majority of its functionality resides on the central node. Every physical machine runs a

daemon process that gathers live system information about itself and additional per virtual machine information when using the “gray-box” approach. This information is then reported to the central node, which in turn profiles the physical machines. Based on these profiles, the central node then makes migration decisions to resolve “hotspots” (if any are present) using a greedy algorithm.

As previously stated, Sandpiper is intended to resolve “hotspots” and thus its functionality cannot be further extended to include other scenarios. Testing the system shows good results for a small number of nodes (16 physical machines) while the main (greedy) algorithm performs decently, reporting results in 5 seconds for 50 virtual machines. Moreover, the “gray-box” approach resolves “hotspots” more efficiently than the “black-box” approach, but on the cost of running an additional monitoring process inside every virtual machine. Although Sandpiper achieves its purpose in small scale systems, with both network and cpu overhead kept to a minimal, its proposed solution is not well suited for today’s Cloud data centres which consist of large numbers of physical machines and consequently even greater numbers of virtual machines.

3.2 A Framework for Autonomic Performance Management of Virtual Machine-Based Services

The system presented in [25] is yet another automated virtual machine management system. Its intended functionality focuses on providing QoS to the individual virtual machines under management while minimising the migration and management costs. The QoS policy for each virtual machine is determined by one or several predefined SLOs (Service Level Objectives). Whenever a SLO fails for a virtual machine, the system takes action as to provide the virtual machine with the resources needed to maintain its QoS level.

The proposed system is composed of three management components: the PM Manager (Physical Machine Manager), the VM Manager (Virtual Machine Manager) and the Pool Manager.

An instance of the PM Manager runs inside every physical machine in the system under management. The PM Manager is then responsible for monitoring the status of its physical machine and periodically reporting it to the Pool Manager. Moreover it handles requests from the Pool Manager concerning the migration of virtual machines. One of the PM Manager’s great advantages is that it can be easily adapted and used on

top of any virtual machine hypervisor.

Gathering information from the virtual machine level is achieved through the VM Managers. Every virtual machine runs a separate copy of the VM Manager that monitors its status and tries to determine whether all of its SLOs are satisfied. Once a SLO fails, the system has to migrate the virtual machine to a better host that meets its current resource requirements. The first part of this process is handled by logic modules inside the VM Manager that notify the Pool Manager with requests for additional resources in the form of “need more of resource X”.

The largest part of the decision making process takes place in the Pool Manager. In order to compute allocations for the managed virtual machines, the Pool Manager collects status information from the individual PM Managers, and processes requests for resources from the VM Managers. Once the new allocations are computed, the Pool Manager applies them by sending migration requests to the relevant PM Managers. Virtual machine allocations are computed using two algorithms. The first one tries to find the required resources in the physical machine pool and then places the virtual machines accordingly. The second algorithm takes this concept even further by also considering the migration cost as well as the value of the resulting system state.

Although [25] is considered a centralised system with the Pool Manager being the central node, it is a more balanced one, as part of the decision making process is also handled by the VM Managers. Overall, [25] presents a great concept but the implementation is in its initial stages, with tests conducted using a single SLO (memory usage). Finally, the centralised Pool Manager might be a bottleneck in larger systems as [25] presents no evidence to support the opposite.

3.3 VIOLIN

VIOLIN [24] is an older system that was proposed when virtualisation technology first made its appearance in data centres. Like all the systems presented so far, VIOLIN is an automated virtual machine management system. VIOLIN’s main purpose is similar to that of [25] as it tries to better allocate the available resources to virtual machines that require them, in order to increase their performance. Resource allocation occurs in two phases. During the first phase VIOLIN tries to redistribute the resources locally, between virtual machines on the same host. Whenever this first phase fails, due to limited resources on the host physical machine, VIOLIN proceeds to the second phase where it tries to find the required resources on a different host.

VIOLIN's management decisions are computed through the use of special components called "adaptation managers". Adaptation managers can either be local or global according to their level of operation. Local adaptation managers operate on the physical machine level, gathering information about the status of the hosted virtual machines and reallocating resources locally to the virtual machines that require them. Whenever the local resources become insufficient due to high demand from the local virtual machines, the decision making process is then handled by the global adaptation manager. The global adaptation manager runs on a central host and operates on system-wide level, gathering information for all the virtual machines under management by periodically querying the local adaptation managers. Eventually, it creates a global view of the system and in turn makes global resource reallocations by migrating virtual machines between physical hosts.

Besides being able to make system-wide resource allocations, VIOLIN also supports inter-domain virtual machine management. Working with multiple domains gives VIOLIN more flexibility as it can utilise the resources of any other domains it manages. Inter-domain management is achieved by employing an additional global adaptation manager that operates on the domain level. Hence, whenever there are insufficient resources in one domain, this additional global manager will try to find the required resources in another domain and migrate the virtual machines there.

Experiments show that the system achieves autonomic load balancing as well as effective resource allocations. Moreover, VIOLIN is the only system that was deployed in a real world environment (nanoHUB) with real users, and run — as they claim — seamlessly. Another interesting remark is that the combination of both local and global adaptation managers distributes the management task among the physical machines and the central node, while also yielding better allocations as opposed to a purely centralised system. However, it is not clear from the experiments whether VIOLIN could perform equally well in large data centres with hundreds of physical machines.

3.4 Non-automated management tools

The systems presented so far provide automated solutions to the virtual machine management problem. However, there are also many tools that do not incorporate any kind of intelligence but aim at facilitating the management process when it has to be done manually. These tools provide a wide variety of features to the user and are usually accompanied by a graphical user interface.

Using these tools enables the user to monitor the status of virtual machines, dynamically manage the resources that are allocated to each one of them as well as easily create and destroy virtual machines. Moreover, the user can also choose to migrate a virtual machine to a different host whenever this is required. This way, the user can effectively manage the system in hand without having any prior knowledge about hypervisor specific commands or having to deal with complicated configuration files.

Most of these tools are designed to work on top of a specific hypervisor. Examples include “XenServer” [5] which runs on top of the Xen hypervisor and “vCenter” [6] that runs on top of VMWare. However, tools such as “Virt-manager” [7] and “VirtualIQ” [8] support multiple hypervisors.

Chapter 4

Description of work done

4.1 Setup and Configuration

One of the main challenges of this project was that it required working with a number of different technologies. Combining these technologies towards a good system design also involved experimental work and constant testing of the different design approaches. Hence, a large part of this project was dedicated in properly setting up a working environment.

In terms of hardware, this project required an array of servers that support hardware virtualisation. The hardware requirements were eventually fulfilled, with HP [9] donating four ProLiant DL120 G6 servers to the University of Edinburgh as to be used for such projects. The specifications for the HP servers are the following:

- Processors: 2x Intel Xeon CPU X3450 @ 2.66GHz
- Storage: 600 GB on RAID-5
- Memory: 4 GB DDR-3
- Network controller: Gigabit Ethernet

The initial setup of the HP cluster was done by my supervisor, Paul Anderson. It included a complete installation of Scientific Linux 6 [10] (SL6) and was done using LCFG [18] as to facilitate the configuration and management of the cluster. Moreover, the setup included a working OpenKnowledge discovery service required for the OpenKnowledge peer-to-peer network, as well as shared storage (NFS [11]) required for virtual machine migration. Lastly, all four servers were available on the uni-

versity's network under the domain *diy.inf.ed.ac.uk* with their hostnames being *hp[1-4].diy.inf.ed.ac.uk* respectively.

The following paragraphs give a detailed description of the steps I took in order to install and configure the necessary tools for this project on the HP cluster. Initially, the setup process took place on only one of the HP servers. The complete configuration was later applied to the whole cluster through LCFG.

4.1.1 Installing KVM

KVM is used in this project as the default hypervisor. In order to install KVM on a system, it must be supported by both the system's hardware and operating system.

In terms of hardware support, the system's CPU must have hardware virtualisation enabled. Checking for CPU compatibility in SL6 can be done by executing `cat /proc/cpuinfo` and looking for the *vmx* flag in the processor's flag list. In my case, the *vmx* flag was present, as the Xeon X3450 CPU had support for hardware virtualisation enabled.

In terms of OS support, the OS's kernel must be compiled with KVM support. Checking whether the system's kernel supports KVM can be simple done by trying to load the necessary kernel module, in this case *kvm-intel*. Loading the module by executing `modprobe kvm-intel` returned successfully with no errors, indicating that SL6's kernel was compiled with support for KVM.

Knowing that the system fully supports KVM I then proceeded with the actual software installation. SL6 is a RedHat [12] based Linux distribution, so installing software can be done through the *yum* [13] package manager. The packages required for a basic KVM installation are *qemu-kvm* and *bridge-utils*, both of which were available on the SL6 repositories. Finally, I installed the required packages by issuing the following command: `yum install qemu-kvm bridge-utils`

At this point, KVM was not configured further and no additional utilities were installed. Nevertheless, *libvirt* was used to control and configure KVM.

4.1.2 Installing Libvirt

Libvirt is one of the most useful pieces of software involved in this project. Apart from facilitating the configuration and control of the underlying hypervisor (in this case KVM) it can also be used through its API thus providing all its functionality to the programming level.

Installing libvirt on SL6 can be easily done using the *yum* package manager. The necessary packages in this case are `libvirt` and `virt-manager`. Hence, I installed the above packages with the following command: `yum install libvirt virt-manager`

4.1.2.1 Configuring Libvirt

Having successfully installed libvirt the next step was to configure it. Libvirt, as most of Linux software, is configured through configuration files. In this case the relevant files are `/etc/libvirt/libvirtd.conf` and `/etc/sysconfig/libvirtd`. The first file contains the core configuration of libvirt while the second is used to define additional environment variables that are passed to libvirt during startup. The complete configuration files for libvirt are also included in this project's files.

One of the most important parts of libvirt's configuration, was choosing the authentication method. Libvirt is implemented as a service which can then accept both local and remote connections. Connecting to a libvirt service is necessary both for creating and managing virtual machines locally, as well as for migrating virtual machines between remote hosts. Libvirt provides three authentication methods: SSH, SSL/TLS and Kerberos. Using the SSH method, requires root (superuser) access to the target machine, which could lead to security issues. Moreover, using the Kerberos method was not practical for this project as it would require setting up a separate Kerberos service. Thus, the SSL/TLS method was chosen over the other two.

Authenticating using the SSL/TLS method is done through the use of SSL/TLS certificates. These certificates must be present in both the server and the client participating in the connection. In addition, the certificates must be signed by the same Certificate Authority (CA) and have their Common Names (CN) set to their corresponding hostnames. In order to generate these certificates, I used a utility called *certtool* which is part of the `gnutls-utils` package. First, I created a CA in order to be able to sign the rest of the certificates. Then for every server in the cluster, I generated a certificate along with a private key. Finally, I used the CA's private key to sign the server certificates. Apart from the server certificate, each server should also hold a client certificate. As mentioned above, the servers need to be able to connect to each other in order to migrate virtual machines. Hence, using the same procedure as before, I also generated client certificates for each of the servers. Lastly, I moved all the generated certificates and private keys of each host to their default locations, under `/etc/pki/libvirt/`.

The rest of libvirt's configuration involved editing the two configuration files mentioned above. In particular, I specified the correct hostname for each server as well as

the locations of the server's certificates, private key and the CA's certificate. Finally, I enabled incoming SSL/TLS connections as well as listening for connections on all network interfaces.

4.1.3 Libvirt Java API

Another requirement for this project was libvirt's API. The API enables the programmer to establish a client connection to an instance of libvirt (either local or remote), and interact with it. Originally, the API was written for use with C and C++, though bindings are also available for other programming languages. In this project we are interested in the Java version of the API. The original C API is exposed in Java through the use of JNA [14] which allows C libraries to be loaded and used in Java programs. Thus, JNA must be present in the server's classpath for the API to function correctly.

The first step I took in order to properly setup libvirt's API was to get the latest version of JNA. JNA's latest version was available at the JNA git repository both in source code format and in pre-compiled *jar* format. Since the library was already available in jar format, I chose using it directly instead of compiling it myself. Finally, JNA's jar was downloaded and added to the server's classpath.

The next step was getting the libvirt Java API. Unfortunately in this case there was no pre-compiled version available, so the Java API had to be built manually. In order to do so, I first checked out the latest version of the Java API from libvirt's git repository using the following command:

```
git clone git://libvirt.org/libvirt-java.git
```

Having the source code available, I then compiled it using ant by issuing the following command:

```
ant build
```

The build process finished successfully producing libvirt's Java API in jar format. Finally, I added the newly created jar in the server's classpath. Moreover, to verify that the API was indeed usable, I wrote and compiled a simple test class that made use of it.

4.1.4 Applying the configuration

So far the installation and configuration has been taking place only on one of the cluster's servers. But for the cluster to work as intended all the servers need to have identical configurations. In essence, this requires all servers to have the same packages installed and configured. Similarly, all the required libraries needed to be present on all four servers.

In order to ensure that all four server installations were identical in terms of packages, the installation specifications had to be transferred to the individual LCFG profile of every server. For this purpose, all the required packages had to be specified in LCFG package lists [18]. Using these package lists enables LCFG to automatically install the packages on all servers.

LCFG package lists are plain text files in which every line explicitly specifies a software package that should be included in the system's installation. The package specification includes the name of the package, its version and optionally its architecture. Compiling the necessary package lists involved getting the exact version of the required packages. Moreover, it also involved specifying every package's additional dependencies as to ensure that the configuration will not break during future system updates. The additional dependencies for every package were gathered by issuing the following command:

```
yum deplist package-name
```

Yum's output included all the required dependencies for the given package. The output along with the original package specification was then reformatted using regular expressions as to match the LCFG package list format. Finally, the complete package lists were included in each server's LCFG profile.

In addition to having the same software installed, the servers are also required to use the same libraries. This further increases the setup's consistency and ensures that all servers will behave in a similar way. For this purpose, all the libraries were moved to the shared storage, where they could be accessed by all servers. Moreover, the classpaths on each server were reconfigured as to include the new location of the libraries. Setting up a special shared folder for the libraries proved very useful along the project's way, as it made the task of including additional libraries much simpler.

4.2 First steps

A large part of this project was dedicated in understanding the technologies in hand. Experimenting with the tools involved in this project was more than necessary in order gain the knowledge required to produce a good system design. This section describes my first experiences from working with these tools.

4.2.1 Working with libvirt

In order to be able to work with libvirt, one of the first things that needs to be done is installing a virtual machine. For the purposes of this experiment, I decided to install the latest 64-bit version of Debian Linux on a virtual machine. Installing a virtual machine using libvirt is done through the *virt-install* utility. The initial virtual machine configuration is specified through command line arguments. Using these arguments *virt-install* then automatically creates all the configuration files required and registers the virtual machine with libvirt. Figure 4.1 shows the command issued in order to install the Debian virtual machine.

```
virt-install --connect qemu:///system --name debian-demo --ram 512
--disk path=/nfs/vms/alexandros/vms/debian-demo.img,size=3
--network network=default,model=virtio
--cdrom=debian-6.0.1a-amd64-i386-netinst.iso --os-type=linux
--os-variant=debiansqueeze --vnc --vnclisten=0.0.0.0
```

Figure 4.1: Virtual machine installation command using *virt-install*

The command above creates a new virtual machine on the local server under the name “debian-demo” with 512 MB of RAM and 3 GB hard disk drive stored in the image file “debian-demo.img”. The `--vnc` (Virtual Network Computer) switch is used to enable viewing and controlling the virtual machine over the network. Likewise the `--vnclisten` switch enables listening for VNC connections on all of the server’s network interfaces, as to allow both local and remote connections.

Libvirt uses a special URI syntax in order to define connections to libvirt hosts and their respective virtual machines. The URIs use the following syntax:

```
hypervisor+protocol://hostname/virtual_machine_name
```

As QEMU and KVM are essentially part of the same project, “qemu” is used to specify both

hypervisors. Moreover, the protocol field is optional, while leaving it empty defaults to TLS. Another special case of URIs can be seen in the installation command above. Specifying no hostname results in connecting to the local instance of libvirt. In addition, using the keyword “system” instead of a virtual machine name results in connecting to libvirt’s daemon process as opposed to an actual virtual machine.

In order to continue with the virtual machine’s installation, a VNC connection was required as to be able to view its output and control it. As the installation was taking place on a remote host (hp1) I had to use a VNC client from my local machine. For this purpose, I installed libvirt’s *virt-viewer* on my machine and tried to connect to the remote virtual machine using the following command:

```
virt-viewer qemu+tls://hp1.diy.inf.ed.ac.uk/debian-demo
```

Unfortunately, the command failed as my local machine did not have the necessary SSL/TLS certificates to establish a connection to the remote host. Therefore, I generated the required client certificates following the process described in the configuration section, and retried. The second attempt was successful and I was able to complete the installation.

Having installed a virtual machine, I was ready to start experimenting with libvirt’s features. Interacting with libvirt is done through a special command shell called *virsh*. *Virsh* comes as part of the default libvirt installation and enables the user to control the underlying virtual machines, view their status, as well as to migrate them to other hosts running libvirt. Executing *virsh* must always be done with root privileges as its functionality is not available to normal users by default.

The initial experiments involved using *virsh* for basic tasks such as starting and shutting down virtual machines, viewing their status and statistics. Disappointingly, the commands for changing a virtual machines memory and CPU resources (namely *setmaxmem* and *setvcpus*) did not work “on the fly” as the operations were not supported.

The next step was to try out the migration functionality of libvirt. Libvirt had already been installed on all four servers of the HP cluster, hence the virtual machine running on hp1 could be migrated to any of the other hosts. Additionally, the virtual machine was installed on the shared storage, thus satisfying the basic requirements for migration. Knowing that migration should be possible with this setup, I proceeded in migrating the “debian-demo” virtual machine. After starting *virsh* I issued the following command:

```
migrate --live debian-demo qemu+tls://hp2.diy.inf.ed.ac.uk/system
```

The command succeeded and after a few seconds the virtual machine was transferred to hp2.

Listing the virtual machines on hp2 and accessing it through virt-viewer verified that the migration was indeed successful. The additional `--live` switch ensured that the migration would occur seamlessly without affecting the virtual machine's execution. Further tests including migrating the virtual machine while performing CPU and network related activities also verified this last point.

4.2.2 Working with the libvirt Java API

Having already worked with libvirt through `virsh`, working with the libvirt Java API required only few adjustments. Similarly to `virsh`'s case, using the API requires setting up a connection with libvirt. In this case though, connecting to the local libvirt is not done automatically. On the contrary, the connection is declared manually by creating a new instance of libvirt's "Connect" Java object. The following Java statement demonstrates the creation of such an object:

```
conn = new Connect("qemu:///system", false);
```

`Connect`'s constructor takes two arguments. The first argument is a URI specifying a host running libvirt while the second argument is a boolean value that states whether the connection should be "readonly". Creating a "readonly" connection effectively limits the connection's permissions as it allows only getting information from libvirt and disables commands that actually act on the system (e.g. migrating a virtual machine). After the "Connect" object has been created, all of libvirt's functionality is available to the programmer through that object.

In order to familiarise myself with libvirt's Java API I performed simple tasks such as starting and stopping virtual machines as well as getting information about their status. This involved writing several small programs that used the methods provided by libvirt's "Connect" object in several different ways. Lastly, I also tried migrating virtual machines through the API. An important note concerning this project is that all programs that connect to the local libvirt using the API should be executed with root privileges.

Libvirt's API proved very usable and flexible. For the most part, its usage was very intuitive and well documented. Moreover, the exceptions thrown by the API's methods provide control over the program's execution and return informative messages in erroneous cases.

4.2.3 Working with OpenKnowledge and LCC

Working with OpenKnowledge was a significantly different experience from all the other tools used so far. Although it is based on the same principles as any other peer-to-peer system, OpenKnowledge introduces a new type of service and concepts that are a bit difficult to grasp at once. Hence, understanding the OpenKnowledge structure was essential before actually

designing a system that relied on it.

Specifically, the main focus here was to understand how LCC interaction models work and how OpenKnowledge components are created. Unfortunately there was too little documentation on both, hence understanding their logic was done by studying examples as well as the OpenKnowledge source code.

Since an OpenKnowledge discovery service was readily available on the cluster, the first step was to reproduce the examples from the OpenKnowledge website. Through this process the basic concepts of OpenKnowledge became more clear as the examples demonstrated sample problems and the relevant applications of OpenKnowledge. Moreover, reproducing these examples also gave me the chance to learn about the practical things involved in OpenKnowledge, such as initialising peers and publishing interaction models.

Having studied and reproduced the OpenKnowledge examples, I then proceeded with some experiments in order to further explore OpenKnowledge's functionality. Based on the existing examples, I created new interaction models and OpenKnowledge components that were more relevant to the purposes of this project. In doing so, I discovered a number of important issues that are not well discussed as well as gaps in the documentation.

Although OpenKnowledge interaction models are primarily written in LCC, they also contain additional role declarations prior to the LCC code. These declarations are included as to specify interaction specific parameters about the roles involved. Figure 4.2 shows an example. Such role declarations are present in all the example interaction models, but their exact syntax

```
r(peerGreeter, initial)
r(peerResponder, necessary, 1)
```

Figure 4.2: Role declaration in OpenKnowledge interaction models (taken from [3])

and semantics are undocumented. The following specification was derived by combining bits of information and experimenting with interaction models.

Every role declaration begins with the letter “r” and is followed by a comma separated list of parameters in parentheses. The first parameter specifies the name of the role and must start with a lower case letter. The second parameter specifies the role's type which may be any of the following:

- **initial** the role that initiates the interaction.
- **necessary** a required role, without which the interaction cannot begin.
- **auxiliary** a secondary role used only for role changing within the interaction.
- **optional** a role that may participate in the interaction, yet not necessary for the interaction.

The last two arguments are integer values that specify the minimum and maximum number of peers that can subscribe to the role. These arguments are optional while their default value is 1. Each interaction must include at least two roles with one of them being the “initial”. Furthermore there can be only one “initial” role per interaction. An additional requirement for the “initial” role is that it must be the first to actually send a message in the context of the interaction. Lastly, auxiliary roles can only be used inside the interaction through role changing, thus peers cannot directly subscribe to them.

Creating and using OpenKnowledge components was a fairly simpler process compared to the interaction models. The initial examples provided a good starting point from where I was able to create more complex components. While experimenting with OpenKnowledge components I discovered yet another undocumented point worth mentioning: Every OpenKnowledge component must have at least one argument. The documentation clearly states that every OpenKnowledge component must be a boolean Java method with all its arguments being of type “Argument”. However, it makes no reference to the minimum number of arguments required. Interaction models using OpenKnowledge components with no arguments crash by throwing a null pointer exception right after the component is called.

Although the current OpenKnowledge implementation fulfils its main purpose, the resulting system is not very programmer-friendly. The interfaces provided give little to no control over both peers and interactions. Moreover, the error messages in many cases are too general leading to confusion on the problem’s root cause. Additionally, the source code is fairly complex and its structure makes it difficult to extend.

4.3 Design approaches

This section describes the approaches followed in order to arrive at a complete and working system design. All the approaches presented here, aim towards a peer-to-peer system based on OpenKnowledge, that can automatically manage itself based on policies.

The concept of policies is a rather simple one: Initially, the user specifies each physical machine’s policy. Then, all physical machines participate as peers in the OpenKnowledge peer-to-peer network. Each peer in the OpenKnowledge network acts as an agent with its policy specifying its goals. Finally, the peers act according to their policies and exchange virtual machines until their individual goals are reached.

The approaches are presented in the order they were attempted. The first two failed, leading to the third approach which was actually implemented. Each approach first presents the main idea behind it followed by its technical requirements and concludes with the reasons behind its failure. Finally, the implementation of the third approach is described on the next section of this chapter.

4.3.1 First approach

Given the original system specification, the main design problem can be reduced to properly designing and placing the policies within the system. Once this initial point is specified, the rest of the design comes as a direct consequence of this decision.

In this first approach, the main idea was to place the policies inside the OpenKnowledge interaction models. Using this approach, the policies would be implemented as roles within the interaction model. The peers would then be able to join in interactions and take up roles according to the desired policy. By following the steps included in their role, the peers would be able to determine whether their goals have been accomplished and decide whether or not to interact with other peers. Figure 4.3 depicts this system design.

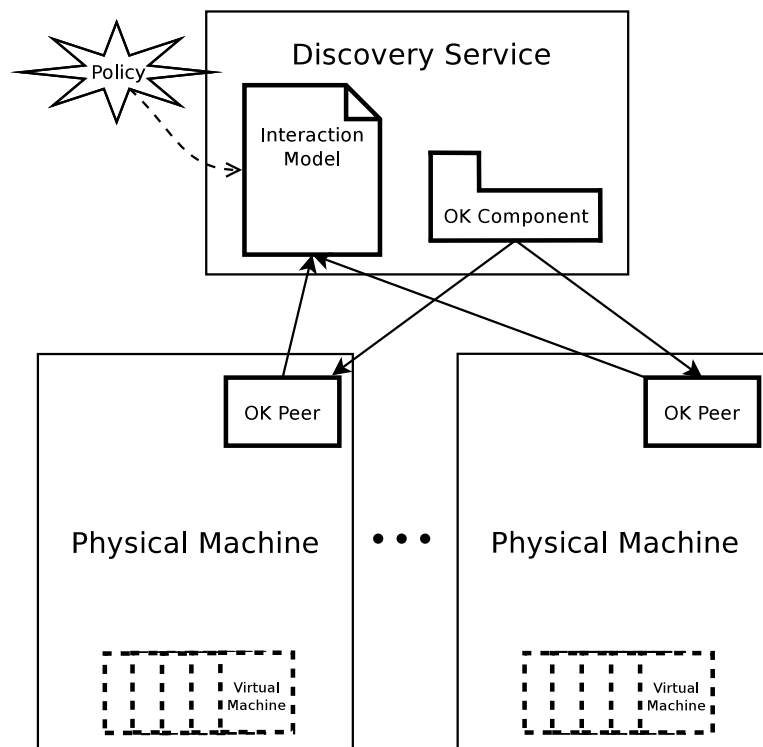


Figure 4.3: First system design

In order to actually implement this design, some technical requirements must first be addressed. The peers need to be able to get live system information about their current state (including RAM usage and CPU load) and use it within the interaction model. Moreover, virtual machine migration should also be available to use from within the interaction.

All of the technical requirements above can be satisfied through the use of OpenKnowledge components. Using OpenKnowledge components is the sole way for a peer to communicate with its outside environment while being engaged in an interaction.

At this point I had no available OpenKnowledge components, which were a necessity in order to try out example policies. Hence, the first step that also involved coding for this design,

was the creation of OpenKnowledge components. As the goal was to try out some example policies, the components required were essentially the migration component and any component that provided live system information. For the latter, I decided to implement a component that returned the amount of free system memory. Having created a minimal set of components, I was now ready to experiment with implementing policies as part of the interaction models.

As previously mentioned, the interaction models in OpenKnowledge are written in LCC. LCC is by definition a coordination calculus and its main operators are for receiving and sending messages between the roles involved. Its lightweight nature also requires that only the absolute necessary operators are included. Therefore, LCC does not support most of the operations that are common in other programming languages.

These limitations of LCC were the main reason for the failure of this approach. Without a proper set of operators, defining policies was impossible. This by no means implies that LCC was to blame for the failure, as the design failed mostly due to my own false understanding of LCC's field of application. By then, it was clear that all the computations required within an interaction should be handled by OpenKnowledge components and that LCC should only be used for defining the interaction's flow.

With this first approach failing, it became apparent that placing the policies inside OpenKnowledge interaction models was not feasible. Moreover, it indicated the need for a different design that would rely more on OpenKnowledge components to convey the core logic of the system.

4.3.2 Second approach

The failure of the previous approach, eliminated the option of placing the policies within OpenKnowledge interaction models. Hence, this second approach focused on a design in which the peers' policies would be placed inside OpenKnowledge components. The idea here was that the peers would join in an interaction and then act according to decisions made by OpenKnowledge components. Additionally, these components could be implemented in a way as to support individual policies for each peer rather than global policies. Moreover, this design would also make proper use of the LCC language as no actual computations would take place within the interaction model. Figure 4.4 depicts this system design.

Implementing this system would require designing an efficient combination of an OpenKnowledge interaction model with several OpenKnowledge components to determine the course of the interaction. In contrast to the previous design, components that provide live system information are no longer needed. With this requirements in mind, the following concept emerged.

In any given situation, the individual peers can be reduced into three categories according to their current state:

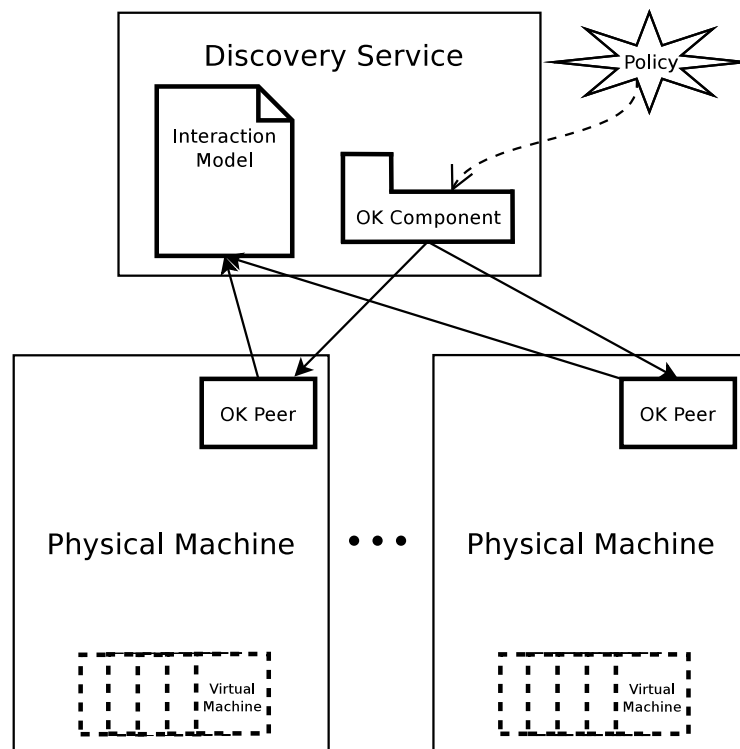


Figure 4.4: Second system design

- Peers that are happy with their current state and therefore do not want to interact.
- Peers that have high resource utilisation and therefore want to interact with other peers to pass them their excess load.
- Peers that have low resource utilisation and therefore want to interact with other peers as to take on some of their load.

Note: From this point on the peers of the first category will be referred to as “idle”, the peers of the second category as “overloaded”, and the peers of the third category as “underloaded”.

This observation led to the idea that each of these states should be represented as a role inside the interaction model. Initially, all the peers would subscribe to the “idle” role and every time their state was altered they would change their role accordingly. Each peer would be able to find out about its current state through the use of OpenKnowledge components. Finally, the “underloaded” peers would interact with the “overloaded” peers in order to exchange virtual machines. The virtual machine migration process would be handled by a separate OpenKnowledge component. Figure 4.5 shows the basic structure of this interaction.

The policy for each peer is implemented by the components `isOverloaded()` and `isUnderloaded()`. Since every OpenKnowledge component is a boolean function the peer can change roles according to the return values of those two components. Finally, if both components

```

a(idle, ID1) ::
(
    (null <- isOverloaded(ID1)
    then
    a(overloaded, ID1))
    or
    (null <- isUnderloaded(ID1)
    then
    a(underloaded, ID1))
    or
    (a(idle, ID1))
)

a(underloaded, ID2) ::
    -- Interact with overloaded peer --
    then
    a(idle, ID2)

a(overloaded, ID3) ::
    -- Interact with underloaded peer --
    then
    a(idle, ID3)

```

Figure 4.5: Second system interaction skeleton

return false, the peer remains on the initial idle role.

Experiments using this interaction model showed that this approach was also not feasible. The main reason was that the peers did not join the interaction as expected. In particular, instead of participating in the same interaction, all peers joined separate instances of the interaction thus making it impossible to interact with each other. Apparently, the problem lied within the initial specification of the interaction, which required at least one peer in order to begin. As this requirement was satisfied once a single peer joined, the interaction then initiated immediately thus preventing other peers from joining it. A possible solution to this problem would be to specify in advance the total number of peers that should participate in the interaction. This solution though would not be correct for a peer-to-peer system as the system should operate regardless of the number of peers in it.

In conclusion, this approach showed that interactions using multiple peers are not suitable

for this system. However, the ideas in this approach led in a better design that was more structured and set the standards for the final design.

4.3.3 Third approach

Although the previous approaches were unsuccessful, they provided great insight on the system and on the way it should be designed. At this point it was clear that the system's policies should not be placed neither within the interaction models nor into OpenKnowledge components. Additionally, the second designed showed that the peers need to interact only when they are not satisfied with their current state. Therefore after considering the observations made so far as well as the available options, it became apparent that the system's policies should be placed on the OpenKnowledge peers themselves. The main idea here is closely related to the one presented in the second approach, as it assumes the same roles, namely "underloaded" and "overloaded". However, in this approach the process of determining a peer's state occurs within the peer itself as opposed to using OpenKnowledge components. Figure 4.6 depicts this system design.

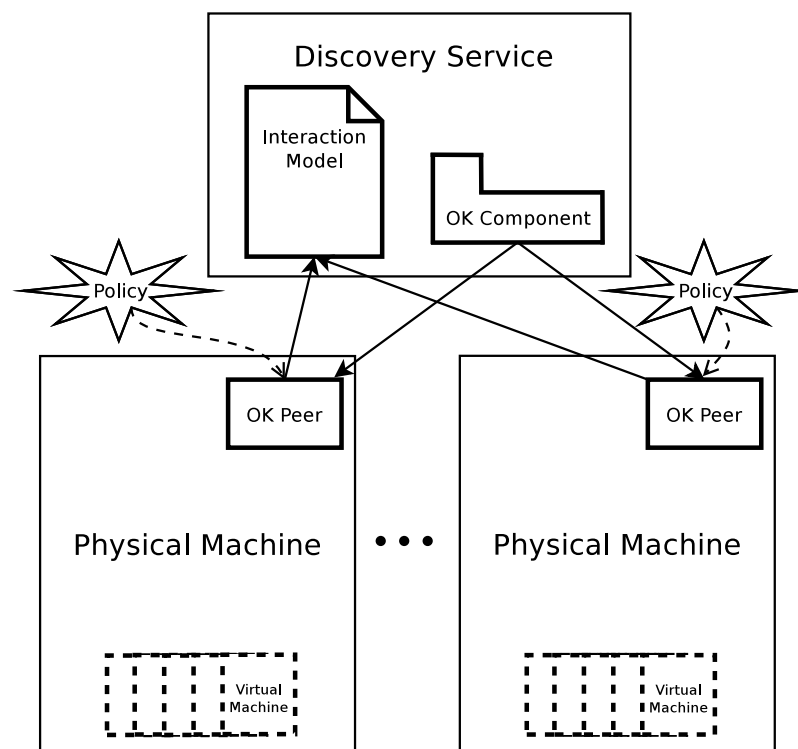


Figure 4.6: Third system design

Implementing this system would require an extended version of the OpenKnowledge peer. The extended peer should include methods that provide live system information about the underlying physical machine as well as methods for joining interactions under specific roles.

Consequently, the policies would then be implemented in Java, using the methods provided by this new version of the OpenKnowledge peer. The interaction model required for this design need not be complicated. On the contrary, it should be a simple one composed of two roles namely “underloaded” and “overloaded”. Once two peers have joined with each of them subscribing to one of the roles, the interaction should begin and result in one virtual machine migrating from the “overloaded” peer to the “underloaded” peer. As for OpenKnowledge components, the requirements in this case are minimal. Specifically, the system should include OpenKnowledge components solely for the migration of virtual machines.

This final approach proved to be a successful one. Eventually, this design was implemented as a proof of concept. The next section provides additional information regarding the system’s implementation while chapter 5 demonstrates its functionality through usage scenarios.

4.4 Implementing the system

The implementation of the system was based on the final design as described on section 4.3.3. Realising the design was done by creating the following components:

- The interaction model.
- The relevant OpenKnowledge components.
- The extended version of the OpenKnowledge peer.

The interaction model for this system was a rather simple one. It included the roles “underloaded” and “overloaded” as specified in its design. The goal of the interaction was to successfully migrate a virtual machine from the “overloaded” host to the “underloaded” host. Figure 4.7 shows the final implementation of the interaction model. Initially, the “underloaded” peer discovers its hostname and sends it to the “overloaded” peer. The “overloaded” peer then randomly selects one of its virtual machines and migrates it to the hostname received by the “underloaded” peer.

The final interaction model uses three OpenKnowledge components namely `getHostname`, `getRandomVM`, and `migrateVM`. All the components were implemented as part of a new Java class named “`libvirtOKC`”. The “`libvirtOKC`” class extended the “`OKCFacadeImpl`” class and included the OpenKnowledge components as public boolean methods. Additionally, all the arguments passed to the components were of type “`Argument`”. This was done in order to comply with the OpenKnowledge specification on components [17]. The following paragraph briefly describes the implementation of the system’s OpenKnowledge components.

The `getHostname` component is used by the “underloaded” peer in order to find out its own hostname. It takes one argument of type `Argument` and sets its value to the hostname of the

```

a(underloaded, ID1) ::
    response(Hostname) => a(overloaded, ID2) <- getHostname(Hostname)

a(overloaded, ID2) ::
    response(Hostname) <= a(underloaded, ID1)
    then
    null <- getRandomVM(VMname)
    then
    null <- migrateVM(VMname, Hostname)

```

Figure 4.7: Implementation of the interaction model

calling peer. Implementing the `getHostname` component was done by using Java's `InetAddress` class as to retrieve the hostname of the local host.

The `getRandomVM` component is used by the “overloaded” peer as to randomly select one of its hosted virtual machines for migration. Similar to the `getHostname` component, it takes one argument and sets its value to the name of one of the hosted virtual machines. In order to implement the `getRandomVM` component, the `libvirt` API was used to get the virtual machines currently hosted on the peer. Moreover, Java's an instance of Java's “Random” was used to facilitate the randomised selection process.

Lastly, the `migrateVM` component is used by the “overloaded” peer as to migrate a single virtual machine to the “underloaded” peer. It takes two arguments of type “Argument” with the first one specifying the name of the virtual machine to be migrated and the second one the hostname of the migration's destination. Its implementation uses the `libvirt` API to connect to the local instance of `libvirt` and issue the migration command.

All of the above OpenKnowledge components return a boolean value of *true* upon successful execution and a boolean value of *false* if any problems occurred that prevented them from completing their given task. In case any component returns *false* after being called, the corresponding step of the interaction will fail. Nevertheless, the system as a whole will remain operational while its peers will continue joining new interactions.

The last part of the system's implementation involved creating an extended version of the OpenKnowledge peer. The original version of the peer was extended as to include the following additional methods:

- `getCpuLoadOne()`
- `getCpuLoadFive()`

- `getCpuLoadFifteen()`
- `getSystemActualFreeMemory()`
- `getSystemActualUsedMemory()`
- `getTotalVMMemory()`
- `getLocalVMCount()`
- `doOverloaded()`
- `doUnderloaded()`

The methods presented above can be divided into two groups according to their functionality. The first group is composed of the first seven methods which provide live information about the peer's current state. Likewise, the second group is composed of the last two methods which make the peer join an interaction while assuming a specific role. The following paragraph gives a description of each method's functionality as well as details about its implementation.

In order to enable the peer to get live information about its CPU resource usage the extended version provides three methods, namely *getCpuLoadOne*, *getCpuLoadFive*, and *getCpuLoadFifteen*. These three methods return the average system load for the past one, five, and fifteen minutes respectively. Using the average system load [20] as a metric was more preferable than using the current CPU load as the former gives a better view of the system's given state. As far as implementation goes, all three methods obtain the required load information from the unix *proc* filesystem [22].

Access to live system memory information is made available through the *getSystemActualFreeMemory* and *getSystemActualUsedMemory* methods. These methods return the real amount of free and used system memory respectively. The amount of memory is returned in kilobytes and does not include memory kept in buffers. Both methods use the Sigar [15] system information API in order to obtain the desired memory amounts.

The methods *getTotalVMMemory* and *getLocalVMCount* aim at providing virtual machine related system information. Although memory usage information was already available from the previous methods, it did not distinguish between system usage and virtual machine usage. Therefore, the *getTotalVMMemory* method was introduced as to make this distinction possible by returning the total memory used by the system's virtual machines. The *getLocalVMCount* method provides yet another virtual machine related metric as it returns the total number of virtual machines hosted locally. Both methods make use of the libvirt API in order to obtain the required information.

The methods described so far focused on providing live system information, as to enable the peer to determine its current state. However, in order to apply its policy, a peer also needs

the ability to act. In this implementation the peers actions are realised by the *doOverloaded* and *doUnderloaded* methods. These methods enable the peer to join an interaction by subscribing to the relevant role and thus to act according to its state. The two methods handle the subscription to the roles “overloaded” and “underloaded” respectively. Their current implementation involves loading the necessary OpenKnowledge components as well as getting the relevant interaction model and informing the OpenKnowledge Discovery service about their role subscription. Moreover, it sets a timeout on the interaction process as to prevent the peer from waiting indefinitely in case no other peers are available to interact with.

The components described above compose the final implementation of the system. The next few paragraphs describe how actual peer policies can be implemented using this system.

4.4.1 Implementing policies

The initial goal of this system was to provide a way to manage a set of physical machines by applying individual policies to each of them. The current implementation of the system offers a basic set of tools which can be used to define these policies. This section presents a basic pattern for implementing policies as part of an OpenKnowledge peer.

In order to include a policy within an OpenKnowledge peer, the policy needs to be defined as a simple Java program. The extended version of the OpenKnowledge peer provides a variety of additional functions that aim at facilitating this process. Additionally the programming pattern presented here, will provide a simple, yet flexible way that can be used to define a large number of policies.

The programming pattern is composed by two main stages. During the first stage the peer computes its current state and based on that information, decides whether its “overloaded” or “underloaded”. Thus, the peer’s policy is coded as part of this first stage. In case the peer is happy with its current state, the first stage is repeated, otherwise the peer proceeds to the second stage. During the second stage, the peer engages in an interaction in order to improve its current state. The role assumed for this interaction stage is the one computed during the previous stage. Once the interaction stage is finished, the peer returns to the first stage and the process is repeated. Figure 4.8 depicts this pattern using a flowchart.

To compute a peer’s current state, the system information provided by the extended version of the OpenKnowledge peer can be combined with specific conditions. Once the state is determined, the two additional conditions showed in the flowchart (Figure 4.8) will determine the appropriate action for the peer. Then, the peer can act accordingly by calling either the *doOverloaded* or the *doUnderloaded* methods of the extended OpenKnowledge peer. These last two calls are represented in the flowchart as the “Interact as overloaded” and “Interact as underloaded” operations. Finally, all the operations must be nested in a infinite loop, optionally accompanied by a wait/sleep statement. The infinite loop is required as to repeat the two stages

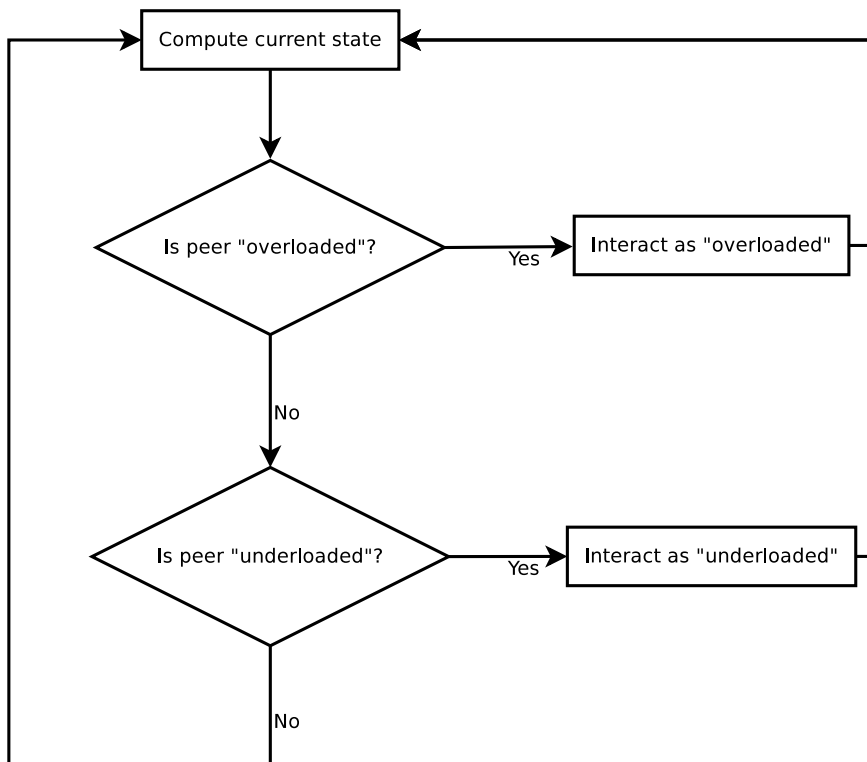


Figure 4.8: Programming pattern for policies

while the wait/sleep statement will prevent the program from wasting CPU time. The usage scenarios presented in the next chapter include Java policies implemented using this pattern. The Java code for these policies can be found in the appendix.

Chapter 5

Usage Scenarios

This chapter demonstrates the functionality of the final implementation through sample usage scenarios. All the scenarios were executed on the four servers of the HP cluster using a set of virtual machines running the latest version of Ubuntu [16] Linux (11.04). The virtual machines follow a specific naming scheme beginning with “okvm” and followed by a single digit (for example “okvm3”). Before executing each scenario, the virtual machines were reconfigured as to fit the scenario’s needs. For the purposes of this demonstration I also created a set of policies that use the main functions provided by the implementation. These policies were then executed on the individual physical machines of the cluster as to provide the core logic for each peer. The code for the policies used in the following scenarios can be found in the appendix.

At this point I would like to stress that these scenarios merely demonstrate the functionality of the implementation. Therefore, the policies used should not in any case be considered optimal.

5.1 Scenario 1

This scenario demonstrates how the implementation can be used for load-balancing. The policy used in this case is solely based on system load and it is the same for all four peers involved. When a peer’s system load is less than 0.50 it considers itself “underloaded”. Correspondingly, when its system load is over 1.50 it considers itself “overloaded”.

The scenario begins with four virtual machines (okvm[1-4]) that are initially located on hp1. Then, each virtual machine is forced to a system load of 1.00 by executing the following command: `while true; do true; done`. Figure 5.1 shows the initial distribution of the virtual machines.

Adding load to each individual virtual machine brings up the system load on hp1 to a total of 4.00. From that point on, hp1 considers itself “overloaded” and will try to migrate its virtual

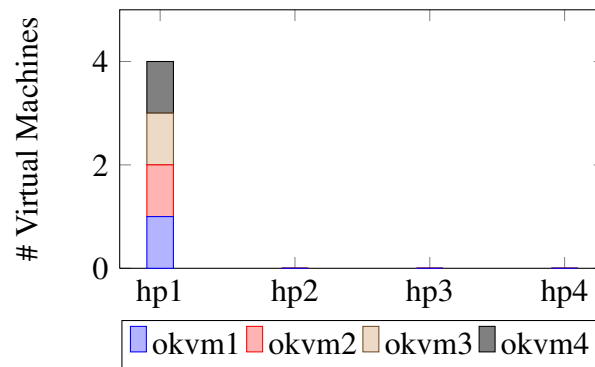


Figure 5.1: Initial Virtual Machine distribution.

machines to other hosts until its load falls below 1.50. Similarly, hp[2-4] consider themselves “underloaded” as their current load is minimal (under 0.50). Therefore, they will try to acquire virtual machines until their load is above 0.50.

Eventually, all peers should be satisfied with their current status (neither “overloaded” nor “underloaded”) with each one hosting a single virtual machine. Figure 5.2 shows the resulting virtual machine distribution, after leaving the peers to interact.

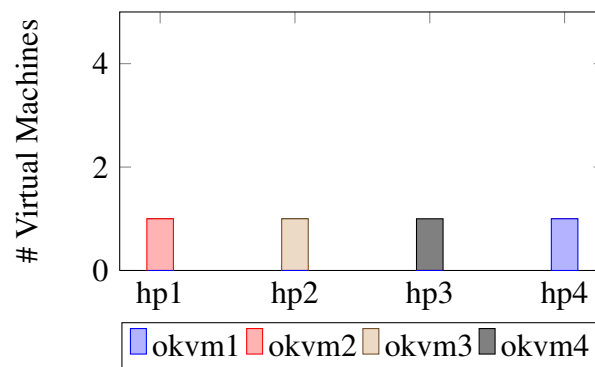


Figure 5.2: Resulting Virtual Machine distribution.

5.2 Scenario 2

This scenario also demonstrates how the implementation can be used for load-balancing, though this time the policies are based on memory load. All peers have the same policy. A peer considers itself “underloaded” when its memory load is below 512 MB and “overloaded” when over 768 MB. For the purpose of this scenario four identical virtual machines were used (okvm[1-4]), each of them having 512 MB of memory.

Initially, hp2 and hp3 host two virtual machines each, while hp1 and hp4 host none. Figure 5.3 shows the initial virtual machine distribution for this scenario.

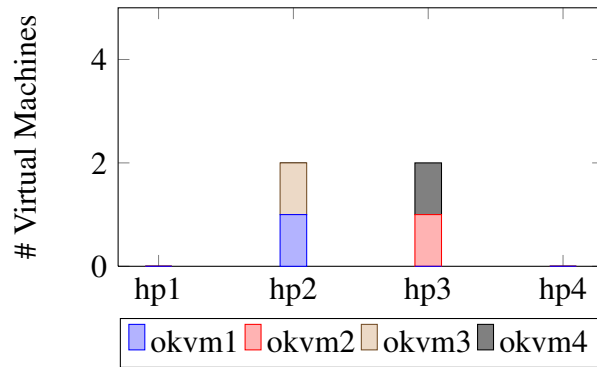


Figure 5.3: Initial Virtual Machine distribution.

With two virtual machines each, hp2 and hp3 have a total memory load of 1024 MB. According to their policies, they consider themselves “overloaded” ($1024 > 768$) and will thus try to migrate the excess virtual machines to some other host. Similarly, hp1 and hp4 are initially “underloaded” as their memory load is 0 MB (< 512) and will thus try to acquire virtual machines.

Using this policy, the memory load should eventually spread across all four physical machines. Finally, all physical machines should host only one virtual machine, giving each a total memory load of 512 MB. Figure 5.4 shows the resulting virtual machine distribution after all peers have finished interacting.

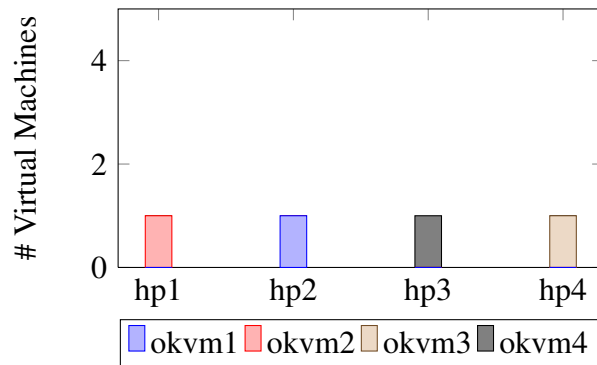


Figure 5.4: Resulting Virtual Machine distribution.

5.3 Scenario 3

This scenario demonstrates how the implementation can be used for shutting down physical machines. In order to shut down gracefully, a physical machine should first migrate all its hosted virtual machines elsewhere, as to avoid interrupting their execution. Hence, the policies used in this scenario are based on the local virtual machine count for each peer. Specifically,

hp1 is the peer that wants to shut down and thus considers itself “overloaded” whenever its local virtual machine count is greater than 0. All other peers aim at hosting exactly three virtual machines and are either “overloaded” when they host more or “underloaded” when they host less.

Initially, hp1 (which wants to shut down) hosts three virtual machines while all other peers host two each. Figure 5.5 shows the initial virtual machine distribution for this scenario.

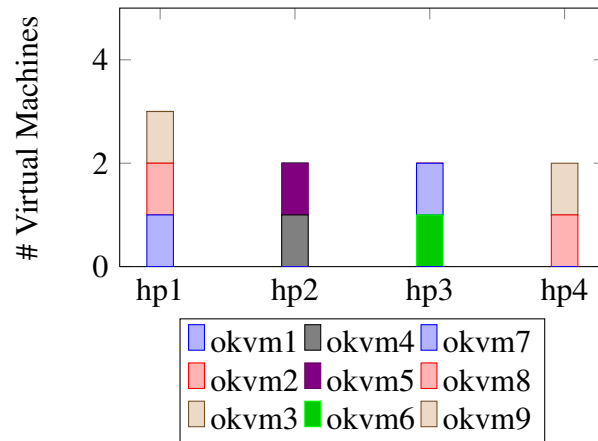


Figure 5.5: Initial Virtual Machine distribution.

Hosting a total of three virtual machines makes hp1 “overloaded”. Consequently, it will have to migrate all of its virtual machines to other hosts as to achieve its goal. Moreover, the rest of the hosts (hp[2-4]) are in an “underloaded” state as they each require a third virtual machine in order to reach their desired state.

Eventually, hp1’s virtual machines should be migrated to the other three hosts, leaving the system in a balanced — according to the policies — state. Figure 5.6 shows the resulting virtual machine distribution after all interactions have finished.

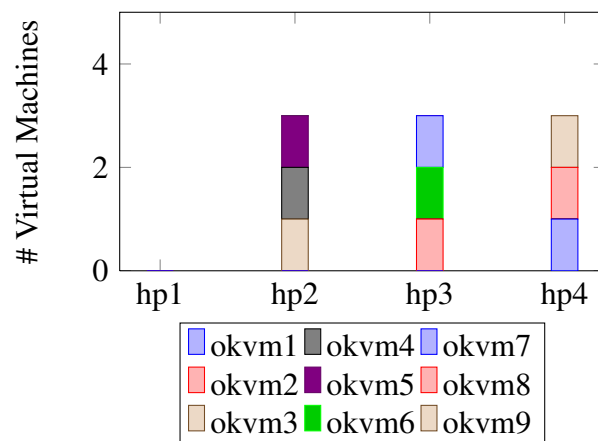


Figure 5.6: Resulting Virtual Machine distribution.

5.4 Scenario 4

This last scenario demonstrates how the implementation can be used for more complicated tasks. It uses a combination of the previous policies as to give each peer a different goal. In particular, the scenario involves two peers that wish to shut down (hp1 and hp4), one peer that wants high CPU load (hp2) and one peer that wants high memory load (hp3). The policies for each peer are the following:

- hp1 and hp4 are “overloaded” when their local virtual machine count is greater than zero.
- hp2 is “underloaded” when its CPU load is less than 1.50 and “overloaded” when its memory load is greater than 1 GB.
- hp3 is “underloaded” when its memory load is less than 2 GB and “overloaded” when its CPU load is more than 0.50.

The four virtual machines used in this scenario are specifically configured as to suit the scenario’s needs. Two of the virtual machines (okvm[1-2]) are configured as to have low memory requirements (512 MB each) while the other two (okvm[3-4]) require a larger amount of memory (1.5 GB each). Moreover, okvm1 and okvm2 are forced to a system load of 1.00 by executing the following command on each of them: `while true; do true; done`. This results in two virtual machines with high CPU load and two virtual machines with high memory load.

Initially, the two peers that wish to shut down (hp1 and hp4) host two virtual machines each, one with high CPU load and one with high memory load. The rest of the peers host no virtual machines. Figure 5.7 shows the initial virtual machine distribution for this scenario.

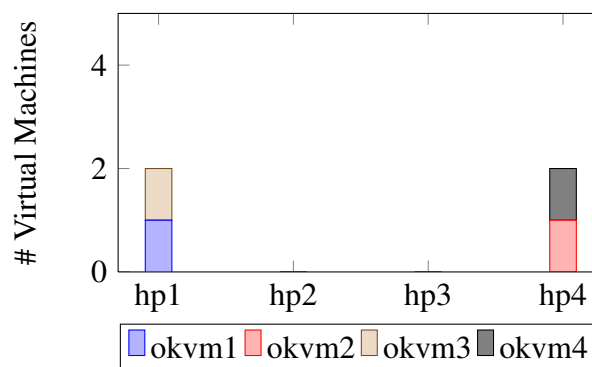


Figure 5.7: Initial Virtual Machine distribution.

By hosting two virtual machines each, hp1 and hp4 consider themselves “overloaded”, as their goal is to end up with no virtual machines at all. At the same time hp2 and hp3 are both

“underloaded” due to having low CPU load and low memory load respectively. Consequently, there is both offer and demand for virtual machines in the system.

After a series of migrations, the virtual machines having high CPU load should end up on hp2, with the ones having high memory load ending up at hp3. Moreover, hp1 and hp4 should eventually host no virtual machines at all, leaving the system in harmony (always according to the specified policies). Figure 5.8 shows the resulting virtual machine distribution after all interactions have finished.

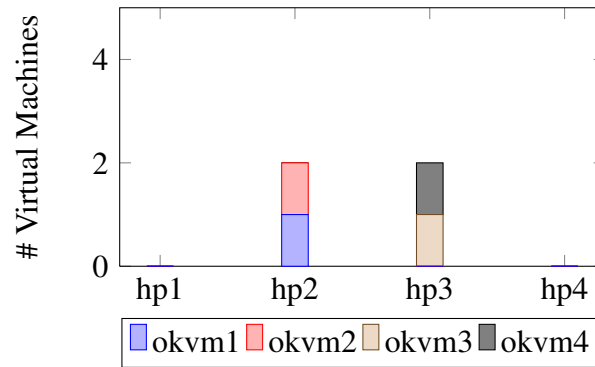


Figure 5.8: Resulting Virtual Machine distribution.

Chapter 6

Conclusions

6.1 Ideas for future work

Up until this point, this document has presented the design and implementation of an automated peer-to-peer virtual machine management system. Although the current implementation provides a working solution for the virtual machine management problem, it only offers basic functionality. This section presents some ideas that can further improve its functionality as well as the overall virtual machine management process.

One aspect of the system that can be improved is the way the virtual machines are selected for migration. Currently, the system operates under the naive assumption that all virtual machines contribute equally to a physical machine's load. Hence, whenever a peer needs to migrate a virtual machine, it randomly chooses one from its hosted virtual machines. However, in reality virtual machines have unique specifications and resource requirements. In order to improve the virtual machine selection process, every choice has to be evaluated in advance as to measure its impact on the peer's current state. Doing so would require the peer to simulate its resulting state after migrating a virtual machine and then make the best choice according to its policy.

Another concept that could benefit the system in general is that of virtual machine trading. As previously mentioned, the peers in the system engage in interactions only when they are not satisfied with their current state. Moreover, the roles currently used in the system's interaction models are too strict, as they only express a peer's need to take or give away a virtual machine. There are situations though in which this approach does not work. For example, consider a scenario with a total of three peers, two of them being satisfied with their current status and one of them being "overloaded". In this case, none of the peers will interact with the "overloaded" peer and thus it will not be able to improve its current status. However, there might exist a different virtual machine distribution that solves this problem. With the current system this

possibility is not explored at all. A possible solution would be to design a new set of roles and interactions in which peers trade virtual machines in order to improve both their status and the status of the other participants. Using this trading mechanism would enable the system to explore a greater number of possible virtual machine distributions and thus better its overall state.

In addition to the ideas presented above, the system would also benefit from being able to dynamically adapt its policies. The need for such a feature derives from the fact that the overall state of a real system (e.g. a large data centre) changes constantly. Thus, the original policies may have to be adapted as to better reflect new system states. However, adapting the policies requires a global view of the system, and since this is a peer-to-peer system, there is no central node that could carry out this task. Instead, the policies would have to be adapted by the peers themselves. In order to do so, each peer needs to have at least a partial view of the overall system state. This would require every peer to periodically query other peers and learn about their status, thus constructing a database of local knowledge. Using this information, each peer can then adapt its policy. The results however, will not be as accurate as those of a centralised system due to the peers' incomplete system state information.

6.2 Conclusion

This project explored the possibility of a peer-to-peer virtual machine management system based on OpenKnowledge. Through experimentation the project managed to arrive at a good system design that was in turn implemented into a working solution. The working implementation provided proof that the virtual machine management problem can indeed be solved using a decentralised peer-to-peer approach as an alternative to the centralised approaches used so far.

However, more work remains to be done for this approach to evolve to the point where it can replace the existing management systems. In particular, the current functionality needs to be extended further in order to be able to express more sophisticated policies and interactions between peers. Furthermore, the tools used in the implementation need to be reconsidered as to better reflect the system's requirements.

Implementing the system on top of the OpenKnowledge peer-to-peer service proved rather unsuitable for this project. Although OpenKnowledge presents a great concept, most of its features were not exploited. OpenKnowledge is more directed towards complex interactions where peers have no prior knowledge of neither each other or the details of the interaction. Thus, its functionality focuses on orchestrating otherwise incompatible peers and enabling them to collaborate as to accomplish tasks. However, in this project's case the interactions were relatively simple, all the peers were compatible with each other, and the details of the interactions in-

volved were known in advance. Moreover, the current implementation of OpenKnowledge is rather unstable and provides little control over its internal procedures. Hence, this system could have been implemented using a more basic peer-to-peer solution.

In conclusion, although the current implementation does not suffice for commercial use, this project can be considered successful. Its final result proves that a decentralised agent-based approach is feasible and suggests that further research can be beneficial for this kind of approaches. But more importantly, it points out certain pitfalls that should be avoided in any relevant future work.

Appendix A

Policy source code

A.1 Scenario1Peer.java

```
public class Scenario1Peer {
public static void main (String args[]){
    PhysicalMachinePeer peer = new PhysicalMachinePeer();
    peer.init(args);

    int cpuLoad = 0;
    while (true) {
        try {
            Thread.sleep(60 * 1000); // check status every 60 seconds.
        } catch (InterruptedException e) {
            continue;
        }
        try {
            cpuLoad = peer.getCpuLoadOne();
        } catch (Exception e) {
            System.err.println(e.getMessage());
            continue;
        }
        if (cpuLoad > 150) { // load over 1.50
            System.out.println("Status: Overloaded");
            peer.doOverloaded();
        } else if (cpuLoad < 50) { // load under 0.50
```

```
        System.out.println("Status: Underloaded");
        peer.doUnderloaded();
    } else {
        System.out.println("Status: OK");
    }
}
}
}
}
```

A.2 Scenario2Peer.java

```
import org.libvirt.LibvirtException;

public class Scenario2Peer {
    public static void main(String args[]){
        PhysicalMachinePeer peer = new PhysicalMachinePeer();
        peer.init(args);

        long totalMem = 0;
        while(true){
            try {
                Thread.sleep(60 * 1000); // check status every 60 seconds.
            } catch (InterruptedException e) {
                continue;
            }

            try {
                totalMem = peer.getTotalVMMemory();
            } catch (Exception e) {
                System.err.println(e.getMessage());
                continue;
            }

            if (totalMem < 512 * 1024){ // memory used for VMs < 512 MB
                System.out.println("Status: Underloaded");
            }
        }
    }
}
```

```
        peer.doUnderloaded();

    }

    else if (totalMem > 768 * 1024){ // memory used for VMs > 768 MB
        System.out.println("Status: Overloaded");
        peer.doOverloaded();

    }
    else{
        System.out.println("Status: OK");
    }

}
}
}
```

A.3 Scenario3Peer.java

```
public class Scenario3Peer {
    public static void main(String args[]){
        PhysicalMachinePeer peer = new PhysicalMachinePeer();
        peer.init(args);

        int vMCount = 0;

        while(true){
            try {
                Thread.sleep(60 * 1000); // check status every 60 seconds.
            } catch (InterruptedException e) {
                continue;
            }

            try {
                vMCount = peer.getLocalVMCount();
            } catch (Exception e) {
```

```
        System.err.println(e.getMessage());
        continue;
    }

    if (vMCount < 3){
        System.out.println("Status: Underloaded");
        peer.doUnderloaded();
    }
    else if (vMCount > 3){
        System.out.println("Status: Overloaded");
        peer.doOverloaded();
    }
    else{
        System.out.println("Status: OK");
    }
}
}
```

A.4 Scenario3ShutdownPeer.java

```
public class Scenario3ShutdownPeer {
    public static void main(String args[]){
        PhysicalMachinePeer peer = new PhysicalMachinePeer();
        peer.init(args);

        int vMCount = 0;

        while(true){
            try {
                Thread.sleep(60 * 1000); // check status every 60 seconds.
            } catch (InterruptedException e) {
                continue;
            }

            try {
```

```
        vMCount = peer.getLocalVMCount();
    } catch (Exception e) {
        System.err.println(e.getMessage());
        continue;
    }

    if (vMCount > 0){
        System.out.println("Status: Overloaded");
        peer.doOverloaded();
    }
    else{
        System.out.println("Status: OK");
    }
}
}
```

A.5 Scenario4HighLoadPeer.java

```
public class Scenario4HighLoadPeer {
    public static void main(String args[]){
        PhysicalMachinePeer peer = new PhysicalMachinePeer();
        peer.init(args);

        int cpuLoad = 0;
        long totalMem = 0;

        while(true){
            try {
                Thread.sleep(60 * 1000); // check status every 60 seconds.
            } catch (InterruptedException e) {
                continue;
            }

            try {
```

```
        cpuLoad = peer.getCpuLoadOne();
        totalMem = peer.getTotalVMMemory();
    } catch (Exception e) {
        System.err.println(e.getMessage());
        continue;
    }

    if (totalMem > 1024 * 1024){
        System.out.println("Status: Overloaded");
        peer.doOverloaded();
        continue;
    }

    else if (cpuLoad < 150){
        System.out.println("Status: Underloaded");
        peer.doUnderloaded();
        continue;
    }

    else{
        System.out.println("Status: OK");
    }
}
}
```

A.6 Scenario4HighMemoryPeer.java

```
public class Scenario4HighMemoryPeer {
    public static void main(String args[]){
        PhysicalMachinePeer peer = new PhysicalMachinePeer();
        peer.init(args);

        int cpuLoad = 0;
        long totalMem = 0;
```

```
while(true){
    try {
        Thread.sleep(60 * 1000); // check status every 60 seconds.
    } catch (InterruptedException e) {
        continue;
    }

    try {
        cpuLoad = peer.getCpuLoadOne();
        totalMem = peer.getTotalVMMemory();
    } catch (Exception e) {
        System.err.println(e.getMessage());
        continue;
    }

    if (totalMem < 2 * 1024 * 1024){
        System.out.println("Status: Underloaded");
        peer.doUnderloaded();
        continue;
    }

    else if (cpuLoad > 50){
        System.out.println("Status: Overloaded");
        peer.doOverloaded();
        continue;
    }

    else{
        System.out.println("Status: OK");
    }
}
}
```

A.7 Scenario4HighShutdownPeer.java

```
public class Scenario4ShutdownPeer {
    public static void main(String args[]){
        PhysicalMachinePeer peer = new PhysicalMachinePeer();
        peer.init(args);

        int vMCount = 0;

        while(true){
            try {
                Thread.sleep(60 * 1000); // check status every 60 seconds.
            } catch (InterruptedException e) {
                continue;
            }

            try {
                vMCount = peer.getLocalVMCount();
            } catch (Exception e) {
                System.err.println(e.getMessage());
                continue;
            }

            if (vMCount > 0){
                System.out.println("Status: Overloaded");
                peer.doOverloaded();
            }
            else{
                System.out.println("Status: OK");
            }
        }
    }
}
```

Bibliography

- [1] <http://www.linux-kvm.org/page/>.
- [2] <http://libvirt.org/>.
- [3] <http://www.openk.org/>.
- [4] <http://xen.org/>.
- [5] http://www.citrix.com/English/ps2/products/product.asp?contentID=683148&ntref=prod_cat.
- [6] <http://www.vmware.com/products/vcenter-server/overview.html>.
- [7] <http://virt-manager.org/>.
- [8] http://www.toutvirtual.com/products/viq_525.php.
- [9] <http://www.hp.com/>.
- [10] <http://www.scientificlinux.org/>.
- [11] <http://nfs.sourceforge.net/>.
- [12] <http://www.redhat.com/rhel/>.
- [13] <http://yum.baseurl.org/>.
- [14] <http://jna.java.net/>.
- [15] <http://www.hyperic.com/products/sigar>.
- [16] <http://www.ubuntu.com/>.
- [17] **Openknowledge manual.** <http://www.cisa.inf.ed.ac.uk/OK/download/manual.pdf>.
- [18] Paul Anderson. The complete guide to lcfg. Internal Document, 2001.

- [19] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, April 2010.
- [20] Neil J. Gunther. Linux load average revealed. In *Int. CMG Conference'04*, pages 149–160, 2004.
- [21] N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296, 2000.
- [22] M. Tim Jones. Access the linux kernel using the /proc filesystem. <http://www.ibm.com/developerworks/library/l-proc/index.html>.
- [23] David Robertson. A lightweight coordination calculus for agent systems. In Joo Leite, Andrea Omicini, Paolo Torroni, and Pnar Yolum, editors, *Declarative Agent Languages and Technologies II*, volume 3476 of *Lecture Notes in Computer Science*, pages 109–115. Springer Berlin / Heidelberg, 2005.
- [24] P. Ruth, Junghwan Rhee, Dongyan Xu, R. Kennell, and S. Goasguen. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. pages 5 – 14, 2006.
- [25] Markus Schmid, Dan Marinescu, and Reinhold Kroeger. A framework for autonomic performance management of virtual machine-based services, 2008.
- [26] Ronny Siebes, Dave Dupplaw, Spyros Kotoulas, Adrian Perreau De Pinninck, Frank Van Harmelen, and David Robertson. The openknowledge system: an interaction-centered approach to knowledge sharing. In *Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I, OTM'07*, pages 381–390, Berlin, Heidelberg, 2007. Springer-Verlag.
- [27] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th Usenix Symposium on Networked Systems Design and Implementation*. Usenix, April 2007.