2009

Dynamic Resource Management in Virtualized Environments

Gaston Keller

Dynamic Resource Management in Virtualized Environments

(Spine Title: Dynamic Resource Management in Virtualized Environments)

(Thesis Format: Monograph)

by

Gastón Keller

Graduate Program in Computer Science

Submitted in Partial Fulfillment of the Requirements
for the Degree of
Master of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada
August 21, 2009

Abstract

System Virtualization has become in the last few years an essential technology in the data center. Among other benefits, virtualization improves resource utilization through server consolidation, and provides for highly customizable computing environments. However, virtualization also makes resource management more complex.

Golondrina, an autonomic resource management system, was built to use virtual machine migration and replication to handle resource stress situations (when resource demand is greater than resource availability).

Preliminary experiments show that replication offers improvements over migration, and both mechanisms offer improvements over taking no action upon a resource stress situation.

This work is one of the first ones in proposing a resource management system for operating system-level virtualized environments. Moreover, this is the first study that uses replication as an alternative to migration and compares both mechanisms.

Keywords: virtualization, resource management, replication, migration, autonomic computing.

For those who came before and

those who will come after.

Acknowledgements

First and foremost I would like to thank my advisor, Prof. Hanan Lutfiyya, for her guidance and dedication. She encouraged me to pursue research that I really care about, and had patience while I did many, many things beyond research. She provided support for every project or idea I came up with (again, many times not directly related to research), and had time for frequent meetings and for reading once and again this manuscript.

I would also like to thank Magi and Jim, from the Systems Group, for helping with the infrastructure.

My friend Lisandro D. N. Pérez Meyer back in Argentina, with whom I had many discussions about GNU/Linux, virtualization and networking, also deserves my gratitude.

I should not forget my fellow DiGS members, who discussed with me about my project, provided me with feedback and sometimes even directions. Thanks to them.

A very special thank goes to my unofficial *mentor*, Nadine LeGros. I would probably have not achieved all I have done without her encouragement and support.

And last but not least, many thanks to my friends here in London, who beared with me (and my shrinking patience) during my *hibernation mode* (period in which I wrote the thesis), and *mi gente* (my people) back in Argentina, who have always been *there*.

"All the theses are ethereal... until materialized by deadlines." —Me

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Autonomic resource stress release in virtualized data centers through migration and replication is possible and results in better resource utilization and improved service for the client.

System Virtualization is a software technique that enables the simultaneous execution of multiple computer systems in one physical machine. Virtualization is highly supported by industry, which sees virtualization as an opportunity to make better use of computing resources. It is now possible to buy a single physical machine and use it to run many computer systems.

Section 1.1 introduces the motivation for this research. Section 1.2 describes the research problem. Section 1.3 introduces the proposed solution.

1.1 Motivation

Today data centers tend to over-provision in order to cope with demand. Virtualization can potentially benefit data centers reducing the need for over-provisioning and hence the costs.

1.1.1 Pre-virtualization Data Centers

A data center is a collection of computers connected through a high-speed network. The number of computers in a data center can vary from several dozen machines (very modest) to hundreds of thousands of machines. Data centers are often used

to provide services to clients through the Internet. Different clients often require a varying amount of resources.

A challenge with data centers is resource provisioning. The demand for resources varies over time. The challenge faced by data centers is the tradeoff between resource utilization and being able to cope with demand. There are two resource provisioning approaches. The first one is *under-provisioning* (or provisioning for less than the peak demand). The resource investment is low to moderate, at the expense of not being able to support peaks in demand. The failure to satisfy those peaks in demand translates into poor service provided to the client, and hence this approach is rarely taken.

The second approach to provisioning is to provide resources for the peak demand (or *over-provisioning*). It guarantees (under the assumption that the peak demand is known beforehand) that all workloads will be successfully handled. However, this solution is expensive since huge numbers of resources might be needed to support peaks in demand that seldom occur (which also results in low resource utilization levels). This is usually the approach taken by companies.

Thus, the scenario in most data centers (if not all) is that a large number of resources is idle while waiting for those rare and brief moments when a peak in demand makes use of the resources.

1.1.2 Post-virtualization Data Centers

Data centers tend to run a large number of physical servers with a low resource utilization level [12, 26]. Virtualization could enable server consolidation, increasing the resource utilization level and decreasing the necessary investment in equipment.

Server consolidation could result in effectively reducing over-provisioning, but at the expense of increasing resource management complexity. Now there is not only one computer system running on the physical server, but several. It is necessary to monitor each virtual machine's workload and dynamically adjust resource allocation on-demand. Different virtual machines may run applications that have competing

Quality of Service requirements that may not be satisfiable at a given point in time.

Thus, for server consolidation to be effective, improved resource management mechanisms need to be devised. Mechanisms are needed that can automatically respond in a timely fashion to the unpredictable, time-varying demand experienced by the virtual machines running in a physical server and for the hundreds of servers that could be running together in a data center.

1.2 Problem Statement

The use of virtual machines in data centers for consolidation purposes is based on an assumption about the behaviour of the virtual machines (or more precisely, of the hosted applications). The assumption is that it is not likely that two or more virtual machines hosted in the same physical server will experience a peak in demand at the same time. However, if that situation were to happen, it would only be acceptable provided that the sum of the virtual machines' resource needs did not exceed the amount of available resources. Otherwise, the virtual machines would receive less resources than required. We call this scenario a *resource stress situation.*

When the combined resource needs of the virtual machines exceeds the total available resources, two approaches can be taken. The first one reallocates resources among the virtual machines based on a given priority. This method would be similar to the concept of *differentiated services* in networks.

Although an acceptable approach when available resources in the data center are limited or completely exhausted, priority-based resource allocation transfers the problem from one virtual machine to another, enabling privileged virtual machines to keep working at the expense of unprivileged virtual machines that cannot do so due to a lack of resources. This approach contradicts the philosophy of the *pay-as-you-go* model.

The second approach that can be taken is to rely on another physical server with enough spare resources to instantiate one or more virtual machines, so as to cope with the local resource needs. This approach is called *relocation.* There are two variants to

this method: *migration* and *replication*. Migration consists of transferring a virtual machine running in the local physical server to the target physical server (the one with spare resources). An effect of this variant is that the resources allocated to the virtual machine in the local physical server are freed.

Replication entails the creation of a local virtual machine's replica in the target physical server. The load would be shared between the two instances, diminishing the stress on the local physical server.

For this model to work, i.e., for a physical server to run several virtual machines and react to a resource stress situation by relocating virtual machines, several mechanisms are needed. First, a mechanism is needed to monitor the resource usage of each virtual machine and physical server. Second, the monitored data needs to be processed and analyzed. Third, upon discovery of a resource stress situation on a physical server, a decision making process has to be invoked to determine the migrations or replications (if any) that could dissipate the resource stress situation.

Little work has been done to develop mechanisms for decision making support. Mechanisms that provide information upon which to make decisions and mechanisms that implement actions to be taken in response to resource stress situations are needed. Of special interest is support for replication. Most work in dynamic resource provisioning in data centers does not use replication. Without having decision making support mechanisms in place, effective strategies for resource management cannot be studied.

1.3 Thesis Focus

There is a need for a resource management system for virtualized environments that incorporates decision making support mechanisms and strategies to automatically deal with resource stress situations.

The system supports three tasks. The first task is the gathering and processing of resource usage statistics from every virtual machine and physical server. The second task is the analysis of the collected statistics searching for potential resource stress

situations. The third task is used upon detection of a resource stress situation: finding a sequence of relocations (migrations or replications) that could dissipate the stress situation.

This functionality is implemented in *Golondrina*, a resource management system based on a client-server architecture [30]. A client instance runs in a virtual machine in each physical node and periodically gathers resource usage statistics, that are sent to the server. In addition, the client executes migrations and replications, based on instructions from the server.

The server runs in a non-virtualized physical server and continually receives and processes clients' statistics. Periodically, it analyzes the collected data, looking for potential resource stress situations. If such a situation is detected, the server tries to find a sequence of relocations that could solve the problem, and then contacts the corresponding clients for them to execute the relocations.

This thesis is one of the first works in proposing a resource management system for operating system-level virtualized environments. Moreover, this is the first study that uses replication as an alternative to migration and compares both mechanisms.

1.4 Outline

This chapter provided the motivation for doing research in the area of resource management in virtualized environments (data centers specifically). The rest of the thesis is organized as follows: Chapter 2 provides the background for this research, Chapter 3 present the literature review, Chapter 4 describes the architecture of the resource management system Golondrina, Chapter 5 describes how the system was implemented, Chapter 6 presents the results of the system's evaluation, and Chapter 7 concludes with a discussion and future work. An appendix is included at the end of the thesis with code snippets (from Golondrina) that are referenced in Chapter 5.

Chapter 2

Background

This chapter presents basic information required to provide a context for the chapters to come. Section 2.1 deals with the concept of virtualization and the architectural approaches that can be taken to provide virtualization. Section 2.2 provides details on the virtualization solution used in this research.

2.1 Virtualization

Resource virtualization consists of creating abstractions of physical resources. This technique enables the multiplexing of physical resources. In this context, the virtualization mechanism's role is to map virtual resources into physical resources. It is possible to build multiple abstractions of a physical machine so that different computing systems can execute simultaneously in the same physical machine.

Subsection 2.1.1 briefly discusses the basic concepts related to virtualization and Subsection 2.1.2 presents two different ways in which virtualization can be provided.

2.1.1 System Virtualization

There are two different kinds of virtualization[1]: *process virtualization* and *system virtualization.* Process virtualization exists in the form of an application whose purpose is to support a process's execution. The second one entails the creation of a proper

[1]For more information on the different types of virtualization see the paper from Smith and Nair [29].

environment for the execution of an operating system (and its processes). Our focus is on system virtualization.

System virtualization[2] is a software technique that enables the simultaneous execution of multiple operating systems in one physical machine. This technique was created in the mid 1960s, when a team at IBM's Cambridge Scientific Center in Massachusetts designed CP-40/CMS. The objective of the system was to provide for each user a virtual machine that was indistinguishable from a physical one by a user program [19]. This idea was then abandoned, but it was brought up again in recent years. In 1998, the company VMware presented its first virtualization solution [1]. It was followed by Xen, an Open Source Software solution [13], and later on by the Linux kernel with its KVM (*Kernel-based Virtual Machine*) virtualization solution [2].

The principal motivations for this virtualization renaissance are cost-cutting (in the areas of hardware infrastructure, power consumption and cooling systems), business continuity and server manageability. Other motivations are the possibility of doing development and testing on virtual environments, having a faster deployment process using virtual appliances, running legacy systems, improving security through isolated environments, and running existing operating systems on multiprocessor architectures.

Virtualization presents some disadvantages as well. For example, system management becomes more complex (there is currently a lack of assistance tools for this task) and the I/O system becomes a bottleneck when the hosted virtual machines are under an intensive I/O workload.

2.1.2 Types of Virtualization Solutions

There are two different ways to provide virtualization: *hypervisor-based virtualization* and *operating system-level virtualization.* The first type uses a *hypervisor* (or *vir-*

[2]From now on, when talking about virtualization, system virtualization should be understood, unless otherwise noted.

tual machine monitor), which is a layer of software that runs directly on hardware[3]. The hypervisor creates virtual machines on top of which operating systems can be installed. The hypervisor manages the allocation of hardware resources to the guest operating systems.

The hypervisor runs in the most privileged mode and the guest operating systems run in a less privileged mode (although not the least privileged one). Every attempt from the guest operating systems to access shared resources is intercepted by the hypervisor, which verifies the validity of the operation and performs it on behalf of the guest operating systems.

One of the operating systems running on top of the hypervisor has higher privileges than the guest operating systems and it is used for management purposes, that is, to run management software that can interact with the hypervisor. This distinguished operating system is known as the host operating system.

The guest operating systems can be *fully-virtualized* or *paravirtualized.* In the first case, the guest operating systems are unmodified operating systems, designed to be run on top of bare hardware. In this scenario, guest operating systems are unaware that they are running on top of virtual machines, and the process of intercepting system calls and executing them on behalf of the guests remains unknown to the guests themselves.

Guest operating systems that are paravirtualized have had the source code modified to replace system calls with *hypercalls* (that is, calls to the hypervisor). This results in no interception and verification of system calls, thus improving performance.

The second form of providing virtualization is through operating system-level virtualization. In this case, an operating system with virtualization capabilities runs on top of the hardware, doing essentially what an operating system would do in a non-virtualized environment.

These operating systems have features for enabling the simultaneous existence of multiple user-space environments called *containers* (also known as Virtual Private

[3]There is an additional type of hypervisor that runs on top of an operating system (hosted virtualization). It is known as *type 2 hypervisor.*

Servers or Virtual Environments). The containers share the operating system's kernel (as it does not happen in hypervisor-based virtualization), but everything that runs on top of the kernel is independent (that is, each container has its own copy).

In the following section, an operating system that uses containers is presented.

2.2 OpenVZ

OpenVZ is an operating system kernel that provides operating system-level virtualization [3, 22]. It is essentially a Linux kernel modified[4] to run multiple, isolated *containers* (i.e. virtual user-space environments) on a single physical server. It provides close to native performance [27], scalability and dynamic resource management. However, due to its nature, this virtualization solution can only run containers based on GNU/Linux distributions.

The purpose of OpenVZ is to support the execution of multiple containers. The host system itself runs inside a privileged container. The containers are isolated program execution environments, which appear as stand-alone physical servers to their users. Each container has its own set of processes starting with the **init** process, file system, users (including **root** account), applications, memory, network interfaces with IP addresses, routing tables, firewall rules, etc.

OpenVZ adds three new functionalities to the Linux kernel: virtualization and isolation (of various subsystems), checkpointing, and resource management. The virtualization and isolation of resources enables the simultaneous execution of multiple containers. For example, each container requires its **init** process to have process identifier (PID) equal to 1. From the point of view of an operating system, it is not possible to have two (or more) processes with the same PID. However, since the containers are isolated from each other, it is possible to repeat PIDs between containers.

Container checkpointing is the ability to suspend an executing container, save its state to a file and restart it again later. Container migration is a natural extension

[4]Some of these modifications have even made its way into mainstream Linux.

of checkpointing: the container is first suspended, its state file is transferred to a new location and the container is restarted there. All the network connections are migrated with the container, so the user perceives no downtime, but does perceive a delay in processing. This process is also known as *live migration.*

OpenVZ implements a dynamic resource management subsystem that enables the definition of limits (and guarantees) for the resources assigned to each container. The subsystem is organized into these four components:

- *Two-level disk quota*: The host administrator can define per-container disk quotas in terms of disk blocks and number of inodes. In the second level, each container administrator can use standard UNIX quota tools to define per-user and per-group disk quotas;

- *'Fair' CPU scheduler*: A two-level implementation of the fair-share scheduling strategy. OpenVZ's CPU scheduler first picks a container to allocate the next time slice based on CPU priorities and CPU limits of the containers. The standard Linux scheduler then chooses a process from the selected container based on standard process priorities;

- *Two-level I/O scheduler*: In the first level, the host's I/O scheduler distributes the available I/O bandwidth among containers based on I/O priorities of the containers. In the second level, the processes inside each container are scheduled with the Completely Fair Queuing I/O scheduler [4, 5];

- *User Beancounters*: Each container has a set of about 20 parameters [17] (counters, limits and guarantees) to regulate the allocation and measure the utilization of memory and in-kernel objects, such as IPC (Inter-Process Communication) shared memory segments or network buffers. Each parameter has five values associated with it: *current usage*, *maximum usage* (over the container's lifetime), *barrier*, *limit* and *fail counter.* The meaning of barrier and limit is parameter-dependant, although they can be considered some kind of soft and

hard limit, respectively. The fail counter is increased each time the resource hits its limit.

For management purposes, each container has a configuration file. This file provides details on the resource allocation for the container. The configuration file also has additional information, such as the container's identification number (CTID), its IP, hostname, etc. The privileged container, identified with CTID 0, does not have a configuration file. Listing 2.1 shows a sample configuration file.

OpenVZ relies on two utilities for the management of containers: `vzctl` and **vzpkg**. These utilities run on the privileged container. `vzctl` implements a high-level command line interface for managing the containers (creation, modification, destruction) and **vzpkg** is a set of tools for the creation of *template caches*. The template caches are compressed archives of **chroot**ed environments with selected software packages installed on them. The selection of software packages to be installed in a template cache is called *template*. When a container is created, a template cache needs to be specified, for it to be decompressed. The decompressed environment is the container.

The live migration process in OpenVZ consists of two phases. First, the container's filesystem is copied to the destination machine while the container is still running. In the second phase, the container is checkpointed, its state file is transferred to the destination machine and a second copy of the container's filesystem is started. This second copy is incremental, in the sense that it only affects those files that were modified after the first copy. When the second copy finishes, the container is restored from the state file at the destination machine.

OpenVZ does not provide container replication, but it can be implemented. The first step is to stop the container that is to be replicated. The container's filesystem and configuration file are then copied. (Once the filesystem is copied, the original container can be restarted.) The third step is to modify the configuration file with the CTID to be used for the replica. After this last step, the replica can be started.

Listing 2.1: Sample Container configuration file

```
ONBOOT="yes"

# UBC parameters (in form of barrier:limit)
KMEMSIZE="11055923:11377049"
LOCKEDPAGES="256:256"
PRIVVMPAGES="65536:69632"
SHMPAGES="21504:21504"
NUMPROC="240:240"
PHYSPAGES="0:2147483647"
VMGUARPAGES="33792:2147483647"
OOMGUARPAGES="26112:2147483647"
NUMTCPSOCK="360:360"
NUMFLOCK="188:206"
NUMPTY="16:16"
NUMSIGINFO="256:256"
TCPSNDBUF="1720320:2703360"
TCPRCVBUF="1720320:2703360"
OTHERSOCKBUF="1126080:2097152"
DGRAMRCVBUF="262144:262144"
NUMOTHERSOCK="360:360"
DCACHESIZE="3409920:3624960"
NUMFILE="9312:9312"
AVNUMPROC="180:180"
NUMIPTENT="128:128"

# Disk quota parameters (in form of softlimit:hardlimit)
DISKSPACE="1048576:1153024"
DISKINODES="200000:220000"
QUOTATIME="0"

# CPU fair sheduler parameter
CPUUNITS="1000"

VE_ROOT="/vz/root/$VEID"
VE_PRIVATE="/vz/private/$VEID"
OSTEMPLATE="centos-5-i386-default-5.2-20090117"
ORIGIN_SAMPLE="vps.basic"
IP_ADDRESS="129.100.18.91"
HOSTNAME="one.com"
NAMESERVER="129.100.16.252 129.100.16.246 129.100.2.12"
NAME="c1891"
```

2.3 Summary

This chapter covered basic information necessary to build a context for the following chapters. The concept of virtualization was presented with a focus on system virtualization. Two types of virtualization solutions were described as well: hypervisor-based virtualization and operating system-level virtualization.

OpenVZ, an operating system with virtualization capabilities, was presented. Its distinctive features were described, as well as its migration mechanism. It was also explained how replication can be implemented.

Chapter 3

Related Work

This chapter reviews the research pursued by other groups in the same or related areas. Section 3.1 discusses resource monitoring. Section 3.2 deals with algorithms and policies. Section 3.3 features diverse resource management systems for virtualized environments. Section 3.4 presents studies on the virtual machine migration process. Section 3.5 deals with decision-making support tools. Section 3.6 has a discussion of systems management.

3.1 Resource Monitoring

Resource monitoring is an essential building part for a dynamic resource management system. Resource utilization statistics are essential to making management decisions.

Wood et al. developed for their resource management system, Sandpiper [33], two monitoring mechanisms: *black-box* and *grey-box*. Black-box monitoring consisted of collecting statistics from the virtual machines (CPU, network and memory utilization) without directly contacting the virtual machines. Only the information available from the host operating system was used. On the other hand, grey-box monitoring required the use of an additional software module running inside each virtual machine. This module (or daemon) would collect operating system statistics (also CPU, network and memory utilization) and process application logs.

In Golondrina, only black-box monitoring is used. Although grey-box monitoring provides for a better understanding of the virtual machines' resource utilization (and

hence better prediction of future demand), it is also a more invasive mechanism. A software module needs to run inside the virtual machines for statistics collection purposes and virtual machines owners may not agree to that.

3.2 Algorithms and Policies

In order to automate resource management, algorithms and policies for determining suitable migrations and consolidation patterns are required. In addition, system administrators should be able to define policies to customize the system's behavior.

Hyser et al. presented a dynamic virtual machine placement system [20]. The proposed architecture consisted of a virtualization management service and a virtual machine placement service. The virtualization management service provided information about the organization of the physical nodes and virtual machines, and performance data to the virtual machine placement service. The virtual machine placement service used the provided information in conjunction with user-defined policies to determine migrations for the virtualization management service to execute.

The system had a clear separation between virtual machine management and decision-making, but there was no in depth study of policies. The following work filled that gap.

Gmach et al. developed a resource management system for workload consolidation through migration [18]. The system was designed to optimize resource utilization (power included) and to provide Quality of Service. The system consisted of a workload placement controller and a workload migration controller. The workload placement controller used trace-based techniques[1] to study workload behavior in order to determine the best workload consolidation. The controller was used to determine the original workload placement and it was periodically called to redo the placement. The workload migration controller (fuzzy-logic based) was responsible for periodically checking the physical hosts for overload situations (requiring workload

[1]Trace-based techniques use sequences of observations collected during a system's run to study the behaviour of the system.

migration) and underload situations (requiring workload eviction and host turn off).

Different overload and underload thresholds for the migration controller were evaluated. The threshold values that offered the best balance between resource utilization and quality of service were selected. This work defined policies for the workload placement controller and studied the effectiveness of each policy. They found that using a historical policy (for studying the traces) in combination with the migration controller provided the closest results to an ideal situation (i.e. perfect knowledge of future workload behavior).

Currently, Golondrina implements the equivalent to the workload migration controller. However, a placement controller could be implemented as well with a historical policy. The experiments to determine the thresholds for the workload migration controller could be replicated as well, so as to determine the most suitable thresholds for Golondrina.

3.3 Managing Multiple Virtual Machines within Multiple Physical Servers

A data center can be composed of dozens to thousands of physical servers. Each physical server can run multiple virtual machines. Thus, a resource management system for a data center needs to be able to manage such a crowded environment.

Bhatia and Vetter worked on the development of a Virtual Cluster Manager [15]. The objective was for the system to manage a cluster of physical servers, each of them hosting virtual machines, while hiding from the system administrator virtualization platform details. The system provided features such as automatic load balancing and eviction of all the virtual machines hosted in a physical server in preparation for maintenance tasks. The system would also enable the definition of policies for load balancing and node failure handling.

The system was based on a client-server architecture. XCM Daemons ran in the physical servers gathering performance metrics which were sent to the XCM Client.

The XCM Client, running in a remote node, combined the collected data with cluster-wide information. The XCM Client could display performance information from the virtual machines running in each physical server and provided three features for system administration: manual live migration of virtual machines, automatic load balancing, and preemptive node maintenance (eviction of all the virtual machines hosted in the node).

The authors did not build an autonomic system, but rather a set of system management tools for an administrator to use. In addition, the system worked with the granularity of virtual machines. In other words, the system used virtual machine migration to do load balancing, but did not consider doing resource reallocation.

Wood et al. developed Sandpiper, a resource management system that used virtual machine migration to deal with resource stress situations [33]. The system would automatically monitor resource utilization, detect *hotspots* (or resource stress situations), and trigger migrations. They studied two approaches to monitoring: black-box and grey-box (see Section 3.1 for more information).

Sandpiper was based on a client-server architecture. A component called *Nucleus* ran in each physical server. It monitored resource utilization and sent resource utilization statistics to the central system. The central system ran in an individual physical node and it was composed of three parts: *Profiling Engine*, *Hotspot Detector*, and *Migration Manager*. The Profiling Engine received the resource utilization statistics collected at each physical server. This was used to build resource utilization profiles for each virtual server and aggregate profiles for each physical server.

The Hotspot Detector constantly monitored the resource utilization profiles in order to detect hotspots (situations where the resource utilization exceeds certain threshold or a Service Level Agreement is violated during a certain amount of time). The Migration Manager was invoked upon a hotspot detection to decide which virtual servers had to be migrated, where they had to be moved, and how much resources had to be allocated in the new host.

Golondrina is based on some of the ideas implemented in Sandpiper, but adds replication as a mechanism to deal with resource stress situations. In addition, Golon-

drina only implements black-box monitoring. Grey-box monitoring can be considered an invasive mechanism that clients might find undesirable.

Zhu et al. developed an automated resource management system that consisted of three controllers that worked at different hierarchichal levels and intervals in time [36]. The purpose of the system was to enable clients and system administrators to focus on policy settings.

The system's design combined three controllers: node controller, pod controller and pod set controller. The node controller did resource reallocation among the workloads hosted in a physical node. It was a two-level control system, composed of utilization controllers and an arbiter controller. The utilization controllers measured the average resource consumption and requested the next resource allocation based on resource utilization objectives. The arbiter allocated resources based on resource requests, resource availability and workload priorities.

The physical nodes were organized in pods (groups). The pod controller received information from each node controller and triggered migrations with the objective of achieving Quality of Service goals (e.g. avoid resource stress situations) while maximizing resource utilization. The pod set controller was responsible for studying the overall performance of various pods and migrating workloads between pods to improve performance.

The authors reported that the integrated work of the three controllers offered good results in terms of resource utilization and Quality of Service. This work was built upon previous studies [27, 20, 18].

Marinescu and Kroeger proposed an autonomic framework for resource management [24]. The objective was to design a modularized framework with a clear separation between control and management modules. This characteristic would enable different intelligent controllers to be plugged into the system, allowing for management strategies (developed by different research groups) to be compared over different infrastructures with a common evaluation mechanism.

The framework was composed of VM Managers (one per virtual machine), Physical Managers (one per physical server) and one Cluster Manager. The VM Manager

would monitor the virtual machine's resource utilization and the hosted application (inside the virtual machine) performance regarding Service Level Objectives (e.g. keep a web server's response time below a certain threshold). If the Service Level Objectives were not satisfied, the controller would request more resources from the Cluster Manager.

The Physical Manager was responsible for monitoring the physical server's resource utilization and for executing instructions from the Cluster Manager. The Cluster Manager (an intelligent controller) would use the data provided by the Physical Managers to determine how to satisfy the resource requests from the VM Managers.

Finally, Zhang et al. took a very different approach in the matter of resource management. They developed a control model for virtual machines self-adaptation [34]. The objective was for virtual machines to adapt their resource demand based on feedback from the system regarding resource availability.

For this purpose, they developed a mechanism to provide the virtual machines with feedback concerning the state of their bottleneck resource. They also developed a mechanism for virtual machines to scale up/down the resource demand. Finally, they came up with an adaptation strategy that would combine the work of the two previous mechanisms. This strategy would determine how to modify the resource demand based on the system's feedback.

3.4 Characterizing Migration Process Behavior

Virtual Machine Migration is a useful mechanism for resource management in data centers. However, the migration process needs to be well understood in order to make an efficient use of it.

Zhao and Figueiredo presented an experimental study on characterizing the virtual machine migration process and predicting its performance [35]. The experiments included parallel and in sequence migrations, and the use of CPU- and memory-intensive applications inside the virtual machines. The migration process time length was measured and the effect the migration process had on the hosted applications'

performance (measured as the increase of the applications' runtime) was measured as well. The authors concluded that the migration process time and performance could be predicted for a number of virtual machines based on measurements from a single virtual machine.

Kochut and Beaty developed an analytical model for virtual machine migration to help in estimating the improvement in response time due to a migration decision [21]. The model accounted for virtual machine resource demand prediction, the time it took to migrate a virtual machine between two physical servers, and the migration process's overhead. Given the current system's load and considering the virtual machines' future demand, the model could help decide if a migration process should be started and which virtual machines should be migrated.

Both studies represent a step forward in the development of decision-making support mechanisms and could be replicated in Golondrina for that purpose.

3.5 Decision Support Infrastructure

If the resource management system is to be autonomic, mechanisms for decision-making support are required. Once there is a resource management system available, it is possible to move on with research in this area.

Munasinghe and Anderson worked on the development of a data center architecture (hardware and software configuration) that could provide guaranteed Quality of Service to its clients [25]. They built the system's infrastructure (that is, virtualization solution, shared storage, and control infrastructure) and worked on adding autonomic capabilities to the system so it could reconfigure itself based on monitoring data and Quality of Service objectives.

For the automatic resource scaling feature, they designed two alternatives: *vertical scaling* and *horizontal scaling*. For vertical scaling, the system would reallocate resources among the virtual machines hosted in a physical server, or the system would trigger virtual machine migrations. For horizontal scaling, the system would clone virtual machines and start the copies in different physical servers. A load balancer

would distribute the load between the cloned virtual machines.

The current prototype of the system only implements horizontal scaling, which Golondrina can achieve as well. In addition, Golondrina is already able to do vertical scaling through virtual machine migration.

Begnum et al. worked on decision support mechanisms for virtual machine placement [14]. They proposed three technology-independent metrics for evaluating the state of virtualized infrastructures: *redundancy level*, *resource conflict matrix* and *location conflict table*. These metrics would help in defining administration policies and evaluating policy compliance.

The first metric had the form R/S, where R was the total number of physical servers available and S was the number of servers that could be taken down without impeding a virtual machine from keeping running. The resource conflict matrix metric was a per-resource tool that showed which virtual machines were using the same resource and hence could be in conflict. The location conflict table metric ordered the physical servers based on the overall number of conflicts that resulted from considering the different resources. This metric would help in evaluating which server would have the least number of conflicts when placing an additional virtual machine there.

3.6 Management Tools for Virtual Machines

System Management involves the deployment and overall management of a cluster of physical servers. A resource management system could be integrated into system management software or could work side by side with such software.

Vallée et al. developed OSCAR-V, a set of tools for the deployment and management of host operating systems and virtual machines in a cluster [31]. These tools were extensions to OSCAR, a toolkit for cluster installation, configuration and management. OSCAR-V consisted of the tool V3M (Virtual Machine Management and Monitoring tool). However, the monitoring features were not implemented.

Krsul et al. worked on a framework for virtual machine management in the Grid [23]. They developed the Grid service VMPlant, which enabled the automatic config-

uration and instantiation of virtual machines, and also provided monitoring capabilities. The client interacted with a VMShop, a front-end for the Grid service, to create, query and destroy virtual machines. In order to create a virtual machine, the client provided VMShop with a series of specifications and VMShop would find a VMPlant (physical server) to instantiate the virtual machine. The VMPlant selection process consisted of a bidding process where each VMPlant provided a creation cost for the requested virtual machine and the VMPlant with the lowest cost was selected.

Ramakrishnan et al. advocated for enabling live virtual machine migration between data centers across WANs (Wide Area Networks) [28]. This feature would help in dealing with scheduled data center maintenance and unexpected failures (outages). They proposed a Migration Management System that would coordinate the migration process with these three subsystems involved: virtualization platform, network management system, and storage subsystem. However, some of the necessary capabilities they pointed out are not yet available. New network features are needed for promptly redirecting network traffic to the virtual machines' new location and storage replication technologies need to incorporate a different approach to remote replication.

3.7 Summary

This chapter presents the work done by other researchers in areas related to the topic of this research. Resource monitoring techniques are discussed, as well as studies on algorithms and policies for virtual machine placement. Different resource management systems developed as research projects are presented. Decision-making support tools are described, jointly with studies on the virtual machine migration process. System management is also presented at the end for completeness.

This survey shows that research is being conducted to develop mechanisms for decision making support. Once effective mechanisms are developed, it will be possible to study resource management strategies that make use of those mechanisms. Our contribution in this direction consists of studying virtual machine replication as a

mechanism to deal with resource stress situations. Most work in dynamic resource provisioning in data centers focus only on virtual machine migration.

Chapter 4

Architecture

This chapter presents the architecture of *Golondrina*, an autonomic resource management system. Section 4.1 describes the infrastructure in which the system operates. Section 4.2 provides a bird's-eye view of the whole system. Section 4.3 describes the interaction between the system's components. The remaining Sections focus each on a different component of the system, detailing its functionality, design and interface.

4.1 Infrastructure

The infrastructure where the system runs is a cluster of homogeneous physical servers. A physical server can have one of three roles. One role that a physical server may have is that of an *OpenVZ hardware node*, that is, a host of containers.

Another role is that of the *manager server*. All communications relative to the management of the cluster take place between the manager server and any other physical server.

The last role is that of the *gate of the cluster*. All service requests (external communications) come in through this physical server, which redirects the requests to the appropriate hardware node using a load balancing software.

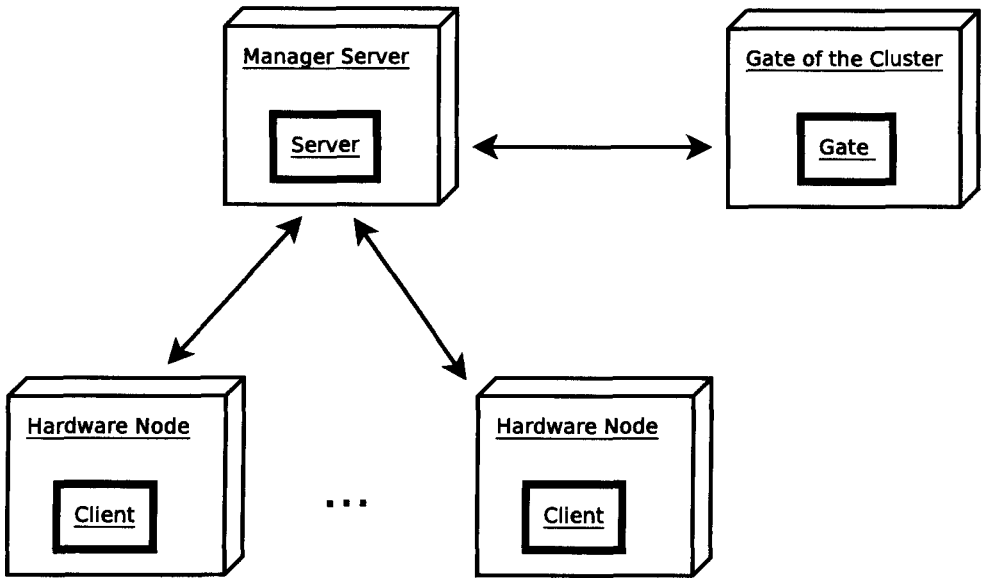

Figure 4.1: Golondrina's architecture

4.2 Bird's-eye View

Golondrina is organized as a client-server architecture [30] (see Figure 4.1). A *Client* component runs in every hardware node and the *Server* component runs in a non-virtualized physical server. An additional component, the *Gate*, runs also in a non-virtualized physical server, which is used as the gate of the cluster.

The Client component has two responsibilities. The first responsibility is to gather resource utilization statistics[1] from the containers and the hardware node and to communicate these statistics to the Server. The second is to execute relocations based on instructions from the Server.

The Server component has three responsibilities. The first responsibility is to receive and process the clients' statistics. The second responsibility is to analyze the collected data to detect potential resource stress situations. The third responsibility is, upon detection of a resource stress situation, to find a sequence of relocations that could dissipate the problem.

[1]Currently, the system only manages CPU, so the Client gathers CPU utilization statistics. However, collecting other resources utilization statistics is feasible and it is part of our future work.

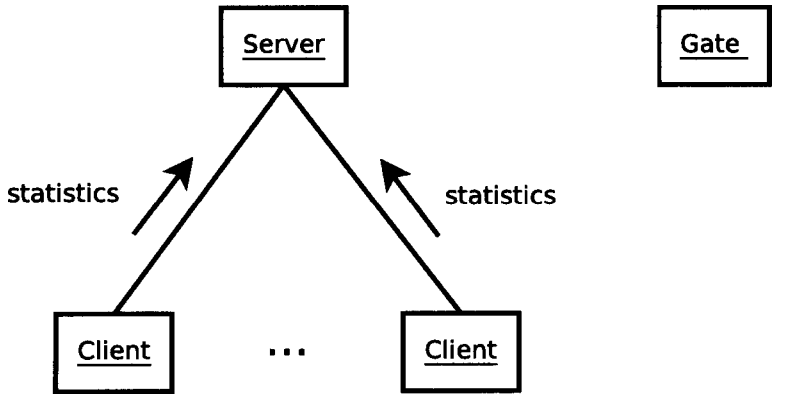

Figure 4.2: Monitoring

Finally, the gate updates the load balancer's configuration based on instructions from the Server.

4.3 Interactions

The components of the system interact in three situations. The most basic interaction is when a Client sends CPU utilization statistics to the Server (see Figure 4.2). This situation happens periodically.

The second form of interaction occurs when a resource stress situation is detected and the Server contacts a Client for it to migrate a container away from its current hardware node (see Figure 4.3). Once the migration is completed, the Client contacts the Server to inform of the operation's completion.

The third form of interaction takes place upon detection of a resource stress situation (see Figure 4.4). In this case, the Server requests that a Client replicate a container. The Client contacts the Server once the replication has been completed and the Server then contacts the Gate to inform it about the replication.

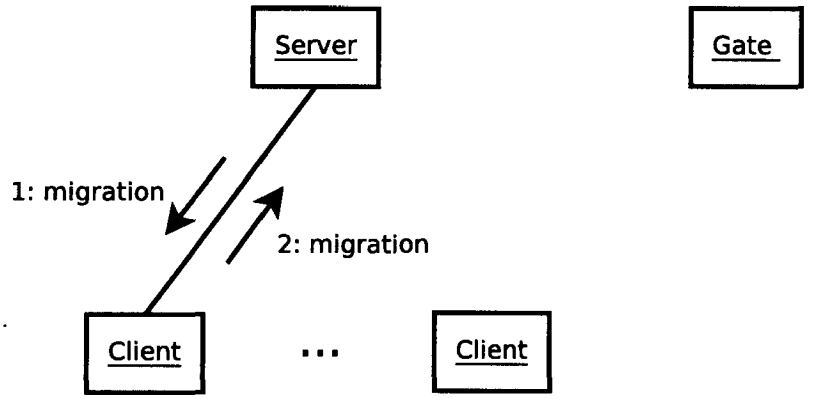

Figure 4.3: Migration

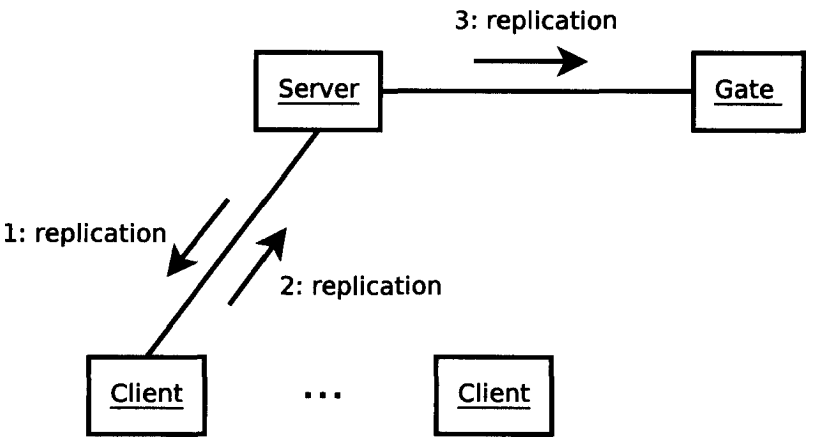

Figure 4.4: Replication

4.4 Client

The Client component was designed to run in a hardware node, implementing the monitoring and relocating mechanisms of the system. Running from the privileged container of the hardware node, the Client has access to the operating system's configuration and accounting files.

One function of the Client component is to periodically gather CPU utilization statistics about the containers and the hardware node, and send these statistics to the Server. The monitoring treats the containers as black boxes, that is, only using information that can be obtained without directly contacting the containers. The Client obtains the CPU utilization statistics from the hardware node's operating system accounting files.

Another function of the Client is to execute relocations based on instructions from the Server. When a resource stress situation is discovered, the Server may instruct a Client that a container has to be migrated out of the hardware node or that a new container (a replica) has to be started in the hardware node. To carry out these tasks, the Client contacts the underlying operating system to use the OpenVZ management tools (see Section 2.2).

There are complementary tasks that the Client has to do. For example, collecting miscellaneous information from the hardware node, such as its host name and the number of cores of the CPU, and from the containers, such as their host names and IP addresses.

4.4.1 Design

The Client component's design consists of six modules: Client, Sensor, Actuator, ContainerData, ClientProtocol and ClientFactory (see Figure 4.5). The Client module implements the logic of the component. It periodically calls the Sensor to gather CPU utilization statistics and calls the ClientProtocol to send messages to the Server. It contacts the Actuator when a relocation has to be done and parses the messages sent by the Server and carries on the appropriate actions.

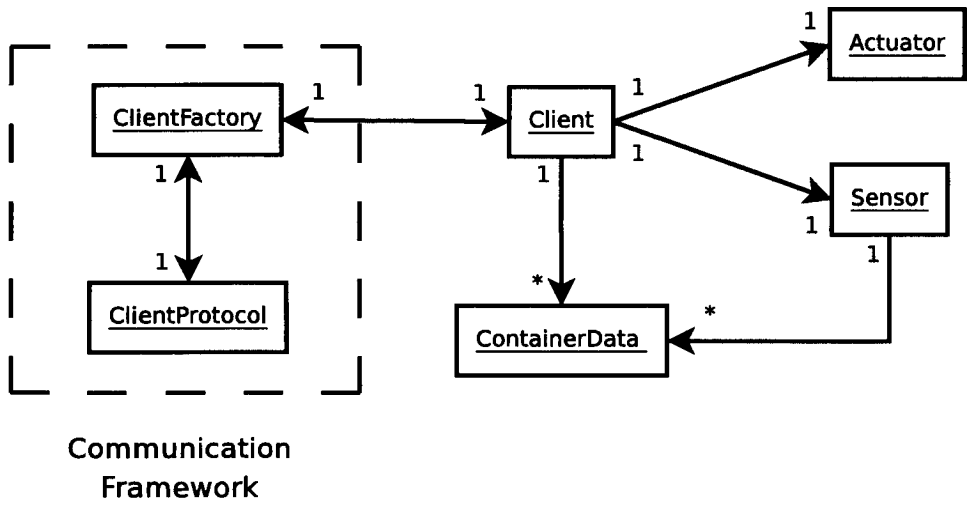

Figure 4.5: Client component's design

The Sensor encloses all the functionality related to obtaining information from the operating system. It reads accounting files to gather CPU utilization statistics, and reads configuration files to obtain hostnames and IP addresses.

The Actuator handles all the tasks that entail a change of state in the hardware node. It modifies configuration files and issues commands to the underlying operating system to trigger container migrations or start replicas.

The ContainerData module is a data storage entity. There is an instance for each container currently running in the hardware node, which stores the container's identification (**ctid**), hostname, IP address, and last CPU utilization statistics.

The ClientProtocol and ClientFactory modules handle the communication with the Server. The ClientFactory creates a ClientProtocol to establish a connection with the Server and reestablishes the connection when the latter gets lost. The ClientProtocol sends and receives messages to and from the Server.

4.4.2 Interface

The Client component uses message passing to communicate with the Server. Two kind of messages can be received:

- migration(CTID, DEST_ID)

The first message indicates to the Client (source of the relocation) that the container with identification **CTID** has to be migrated to the hardware node with identification **DEST_ID**. The Actuator module starts the migration process.

- replication(CTID, IP)

The second message indicates to the Client (target of the relocation) that the container with identification **CTID** has to be replicated with IP address **IP** in the hardware node where the Client is running. The Actuator module starts the replication process.

4.4.3 Required Calculations

The Client has to periodically gather CPU utilization statistics from the containers and the hardware node. These statistics are not directly available, but can be calculated based on information from the operating system's accounting files.

At time t, two values can be obtained for each non-privileged container: $uptime_t$ and $used_t$. The first value is the elapsed time since the container was started and the second value indicates the amount of CPU time that the container has actually used. Based on those two values, it is possible to calculate the proportion of used CPU time over the total available CPU time in the time interval $(t-1, t]$. The following is the equation used for that purpose:

$$u_t = \frac{used_t - used_{t-1}}{uptime_t - uptime_{t-1}} \tag{4.1}$$

The same equation is applied to calculate the proportion of CPU time used by the hardware node in the time interval $(t-1, t]$.

4.5 Server

The Server component runs in a non-virtualized physical server and implements decision making support mechanisms, such as data processing and analysis, and decision making processes.

The Server receives and processes the statistics that are periodically sent by the Clients. The statistics are stored in a registry, organized by hardware node and container.

The second function of the Server is to periodically analyze the collected data to search for potential resource stress situations. Mathematical models are built for each hardware node and container and updated with every new observation. These models are used to predict the next CPU utilization value, in an attempt to be proactive in dealing with resource stress situations.

In order to define a resource stress situation, a threshold for hardware nodes' CPU utilization is defined. Every time the predicted CPU utilization of a hardware node exceeds that threshold, a flag is raised as a warning for that hardware node. A hardware node is considered to be under a resource stress situation if the current resource stress check and k out of the previous n checks resulted in flags being raised.

Finally, the third function of the Server is to find a sequence of relocations that could dissipate a resource stress situation. The system attempts to find a container that could be relocated and a hardware node that could be used as target for the relocation.

4.5.1 Design

The Server component's design consists of seven modules: Server, Registry, HardwareNode, Container, Relocator, ServerProtocol and ServerFactory (see Figure 4.6). The Server module implements the logic of the component. It periodically runs resource stress checks and calls the Relocator upon detection of a resource stress situation. It contacts the Clients to trigger relocations and contacts the Gate component to inform of successful replications. Finally, it parses the messages from the Clients

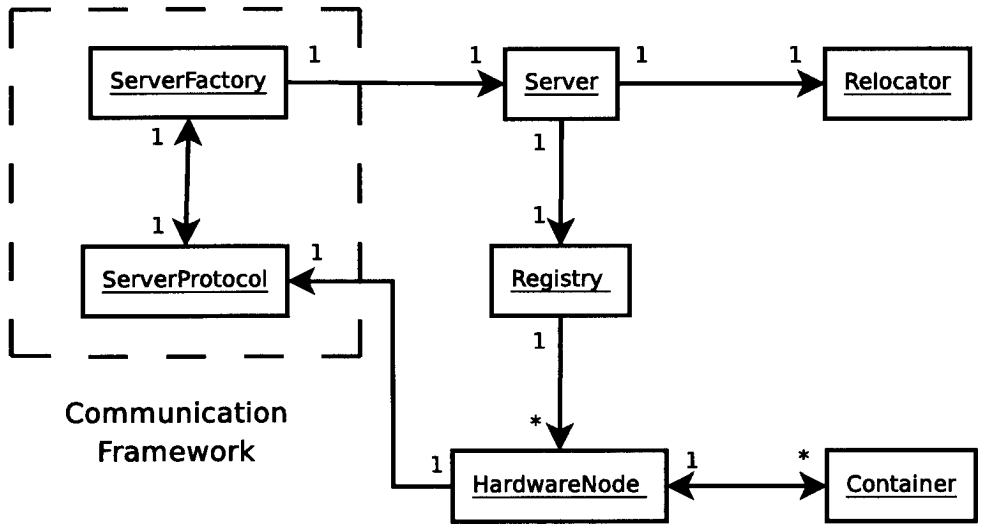

Figure 4.6: Server component's design

and carries the appropriate actions.

The Registry stores the collected data gathered from the hardware nodes and containers. It processes the statistics reports sent by the Clients and maintains a list of HardwareNode modules.

The HardwareNode module is a data storage entity that represents a physical hardware node in the system. It stores static information, such as the hardware node's hostname and the number of cores of the CPU, and dynamic information, such as the number of relocations in which the hardware node is currently involved and the resource stress flags. It is associated with a list of Container modules.

The Container module is a data storage entity that represents a container in the system. It is used to store the collected CPU utilization observations and to build and maintain the mathematical model used to predict the next CPU utilization value.

The Relocator implements the decision making processes of the system. It uses policies to determine which container to relocate and where that relocation should take place.

The ServerProtocol and ServerFactory modules handle the communications with the Client instances and the Gate component. The ServerFactory creates a ServerProtocol to handle an incoming connection and the ServerProtocol exchanges messages

with the ClientProtocol or GateProtocol that started the connection.

4.5.2 Interface

The Server component communicates with the Client instances and the Gate through message passing. Four kind of messages can be received:

- statistics(HNID, CORES, CTID, CPU_USAGE)

The first message is a CPU utilization statistics report. It provides the last **CPU_USAGE** observation of the container with identification **CTID**, which is hosted in the hardware node with identification **HNID**. The message also provides the hardware node's number of **CORES**. The Registry module processes the report and stores the information.

- hello()

The second message is a salutation from the Gate component. Its objective is to register a communication channel with the Server. No additional input information is transmitted. The Server module registers the communication channel.

- migration(SRC_ID, DEST_ID, CTID)

The third message indicates that the container with identification **CTID** has been successfully migrated from the hardware node with identification **SRC_ID** to the hardware node with identification **DEST_ID**. No actions are triggered by this message.

- replication(HNID, CTID, HOSTNAME, IP)

The fourth message indicates that the container with identification **CTID** has been successfully replicated in the hardware node with identification **HNID**. The message also provides the replica's **HOSTNAME** and **IP** address. The Server module forwards the information to the Gate. Clients cannot send the information directly to the Gate because they do not know about the existance of the Gate (i.e. there is no need for the Clients to know how load balancing is achieved in the cluster).

4.5.3 Required Calculations

One of the tasks carried on by the Server is the creation and update of mathematical models for the containers and hardware nodes. These models are used to predict the CPU utilization of a container or hardware node in the next time interval based on a sequence of previous observations (see Subsection 4.4.3 for an explanation on how the Client obtains the observations).

The mathematical model used is the Auto-regressive Model of Order 1 AR(1) [16], which relies on the last observation in the sequence and two parameters: μ, the mean of the values in the sequence, and θ, which accounts for the variations of the values in the sequence.

Given a sliding window of CPU utilization statistics $W = [u_x, ..., u_t]$ with maximum size w and where $x = max(0, t - w + 1)$. The parameters μ and θ at time t are calculated as follows:

$$\mu_t = \frac{\sum_{i=x}^{t} u_i}{t - x + 1} \tag{4.2}$$

$$\theta_t = \frac{\sum_{i=x}^{t-1}(u_i - \mu_t) * (u_{i+1} - \mu_t)}{max(1, \sum_{i=x}^{t-1}(u_i - \mu_t)^2)} \tag{4.3}$$

Having the values u_t, μ_t and θ_t, it is possible to predict the CPU utilization at time $t + 1$:

$$\hat{u}_{t+1} = \mu_t + \theta_t * (u_t - \mu_t) \tag{4.4}$$

The predicted CPU utilization of a hardware node is used during the periodic resource stress check that the Server does for every hardware node. A hardware node is considered to be under a resource stress situation if k out of the previous n resource stress checks exceeded the CPU utilization threshold and the next predicted CPU utilization also exceeds the CPU utilization threshold:

$$overloaded \leftarrow (\sum_{i=1}^{n}(\hat{u}_i > threshold) \geq k) \wedge (\hat{u}_{t+1} > threshold) \tag{4.5}$$

The predicted CPU utilization of a container comes into play when there is a resource stress situation in the hardware node that hosts the container. The system might attempt to relocate the container and for that purpose it queries the container's profiled CPU utilization. This value represents the amount of CPU that should be allocated to the container at the target hardware node.

The profiling method uses a historical policy to calculate the container's profiled CPU utilization. This value has to satisfy a given percentile of the resource needs registered in the last window of time $W = [u_x, ..., u_t]$, in addition to the container's current resource needs. The method sorts in increasing order the collected statistics in W and takes the value corresponding to the 90^{th} percentile. The profiled CPU utilization at time $t + 1$ is the maximum of the 90^{th} percentile and the container's next predicted CPU utilization plus an additional Δ:

$$\bar{u}_{t+1} = max(90^{th}percentile, \hat{u}_{t+1} + \Delta) \tag{4.6}$$

During the relocation process, the Server checks with non-resource-stressed hardware nodes whether they can host (*accept*) an additional container without exceeding the CPU utilization threshold:

$$accept \leftarrow \hat{h}_{t+1} + \bar{u}_{t+1} < threshold \tag{4.7}$$

where $\hat{h}_{t+1}$ and $\bar{u}_{t+1}$ stand for the predicted CPU utilization of the non-stressed hardware node and the profiled CPU utilization of the container, respectively.

4.6 Gate

The Gate component runs in a non-virtualized physical server. This physical server acts as the *gate* of the cluster, forwarding the service requests to the appropriate

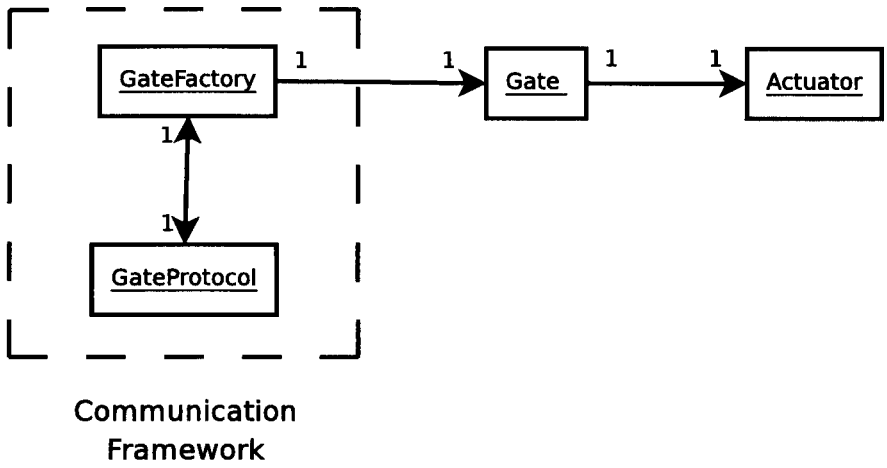

Figure 4.7: Gate component's design

containers. The only function of this component is to update the load balancer's configuration based on instructions from the Server component.

4.6.1 Design

The Gate component's design consists of four modules: Gate, Actuator, GateProtocol and GateFactory (see Figure 4.7). The Gate module implements the logic of the component. It parses the messages sent by the Server and contacts the actuator to carry the appropriate actions.

The Actuator is responsible for dealing with the underlying system. It modifies the load balancer's configuration and contacts the operating system to restart the load balancer.

The GateProtocol and GateFactory modules handle the communication with the Server. The GateFactory creates a GateProtocol to establish a connection with the Server and reestablishes the connection when the latter gets lost. The GateProtocol sends and receives messages to and from the Server.

4.6.2 Interface

The Gate component communicates only with the Server. It sends a message once to establish a connection and then limits itself to receive messages. Only one kind of message can be received:

- replication(HOSTNAME, IP)

This message indicates that a new container for the service **HOSTNAME** has been created and its IP address is **IP**. The Actuator module updates the load balancer's configuration with this information.

4.7 Summary

This chapter presented the architecture of the autonomic resource management system *Golondrina*. The infrastructure in which the system operates was presented as well. The system's design was discussed first at the component level, defining each component's functionality and the interactions between the components. Later on, each component was analyzed. The responsibilities of the components were discussed and the design in modules was described. The interface of each component was also presented and details were provided on the calculations performed by each component.

Chapter 5

Implementation

This chapter provides details about the implementation of the system. Section 5.1 provides general information about the software that was used to build the system and the infrastructure. The remaining Sections present each the methods that implement the functionality of the different components of the system.

5.1 General Information

The system was implemented in the scripting language Python, version 2.4.3. The communications between system's components were handled through the event-driven networking engine Twisted [6], version 8.2.0, which also enabled the definition of periodically triggered tasks.

The physical servers run the operating system CentOS [7], release 5.2, with kernel OpenVZ [3], release 2.6.18-92.1.18.el5.028stab060.2. The gate of the cluster runs Pound [8], version 2.4.3, as a load balancer.

5.2 Client

As explained in Section 4.4, the Client component's responsibilities include gathering CPU utilization statistics from the containers and the hardware node, and executing relocations based on instructions from the Server. This Section presents the implementation details needed to support the functionality described in Section 4.4 and

explains the reasoning behind each method.

5.2.1 Gathering CPU Utilization Statistics

For gathering CPU utilization statistics from the containers, the Sensor module uses two methods: `gatherCtCpuStats` (see Listing A.1) and `calculateCtCpuUsage` (see Listing A.2). The first method reads the file `/proc/vz/vestat` (see Figure 5.1 for a sample file), which is generated by OpenVZ, and creates a ContainerData instance for each container whose information is available in the file. For each container, the values in the columns **uptime** (column 8) and **used** (column 9) are read and stored in the corresponding ContainerData instance. Those two values are expressed in *CPU cycles.*

The second method, `calculateCtCpuUsage`, implements the procedure described in Subsection 4.4.3. It first calls `gatherCtCpuStats` to obtain current observations of CPU utilization. Those values are used in conjunction with the values gathered in the previous iteration of the method and a ratio is calculated with the differences between current and previous observations. The ratio is stored in the corresponding ContainerData instance and the set of ContainerData instances created in the current iteration are stored so that the **uptime** and **used** values can be used as previous observations in the next iteration.

The method `calculateHostCpuUsage` (see Listing A.3) is used to obtain the hardware node's CPU utilization. It follows the same approach that was used for the containers. It reads the file `/proc/stat` (see Listing 5.1 for a sample file), which is generated by the Linux kernel, to obtain the values in the first row (the one that begins with the word *cpu*), each of which correspond to the time spent by the system in a given state since the start of the system. The values are expressed in *jiffies*, that is, $1/100^{ths}$ of a second.

Two sums are calculated from the current observation: *non_idle_time* and *total_time*. The same sums are calculated from the previous observation. With those values, a ratio is calculated with the differences between current and previous obser-

```
VEID  user  nice  system  uptime  idle  strv  uptime  used  maxlat  totlat  numsched
1894  2882  0  3316  660633861  4478263411430844  0  2240511291361515  79062399851  0  0  0
1893  2771  0  3368  660660990  4481046656491778  0  2240603297978835  80525407764  0  0  0
1892  2902  0  3281  660684309  4395767584110638  0  2240682382830654  79591168718  0  0  0
1891  2814  0  3322  660714346  4385662089836350  0  2240784248773261  79998000614  0  0  0
```

Figure 5.1: Sample /proc/vz/vestat file

Listing 5.1: Sample /proc/stat file

```
# cpu  user  nice  system  idle  iowait  irq  softirq  steal
cpu  1260299 250411 138414 322144736 49790 19907 16054 0
cpu0 639009 125893 61055 161088842 23544 6 1330 0
cpu1 621289 124518 77358 161055894 26245 19901 14724 0
intr 1647568313 1619569739 117 0 0 4 0 0 0 3 0 0 0 3287 0 14523922 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    809769 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10957728 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 1703744 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
swap 0 0
ctxt 400189460
btime 1243456181
processes 581830
procs_running 1
procs_blocked 0
```

vations. The set of values obtained in the current iteration are stored to be used as the previous observation in the next iteration.

5.2.2 Collecting Miscellaneous Information

The Sensor module also collects miscellaneous information that includes the hardware node's hostname and number of cores of the CPU. This is achieved through the method `getHostInfo` (see Listing A.4). The hostname is obtained issuing a call to the underlying operating system and the number of cores is obtained parsing the file `/proc/stat` (there is an entry for each core at the beginning of the file). The reader can refer to Listing 5.1 for a sample file.

In order to obtain the hostnames and IP addresses of the containers, the method `getCtsInfo` (see Listing A.5) is used. It parses the file `/etc/vz/conf/xyz.conf` (where xyz is the **CTID** of a container) and reads the relevant entries. The reader can refer to Listing 2.1 for a sample configuration file.

Listing 5.2: Replication procedure

```
1. Generate replica's CTID
2. Obtain CT's image stored in central repository
3. Process image
4. Edit image's configuration file
5. Start replica
```

5.2.3 Executing Relocations

The Actuator module in the Client possesses two methods for executing relocations: `migrateCt` (see Listing A.6) and `replicateCt` (see Listing A.7). The first method, executed in the node source of the relocation, issues a call to the underlying operating system (in fact, to the OpenVZ management tools) indicating which container to migrate and to which hardware node. The reader can refer to Section 2.2 for a description of the live migration process in OpenVZ.

The `replicateCt` method, executed in the node target of the relocation, implements a sequence of actions that provides replication (see Listing 5.2). First, the **CTID** for the replica has to be created. For that purpose, the following principle is followed: *"The **CTID** of a replica will be the **CTID** of the original container preceded by a 9."* If the **CTID** of the original container is *xyz*, the **CTID** of the replica will be *9xyz*. If the container to replicate is itself a replica, its **CTID** will already be *9xyz*. The **CTID** of the replica of a replica will also be *9xyz*.

After determining the new **CTID**, a call to the underlying operating system is issued for the operating system to request from a central repository an image for the given container. Once the image is brought from the remote server, the image is decompressed and moved to its proper place in the hardware node's file system. The configuration file that comes with the image of the container is moved also to its proper destination and later on updated with the new **CTID** and **IP**. Finally, another call is issued to the operating system to start the replica.

Configurable System Constants	
Constant	Purpose
window	Number of statistics to store for each container
k	Number of positive resource stress checks required to flag a resource stress situation
n	Total number of resource stress checks to remember

Table 5.1: Configurable System Constants

5.3 Server

The Server component has three responsibilities: receive and process Clients' statistics reports, search for resource stress situations, and find sequences of relocations upon detection of resource stress situations (see Section 4.5 for a detailed description of the component). This Section details how these tasks are achieved. Subsection 5.3.1 describes the data storage modules HardwareNode and Container, and the remaining Subsections focus each on one of the functionalities of the Server component.

5.3.1 Data Storage Modules

The Server component uses the Container and HardwareNode modules to maintain system status information. Each of the following Subsections focus on a module.

5.3.1.1 Container

The Container module stores information about an actual container running in the system - or a privileged container (see Section 2.2 for a distinction between containers and privileged containers). The Container module stores each container's CPU utilization statistics, and builds the mathematical model for predicting the next CPU utilization value (method `updateCpuModel`). When the Container module represents a privileged container, the CPU statistics that are stored belong to the host system, that is, the hosting hardware node. (Tables 5.2 and 5.3 summarize respectively the attributes and methods of a Container module. Table 5.1 summarizes the system constants used by the Container and HardwareNode modules.)

Container Module	
Attribute	Purpose
ctid	Identification number
hn	Reference to the hosting hardware node
replicas	Set of replicas of this container
migrating	Status flag
cpu	List of CPU utilization statistics
timestamp	Timestamp of the last statistics report
recCnt	Number of statistics reports received
CPUtheta	Parameter for the mathematical model
CPUmu	Parameter for the mathematical model

Table 5.2: Container module attributes

Container Module	
Method	Purpose
`init`	Initialization of the Container module
`update`	Stores new CPU utilization statistic
`updateCpuModel`	Updates the parameters of the mathematical model
`cpu`	Calculates the predicted CPU utilization
`profileCpu`	Calculates the profiled CPU utilization

Table 5.3: Container module methods

The `init` method (see Listing A.8) initializes the attributes of a Container module during creation. In addition to the identification key **ctid**, the Container module has the reference **hn** to the HardwareNode module that represents the physical hardware node where the actual container is hosted. The Container module also maintains a set of references to all existing replicas of itself (every container is considered a replica of itself, so the set is never empty), so all the replicas have knowledge of all the other replicas. The attribute **cpu** is a **window**-sized list of CPU utilization statistics received in a period of time and **timestamp** serves to indicate when the last statistic was received.

Every time a new CPU utilization statistic needs to be stored, the method `update` (see Listing A.9) is called. This method implements the concept of a *sliding window*. Thus, this method drops the oldest statistic when a new one is available. When the method is called on a Container module that stores statistics for a hardware node (that is, a Container module representing a privileged container), the statistics are multiplied by the number of cores in that hardware node. The result of this procedure is that when a hardware node with two cores reports using 50% of its computing power, the statistic that gets stored is 100%, effectively registering that the computing power of a whole CPU is being used (two times 50%).

The CPU utilization prediction, as described in Subsection 4.5.3, involves two methods from the Container module: `updateCpuModel` (see Listing A.10) and `cpu` (see Listing A.11). The first method updates the parameters of the mathematical model, μ and θ, with the statistics received since the last update. The values of the parameters are stored in the Container module attributes **CPUtheta** and **CPUmu**. The method `cpu` calculates the predicted CPU utilization.

The Container module also provides a method to calculate the profiled CPU utilization of a container. As explained in Subsection 4.5.3, the method `profileCpu` (see Listing A.12) sorts the collected CPU utilization statistics of the container and takes the value corresponding to the 90^{th} percentile. Then, the method returns the maximum between the latter value and the container's next predicted CPU utilization plus a Δ equal to 5%.

HardwareNode Module	
Attribute	Purpose
hnid	Identification name
cores	Number of CPU cores in the physical hardware node
pipe	Communication endpoint with Client component in the physical hardware node
ct0	Privileged container of the hardware node
containers	Set of containers hosted in the hardware node
maxCpu	Total CPU capacity of the hardware node
migCpu	Amount of CPU that the hardware node is freeing or allocating through ongoing relocations
overChecks	Set of n flags used for the resource stress check
overCnt	Last position used in the circular queue overChecks
migrating	Number of ongoing relocations

Table 5.4: HardwareNode module attributes

5.3.1.2 HardwareNode

The HardwareNode module represents a physical hardware node. It maintains a list of Container modules that represents the containers hosted by the physical hardware node. The module also provides methods for determining potential resource stress situations, for evaluating the feasibility of relocations (checking if there are enough spare resources to host an additional container), and for updating the system status during and after relocations. (Tables 5.4 and 5.5 summarize respectively the attributes and methods of a HardwareNode module.)

The `init` method (see Listing A.13) shows the initialization of a HardwareNode module during creation. The HardwareNode module receives by parameter its identification **hnid**, the number of CPU **cores** in the physical hardware node, and the communication endpoint (**pipe**) with the Client component running in the physical hardware node. The HardwareNode module uses a Container module (**ct0**) to represent the privileged container of the physical hardware node. This Container module stores the hardware node's CPU utilization statistics and provides the methods to predict the hardware node's next CPU utilization. **migrating** indicates the number of ongoing relocations (that is, relocations that have not been completed) in which

HardwareNode Module	
Method	Purpose
`init`	Initialization of the HardwareNode module
`overloaded`	Executes the resource stress check
`willFitRep`	Determines whether the HardwareNode can be target of a given replication
`willFitMig`	Determines whether the HardwareNode can be target of a given migration
`recordMigration`	Registers a new migration and starts the migration process
`recordReplication`	Registers a new replication and starts the replication process
`migCompleted`	Completes the migration process
`repCompleted`	Completes the replication process

Table 5.5: HardwareNode module methods

the hardware node is involved and **migCpu** records the total amount of CPU that the hardware node is freeing or allocating as a consequence of the ongoing relocations. **overChecks** is a set of **n** flags used during the resource stress check.

The method `overloaded` (see Listing A.14) implements the resource stress check that the Server component runs periodically for every hardware node. As explained in Section 4.5, this method checks if the predicted CPU utilization of the hardware node exceeds a given threshold and if **k** out of the **n** flags in **overChecks** have been set during previous iterations of the method. **overChecks** is updated in every call to `overloaded` and works as a circular queue in order to always account for the last **n** checks.

Another method provided by the HardwareNode module is the check done on non-stressed hardware nodes to determine whether the hardware node can host an additional container. This check is done during the relocation process and it is implemented using the `willFitRep` method (see Listing A.15). For a hardware node to be chosen to host a container, the sum of the hardware node's next predicted CPU utilization, the value **migCpu** and the container's profiled CPU utilization has to remain below the CPU utilization threshold.

The value **migCpu** accounts for the amount of CPU that has already been promised to other containers that have been selected to be relocated to the hard-

ware node. The container's profiled CPU utilization is modified by the factor **share** ($0 < share <= 1$), which reduces the value to a certain percentage of its original value. For replications, **share** equals .6 since the system assumes that doing load balancing between the original container and the replica will result in each container requiring at most 60% of the profiled CPU utilization. (A perfect load balancing between two containers would distribute the load in 50% for each container. Golondrina allocates 60% instead of 50% to account for potential overheads.)

The HardwareNode module also provides the method `willFitMig`. This method is used when the container is to be migrated and the difference with `willFitRep` is that **share** is equal to 1.

In order to keep track of the ongoing relocations, the HardwareNode module provides the methods `recordMigration` (see Listing A.16), `recordReplication` (see Listing A.17), `migCompleted` (see Listing A.18) and `repCompleted` (see Listing A.19). The method `recordMigration` is called on a hardware node that has been selected as source or target for a migration. It first increases the counter **migrating** to indicate that the hardware node is involved in one additional relocation and it records in **migCpu** the amount of CPU that will be freed (if the hardware node is source of the migration) or allocated (if the hardware node is target of the migration). The method also modifies the status of the container to be migrated to indicate that it is going to be migrated, and the method adds the container to the list of containers in the target hardware node.

The method `recordReplication` accomplishes a similar task to that of the method `recordMigration`. The difference is that the status of the container to replicate is not modified and the amount of CPU recorded in **migCpu** is modified by the factor **share**.

The methods `migCompleted` and `repCompleted` are the counterparts of the methods `recordMigration` and `recordReplication`, respectively. `migCompleted` is called after the completion of a migration. It first decreases the counter **migrating** to indicate that a relocation in which the hardware node was involved has finished and to reinitialize **migCpu** to zero (only if that was the last relocation to be completed).

The method also removes the migrated container from the list of containers associated with the source hardware node, sets the **overChecks** set of flags to **k**-1, and updates the container's status to indicate that the migration has been completed.

The method `repCompleted` is called after a replication has been completed. The difference with the method `migCompleted` is that `repCompleted` does not remove the container to be replicated from the source hardware node's list of containers, nor does it modify the container's status.

5.3.2 Processing Statistics Reports

As explained in Subsection 4.5.1, the Register module is responsible for processing the Clients' statistics reports. The method `addNewObservation` (see Listing A.20) is invoked with every new report to store the latest CPU utilization observation from a container or hardware node. If the container or hardware node to which the observation belongs is not yet registered in the system, the corresponding module (Container or HardwareNode) is created and inserted in its proper place in the list of hardware nodes **hns**.

If the statistics report belongs to a container with **ctid** equal to 0, that means that the report actually belongs to the hardware node and the CPU utilization observation has to be stored in **ct0**.

If the container's **ctid** is not 0, the method tries to find the container in the list of containers of the hardware node with **hnid**. If the container is found, the method checks if the container had been relocated (migration or replication of a replica) and the current observation is the first one done at the target hardware node. If the latter is correct, the relocation process is completed.

If the container was not found in the hardware node's list of containers, the method checks again if the statistics report belongs to a replica (replication of an original container). If the latter is correct, the replication process is completed. If it is not the case, then the report belongs to a new container and the corresponding Container module is created.

5.3.3 Searching for Resource Stress Situations

For this purpose, the method `monitor` (see Listing A.21) is used, which calls the method `overloaded` on every hardware node that is not currently involved in a relocation. The method also looks for containers or hardware nodes from which no statistics reports have been received lately[1].

If a hardware node is already involved in a relocation (be it as source or target), no resource stress check is done on it. The reason is that if the hardware node is source of a relocation, it is because it was found under a resource stress situation in a previous iteration of `monitor` and measures were taken to control the situation. If the hardware node is a target for a relocation, its load (understood as the amount of CPU used by the hosted containers) is likely to increase soon, so no additional relocations are to be considered until all current relocations are completed.

The hardware nodes that are not involved in relocations are checked for resource stress and categorized as stressed or non-stressed hardware nodes. Sequences of relocations are only searched for if there are stressed hardware nodes and also non-stressed hardware nodes that could serve as relocation targets.

5.3.4 Finding Relocations

Every time a resource stress situation is detected in a hardware node and there are hardware nodes available to host additional containers, the Relocator module is invoked. Its responsibility is to determine which containers hosted in stressed hardware nodes will be relocated and which non-stressed hardware nodes will serve as target for those relocations.

When the mechanism to apply is migration, the method `migrations` (see Listing A.22) is called. The method first sorts the resource stressed hardware nodes in decreasing order of their *load* (that is, the proportion of CPU that is being used) and the non-stressed hardware nodes in increasing order of their load. The method then analyzes each stressed hardware node at a time, starting with the most stressed

[1] Currently, no action is taken against containers or hardware nodes that have reached their timeout.

one. It sorts the hosted containers with the `decreasingLoadPolicy` method (see Listing A.23) and tries to relocate each container until the resource stress situation is dissipated. For each container, the method cycles through the list of non-stressed hardware nodes trying to find a target for the relocation.

The method `migrations` calls the method `decreasingLoadPolicy` on the list of containers of every resource stressed hardware node that it analyzes. The invoked process removes from the list those containers whose CPU utilization is below a certain threshold and returns the remaining containers sorted in decreasing order of their load. Essentially, the policy is relocate the most loaded containers first and never relocate the lightly loaded containers.

When the mechanism to apply is replication, the method `replications` (see Listing A.25) is called by the Relocator module. The only difference with the `migrations` method is that when it queries a non-stressed hardware node to see if it can accept an additional container, the method also checks that the hardware node does not already host a replica of that same container. This additional check is done invoking the method `twoReplicasInHnPolicy` (see Listing A.24).

5.4 Gate

As explained in Section 4.6, the Gate component has the responsibility of updating the load balancer's configuration when the Server component instructs so. This Section provides details on how that task is achieved.

5.4.1 Updating the Load Balancer's Configuration

The Actuator module is called by the Gate module when a message is received from the Server component requesting that the load balancer's configuration be updated. The Server component always calls after a replication has been completed, in order for the load balancer to know that incoming requests for a certain **hostname** can now be sent to an additional **ip**. Since migrations do not involve new or additional IPs, they do not require an update of the load balancer.

Listing 5.3: Sample Pound configuration file

```
ListenHTTP
        Address 129.100.18.67
        Port 80
        Service
                HeadRequire "Host:.*one.com.*"
                BackEnd
                        Address 129.100.18.91
                        Port 80
                End
        End
        Service
                HeadRequire "Host:.*two.com.*"
                BackEnd
                        Address 129.100.18.92
                        Port 80
                End
                BackEnd
                        Address 129.100.18.95
                        Port 80
                End
        End
End
```

The method `addToProxy` (see Listing A.26) reads the file `/etc/pound.cfg` (see Listing 5.3 for a sample configuration file) and searches for the Service entry of the given **hostname**. Once found, the method adds a new BackEnd (the given **ip**) to the Service entry and it restarts the load balancer.

5.5 Summary

In this chapter, details about Golondrina's implementation were presented. The methods that implement the system's functionality were explained and references to the corresponding Listings in Appendix A added. Chapter 4 was cross-referenced at several points to make clear how different parts of the architecture were implemented.

Chapter 6

Experiments

This chapter presents the evaluation of the resource management system Golondrina. Section 6.1 defines the objectives, strategies and metrics of the evaluation process. Section 6.2 briefly describes the configurations used. Section 6.3 presents the results of the experiments.

6.1 Evaluation

As it was explained in Section 4.1, the environment in which Golondrina runs consists of a cluster where one physical server is the *gate of the cluster*, another physical server is a *manager server* and the rest of the physical servers are *OpenVZ hardware nodes.*

Each physical server is an Intel Pentium D 3.40GHz (dual-core) with two GBytes of RAM. The containers were built with the default resource allocation provided by OpenVZ (see Listing 2.1 for a sample configuration file).

The objective of the experiments was to study the reaction of the system to resource stress situations in one or more hardware nodes. For that purpose, each container executed an Apache web server [9]. Load was generated by running httperf [10] on several physical servers in the cluster that were not part of the managed system. httperf was used to generate HTTP requests.

The HTTP requests requested dynamic content. The file `molinosQuijote.php` (see Listing 6.1) was executed with each request in order to count the number of words in the two MBytes text file `donQuijote` [11]. The execution returned a HTML

Listing 6.1: PHP file molinosQuijote.php

```
<html>
<head>
<title>Word _molinos_ in Don Quijote</title>
</head>
<body>
<?php
$filename = "donQuijote";
$word = "molinos";
if ($fh = fopen($filename, "r")) {
        $total = 0;
        $count = 0;
        while (!feof($fh)) {
                $line = fgets($fh);
                $total += str_word_count($line, 0);
                $array = explode(" ", $line);
                for ( $i = 0; $i < count($array); $i++) {
                        if ( strcasecmp($array[$i], $word) == 0 ) {
                                $count++;
                        }
                }
        }
        fclose($fh);
        echo "The word '$word' appears $count times in the text
            $filename.\n";
        echo "The total number of words in the text is $total.\n";
}
else {
        echo "Could not open file: $filename";
}
?>
</body>
</html>
```

file with the results of the counting. The reason for requesting dynamic content was to increase CPU utilization.

The frequency with which requests were sent determined the *weight* of the generated load. The weight of the load is the percentage of CPU cycles that are required from one CPU core to handle that load. For example, sending 1 request per second (1 req/sec) resulted in a CPU core being used at 70% capacity.

The metrics used to evaluate the system included lost requests and performance of the web servers. A baseline was established where the performance of each web server and the number of lost requests were monitored for varying loads, including loads that caused resource stress. The same loads were applied with Golondrina in

place for each different set of policies.

The requests were classified into three categories: lost, failed and successful. Lost requests were those not processed before a client timeout (**client-timo**). Failed requests were those where the server refused a connection (**connrefused**), sent a RESET (**connreset**) or replied with a Server Error 5xx status code (**reply-status-5xx**). A server's effectiveness was defined as the ratio of the number of successful requests to the total requests generated.

The web servers' performance was measured as the minimum (**min**), average (**avg**) and maximum (**max**) values corresponding to the duration of the established connections (for sending a request, a connection is established between the client and the server, and once the reply is received, the connection is terminated), measured in milliseconds.

6.2 Experiments and Configurations

Three different experiments were designed to evaluate Golondrina. Each experiment used the same number of hardware nodes, but the number of containers and the weight of the loads varied.

Each experiment was run three times. For the first run, Golondrina was configured to monitor the resource utilization and check for resource stress situations. The relocation mechanisms were disabled. This run provided a baseline, enabling observations on how the environment performed without Golondrina taking corrective actions.

In the second and third runs, Golondrina was configured to use replications and migrations, respectively. The results of these runs were compared with each other and against the baseline.

Golondrina's resource stress detection mechanism was configured in all cases with a CPU utilization threshold of 0.75. Given that the physical servers possessed two cores, the threshold was equivalent to 150% of the total CPU capacity. (The total CPU capacity of a physical server with x cores is equal to $x * 100\%$. In this case,

the total CPU capacity of the hardware nodes was 200%.) The mechanism was also configured to trigger the resource stress check every 10 seconds.

The HTTP requests sent to the web servers had an associated timeout of 10 seconds. The time span between the start of two different loads was 60 seconds.

6.2.1 Experiment 1

In the first experiment, the managed system consisted of two hardware nodes, **bravo02** and **bravo03**, and two containers, **1891** and **1892**, hosted in **bravo02**. At a given point in time, the container **1891** received a load of around 70% (450 requests at a rate of 1 req/sec). After 60 seconds, the container **1892** received a load of around 105% (450 requests at a rate of 1.5 req/sec). At this point in time, the hardware node **bravo02** experienced a load of around 175%, which exceeded the CPU utilization threshold of 150%. Thus, **bravo02** was under a resource stress situation.

In this scenario, no request would be lost since the CPU needs of both containers could be satisfied. However, Golondrina would determine that **bravo02** was under a resource stress situation and would try to address this situation through a relocation. Golondrina should have tried first to relocate to **bravo03** the container with the highest load, that is, **1892**.

6.2.2 Experiment 2

The second experiment was similar to the previous one with the exception that both containers received a load of around 105% (450 requests at a rate of 1.5 req/sec) each. As a consequence, the hardware node **bravo02** was resource stressed with a load of 200% (total CPU capacity).

In this scenario, the web servers running in **1891** and **1892** would have lost requests, due to the lack of spare CPU cycles to allocate to the containers. Golondrina would detect the resource stress situation experienced by **bravo02** and would respond by triggering a relocation for the container with the highest load.

6.2.3 Experiment 3

In the third experiment, the managed system consisted of two hardware nodes, **bravo02** and **bravo03**, and four containers, **1891**, **1892**, **1893** and **1894**, hosted in **bravo02**. One after another, with a 60-second separation in time, the containers received a load of around 51% (300 requests at a rate of 0.72 req/sec). Thus, **bravo02** experienced a load of 200%.

In this scenario, the web servers hosted in the four containers would have lost requests. Golondrina would detect the resource stress situation and determine relocations in order to dissipate it. Three replications or two migrations should have been enough to terminate the resource stress situation.

6.2.4 Policies

As explained in Subsection 5.3.4, the relocation process uses several policies during its search for a sequence of relocations. The policy to select source hardware nodes for the relocations is *"solve the resource stress situation of the highest loaded hardware node first."* The policy to find target hardware nodes for the relocations is *"consider the least loaded hardware nodes first. If the relocation would cause a resource stress situation in the target hardware node, do not do the relocation."*

The policy used for choosing a container to be relocated is *"choose the most loaded container first. Do not relocate lightly loaded containers."*

To find target hardware nodes for replications, an additional policy is used: *"no hardware node can host two replicas of the same container."*

6.3 Results

This section presents the results of the experiments described in Section 6.2.

System statistics were printed by Golondrina immediately after each resource stress check (that is, every 10 seconds).

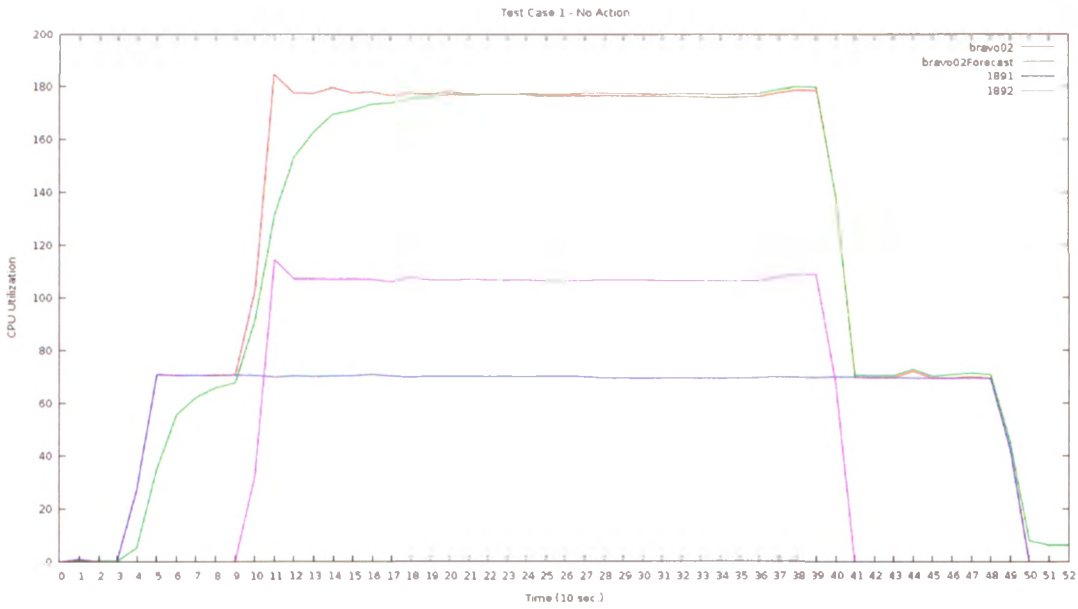

Figure 6.1: Experiment 1 - No Action

6.3.1 Experiment 1

Run 1: In the first run of the experiment, Golondrina was monitoring the resource utilization, but no action was taken in response to a resource stress situation. Figure 6.1 shows the resource utilization of the containers **1891** and **1892**, and the resource utilization and predicted CPU utilization (as explained in Subsection 4.5.3) of the hardware node **bravo02**.

The first time the resource utilization of **bravo02** went over the 150% threshold was at $t = 11$. Golondrina's resource stress detection mechanism signaled the problem at $t = 15$. Since no action was taken, the resource stress situation persisted and was signaled every single time until $t = 39$ (included).

Since there were enough resources to satisfy the demand of the containers, no request was lost or failed.

The web server **one.com**, hosted in **1891**, had connection times $min = 687.5$, $avg = 701.9$ and $max = 979.9$ milliseconds. The web server **two.com**, hosted in **1892**, had connection times $min = 724.0$, $avg = 904.5$ and $max = 7160.2$ millisec-

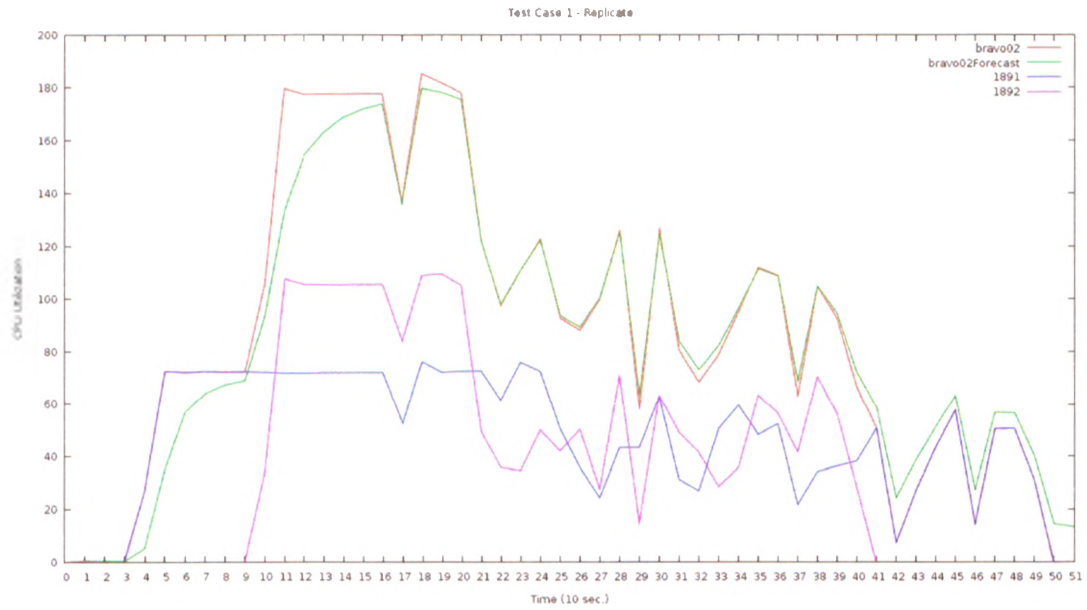

Figure 6.2: Experiment 1 - Replicate, bravo02

onds.

Run 2: In the second run of the experiment, Golondrina was to search for possible replications if a resource stress situation was detected. Figure 6.2 shows the resource utilization of the containers **1891** and **1892**, and the resource utilization and predicted CPU utilization of the hardware node **bravo02**. Figure 6.3 shows the resource utilization of the replicas **91891** and **91892**, and the resource utilization and predicted CPU utilization of **bravo03**.

The first resource stress situation in **bravo02** was signaled at $t = 15$. At that time, Golondrina determined that the container **1892** had to be replicated in **bravo03**. By $t = 16$ the replica **91892** had been created and the load balancer at the *gate of the cluster* was updated. At $t = 17$, **91892** had CPU load, but then it did not process any request for three consecutive periods. As a consequence of container **91892** not receiving any load, the resource stress situation persisted in **bravo02** and a second resource stress situation was signaled at $t = 19$. This time container **1891** was replicated in **bravo03**. It could be said then that the creation of container **91891** took place due to an improper balancing of the load for the web server **two.com**,

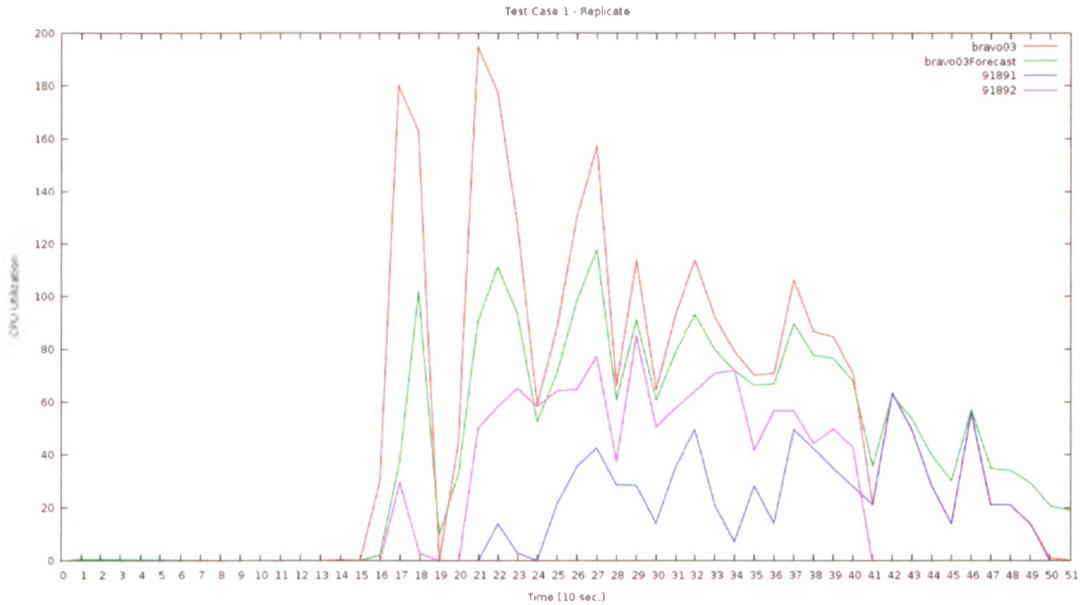

Figure 6.3: Experiment 1 - Replicate, bravo03

hosted in **1892** and **91892**.

At $t = 17$ and $t = 22$ it can be seen in Figure 6.2 and Figure 6.3 that the curves sloped down. During the periods $(16, 17)$ and $(21, 22)$ the load balancer was being updated, that required it to be restarted. As a consequence, some connections were refused or reset, and hence there was a slight decrease in the reported load.

The web server **one.com**, hosted in **1891** and **91891**, had 4 failed requests out of 450 (connrefused 3 connreset 1), which resulted in an effectiveness of 99.11%. The web server **two.com**, hosted in **1892** and **91892**, had 7 failed requests out of 450 (connrefused 5 connreset 2), which resulted in an effectiveness of 98.44%.

Compared with the 100% effectiveness of both web servers in the first run (when Golondrina would take no action against a resource stress situation), triggering replications does not seem convenient since some requests were lost. However, if the load balancer could be updated without requiring a restart, no requests would be lost (this issue is further discussed in Section 7.2).

The web server **one.com** had connection times $min = 695.2$, $avg = 749.1$ and $max = 3655.0$ milliseconds. The web server **two.com** had connection times $min =$

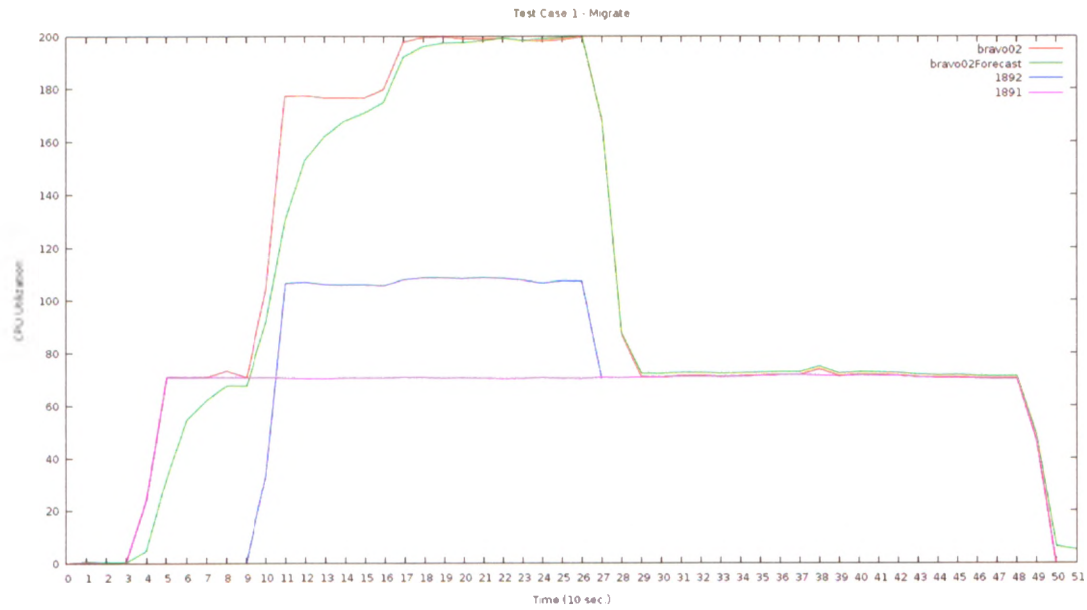

Figure 6.4: Experiment 1 - Migrate, bravo02

692.0, $avg = 853.0$ and $max = 8185.6$ milliseconds.

In comparison with the web server's performance in the first run, **one.com** increased the average connection time by about 6.72% and **two.com** decreased the average connection time by about 5.69%.

Run 3: In the third run of the experiment, Golondrina was to look for migrations upon detection of a resource stress situation. Figure 6.4 shows the resource utilization of the containers **1891** and **1892**, and the resource utilization and predicted CPU utilization of the hardware node **bravo02**. Figure 6.5 shows the resource utilization of **1892**, and the resource utilization and predicted CPU utilization of **bravo03**.

A resource stress situation was signaled at $t = 15$ in **bravo02**. Golondrina determined that the container **1892** was to be migrated to **bravo03**. At that point, the CPU utilization of both hardware nodes increased, due to the start of the migration process. Since there was spare CPU capacity in both hardware nodes, the containers saw their CPU needs unaffected.

In the period $(26, 27)$, the migration process was completed, but it was not until $t = 28$ that a CPU utilization report from the container **1892** was sent to the Server

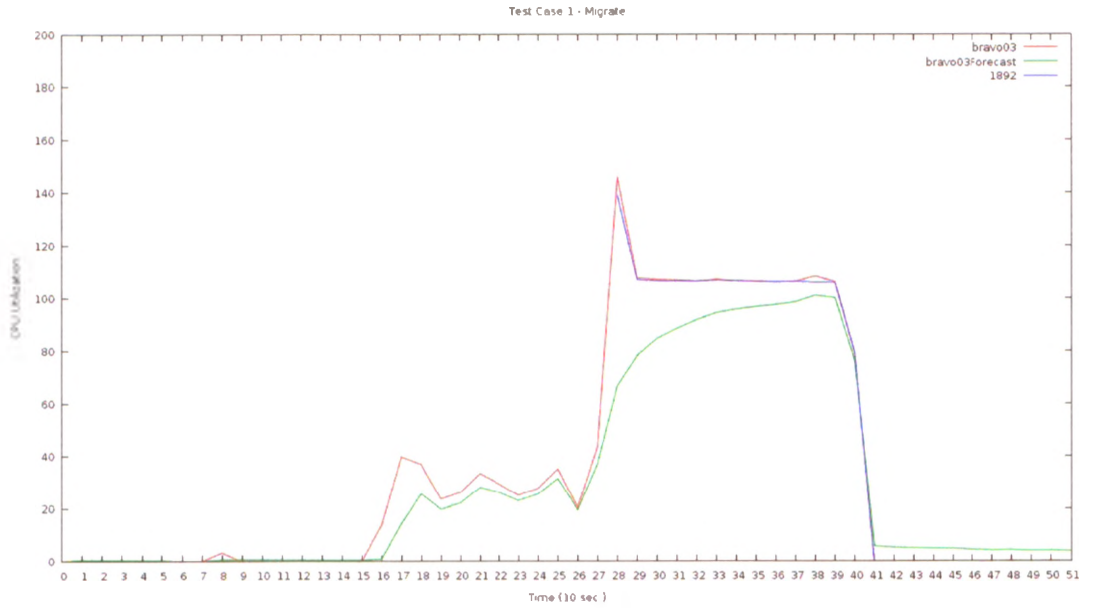

Figure 6.5: Experiment 1 - Migrate, bravo03

component (running in the *manager server*) by the Client component running in **bravo03**. That report showed a peak of around 140% in CPU utilization, which could be attributed to the hosted web server processing the requests that could not be handled during the suspension period of the migration process.

None of the web servers hosted in **1891** and **1892** had lost or failed requests. That means that the web servers had an effectiveness of 100%. Therefore, it cannot be stated that migration results in an improvement over taking no actions, but at least it does not perform worse. When compared with replication, migration provides better results. However, this only happens due to the load balancer requiring a restart when updating its configuration after a relocation.

Web Servers' Effectiveness			
Servers	Run 1	Run 2	Run 3
one.com	100%	99.11%	100%
two.com	100%	98.44%	100%

Table 6.1: Experiment 1 - Percentage of successful requests.

Web Servers' Performance			
Servers	Run 1	Run 2	Run 3
one.com	701.9	749.1	715.2
two.com	904.5	853.0	991.1

Table 6.2: Experiment 1 - Web servers' average connection time in milliseconds.

The web server **one.com**, hosted in **1891**, had connection times $min = 698.6$, $avg = 715.2$ and $max = 875.4$ milliseconds. The web server **two.com**, hosted in **1892**, had connection times $min = 686.7$, $avg = 991.1$ and $max = 9700.8$ milliseconds.

In comparison with the web server's performance in the first run, **one.com** increased the average connection time by about 1.89% and **two.com** increased the average connection time by about 9.57%.

The comparison with the web server's performance in the second run does not offer conclusive results.

In conclusion, when a hardware node experiences a resource stress situation, but the CPU is not exhausted, no requests are lost. Thus, no action is necessary from the management system. However, migration and replication cause no (serious) performance degradation, so they could be used as preventive actions in case the load was expected to increase.

6.3.2 Experiment 2

Run 1: In the first run of the experiment, Golondrina was monitoring the resource utilization, but no action was taken in response to a resource stress situation. Figure 6.6 shows the resource utilization of the containers **1891** and **1892**, and the resource utilization and predicted CPU utilization (as explained in Subsection 4.5.3) of the hardware node **bravo02**.

The first time the resource utilization of **bravo02** went over the 150% threshold was at $t = 11$. Golondrina's resource stress detection mechanism signaled the problem at $t = 14$. Since no action was taken, the resource stress situation persisted and was

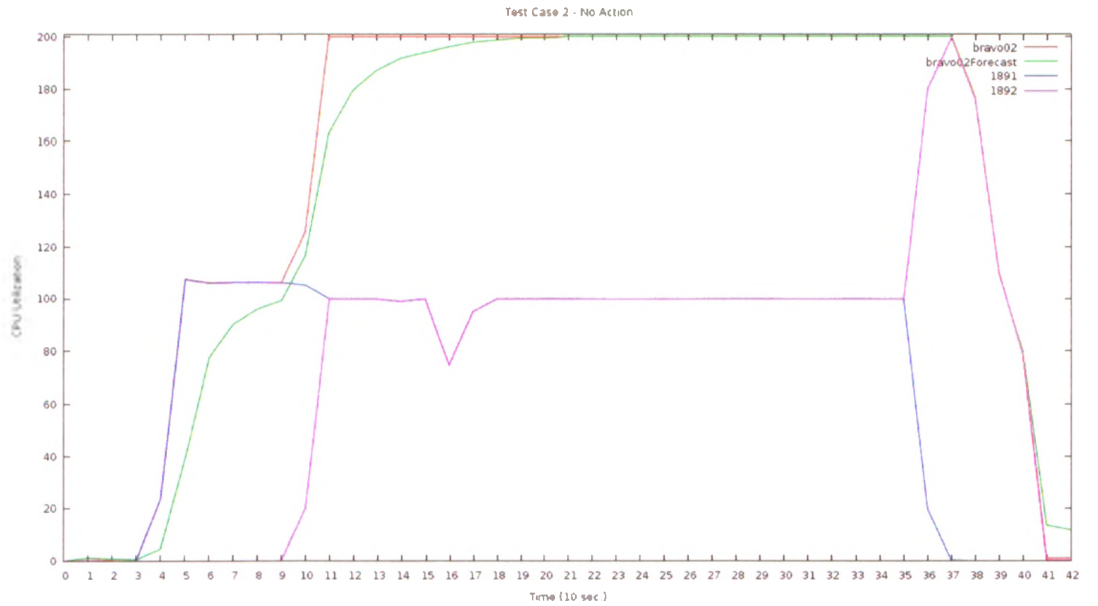

Figure 6.6: Experiment 2 - No Action

signaled every single time until $t = 38$ (included).

Starting at $t = 11$ the CPU was equally shared between the two containers, using almost 100% each. However, the number of CPU cycles allocated to each container was not enough for the hosted web servers to process all requests. The web server **one.com**, hosted in **1891**, had 101 lost requests out of 450 (client-timo 101), resulting in an effectiveness of 77.55%. The web server **two.com**, hosted in **1892**, had 169 lost requests out of 450 (client-timo 169), resulting in an effectiveness of 62.44%.

It can be seen in Figure 6.6 that during the interval $[36, 38]$ the container **1892** almost doubled its CPU utilization, taking advantage of container **1891** not requesting CPU cycles. This behaviour could be attributed to web server **two.com** processing all the requests that it had not been able to satisfy before due to a lack of CPU cycles.

The web server **one.com** had connection times $min = 695.1$, $avg = 2816.3$ and $max = 9995.3$ milliseconds. The web server **two.com** had connection times $min = 700.1$, $avg = 3371.6$ and $max = 9961.4$ milliseconds.

Run 2: In the second run of the experiment, Golondrina was to search for possible replications if a resource stress situation was detected. Figure 6.7 shows the

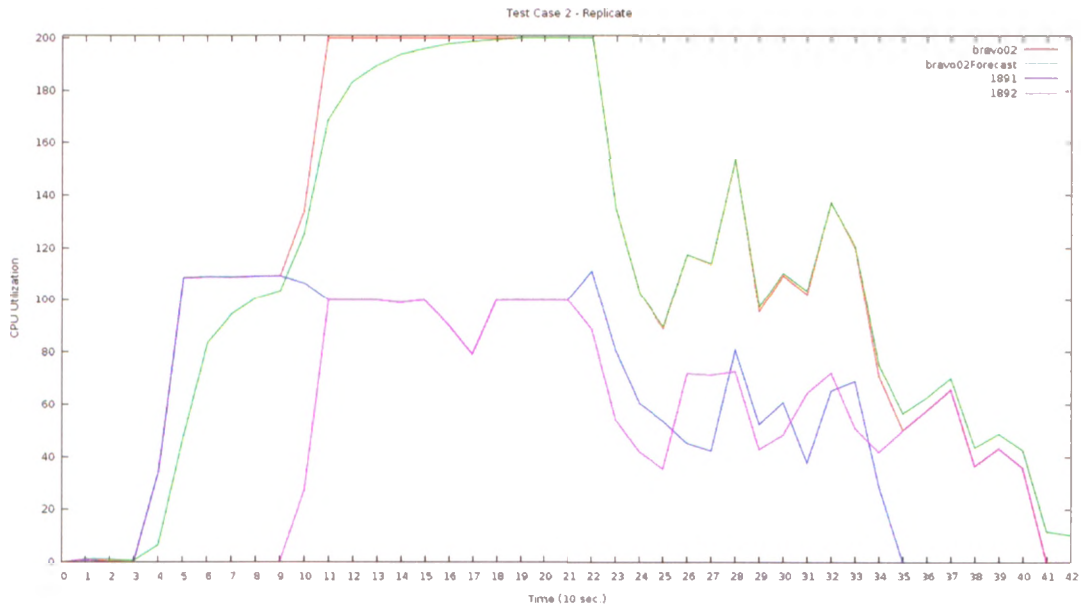

Figure 6.7: Experiment 2 - Replicate, bravo02

resource utilization of the containers **1891** and **1892**, and the resource utilization and predicted CPU utilization of the hardware node **bravo02**. Figure 6.8 shows the resource utilization of the replicas **91891** and **91892**, and the resource utilization and predicted CPU utilization of **bravo03**.

The first resource stress situation in **bravo02** was signaled at $t = 14$. Golondrina determined that both containers had to be replicated in **bravo03**. By $t = 16$ the replicas **91892** and **91891** had been created and the load balancer at the *gate of the cluster* was updated. The load balancer was first updated (and restarted) for **91892** in the period $(15, 16)$ and updated again in the period $(16, 17)$ for the container **91891**. During those two periods, it can be seen in Figure 6.7 that the CPU utilization of **1891** and **1892** decreased, due to connections that were refused or reset.

At $t = 17$, the containers **91891** and **91892** had a low CPU utilization and remained with minimal CPU utilization until $t = 20$ (included). During those periods, the requests were handled by **1891** and **1892**. As a consequence, the resource stress situation continued in **bravo02** and was signaled at $t = 18, 19, 20, 21, 22$. At every one of those five points in time, Golondrina could not find suitable replications since

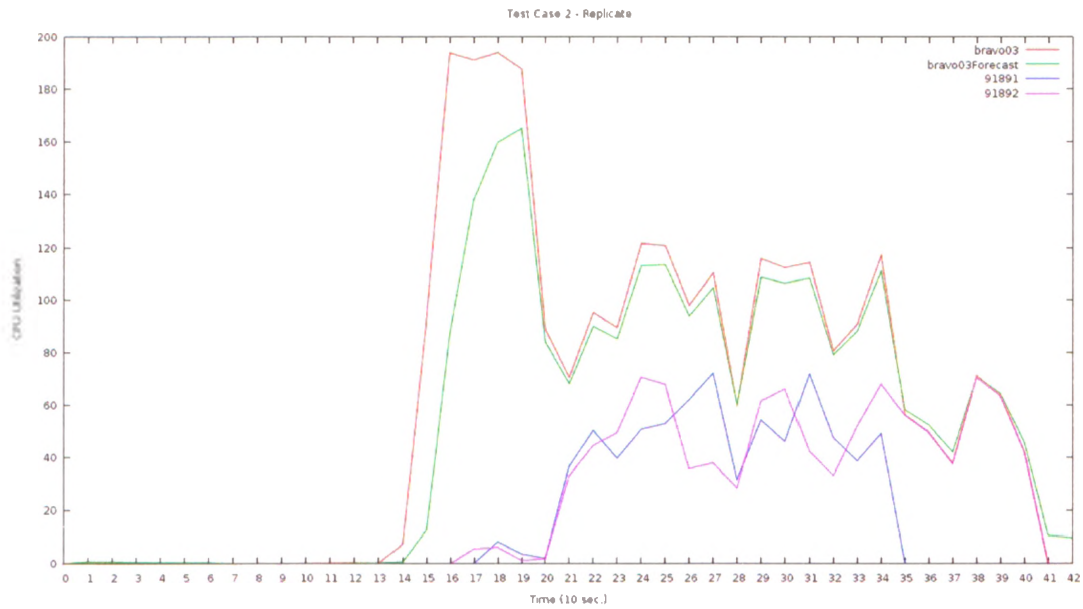

Figure 6.8: Experiment 2 - Replicate, bravo03

1891 and **1892** had already been replicated in **bravo03** and the system does not allow two replicas of the same container to reside in the same hardware node.

The web server **one.com**, hosted in **1891** and **91891**, had 21 failed requests (connrefused 4 connreset 17) and 35 lost requests (client-timo 35) out of 450, which resulted in an effectiveness of 87.55%. The web server **two.com**, hosted in **1892** and **91892**, had 25 failed requests (connrefused 4 connreset 21) and 29 lost requests (client-timo 29) out of 450, which resulted in an effectiveness of 88%.

Compared with the first run of the experiment, where Golondrina took no action upon a resource stress situation being detected, the replications helped improve the effectiveness of web server **one.com** by 10% and of web server **two.com** by 25.56%.

The web server **one.com** had connection times $min = 699.6$, $avg = 1408.1$ and $max = 9933.0$ milliseconds. The web server **two.com** had connection times $min = 695.4$, $avg = 1781.4$ and $max = 9558.6$ milliseconds.

In comparison with the web server's performance in the first run, **one.com** decreased the average connection time by about 50% and **two.com** decreased the average connection time by about 47.16%.

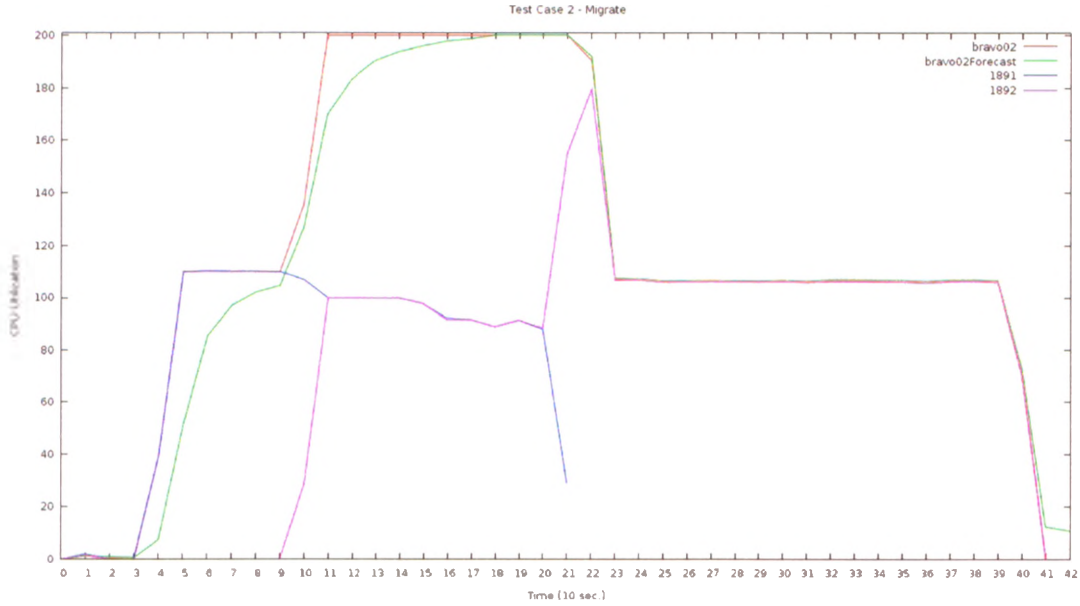

Figure 6.9: Experiment 2 - Migrate, bravo02

Run 3: In the third run of the experiment, Golondrina was to look for migrations upon detection of a resource stress situation. Figure 6.9 shows the resource utilization of the containers **1891** and **1892**, and the resource utilization and predicted CPU utilization of the hardware node **bravo02**. Figure 6.10 shows the resource utilization of **1891**, and the resource utilization and predicted CPU utilization of **bravo03**.

A resource stress situation was signaled at $t = 14$ in **bravo02**. Golondrina determined that the container **1891** was to be migrated to **bravo03**. The migration process was started, increasing the CPU utilization in **bravo03**. The CPU in **bravo02** was already exhausted, so the migration process competed for the CPU with the containers. **1891** and **1892** saw a reduction in their CPU allocation in the interval $(14, 20]$ until the migration process ended in the period $(20, 21)$.

In the period $[21, 22]$, the container **1892** increased its CPU utilization around 180%, and in the period $[24, 25]$ **1891** increased its CPU utilization around 195%. The behaviour of both containers could be attributed to the hosted web servers processing the requests that could not be handled during the migration process.

The web server **one.com**, hosted in **1891**, had 10 failed requests (reply-status-5xx

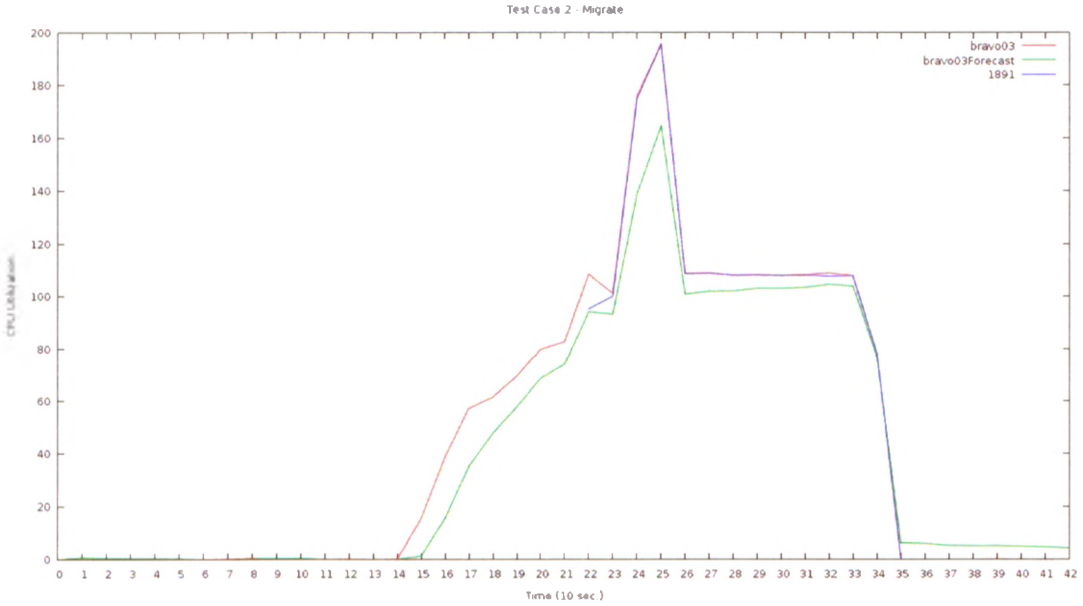

Figure 6.10: Experiment 2 - Migrate, bravo03

10) and 89 lost requests (client-timo 89) out of 450, which resulted in an effectiveness of 78%. The web server **two.com**, hosted in **1892**, had 70 lost requests (client-timo 70) out of 450, which resulted in an effectiveness of 84.44%.

Compared with the first run of the experiment, where Golondrina took no action upon a resource stress situation being detected, the migration helped improved the effectiveness of web server **one.com** by 0.45% and of web server **two.com** by 22%.

In comparison with the second run of the experiment, where Golondrina used the replication mechanism, migration fell short in the effectiveness improvement by 9.55% in the case of the web server **one.com** and by 3.56% in the case of the web server **two.com**.

Web Servers' Effectiveness			
Servers	Run 1	Run 2	Run 3
one.com	77.55%	87.55%	78%
two.com	62.44%	88%	84.44%

Table 6.3: Experiment 2 - Percentage of successful requests.

Web Servers' Performance			
Servers	Run 1	Run 2	Run 3
one.com	2816.3	1408.1	1723.6
two.com	3371.6	1781.4	1661.6

Table 6.4: Experiment 2 - Web servers' average connection time in milliseconds.

Migration showed that it offered an improvement over taking no action upon detection of a resource stress situation. However, the migration mechanism did not provide the same benefit as the replication mechanism.

The web server **one.com** had connection times $min = 0.4$, $avg = 1723.6$ and $max = 9950.2$ milliseconds. The web server **two.com** had connection times $min = 687.6$, $avg = 1661.6$ and $max = 9963.3$ milliseconds.

In comparison with the web server's performance in the first run, **one.com** decreased the average connection time by about 38.79% and **two.com** decreased the average connection time by about 50.71%.

The comparison with the web server's performance in the second run does not offer conclusive results.

In conclusion, when a hardware node is under a resource stress situation and the CPU is exhausted, some requests will not be satisfied. Both migration and replication represent a convenient solution, since they help to reduce the losses. However, the migration process competes with the containers for CPU cycles, diminishing the benefit it could provide.

6.3.3 Experiment 3

Run 1: In the first run of the experiment, Golondrina was monitoring the resource utilization, but no action was taken in response to a resource stress situation. Figure 6.11 shows the resource utilization of the containers **1891**, **1892**, **1893** and **1894**, and the resource utilization and predicted CPU utilization (as explained in Subsection 4.5.3) of the hardware node **bravo02**.

The first time the resource utilization of **bravo02** went over the 150% threshold

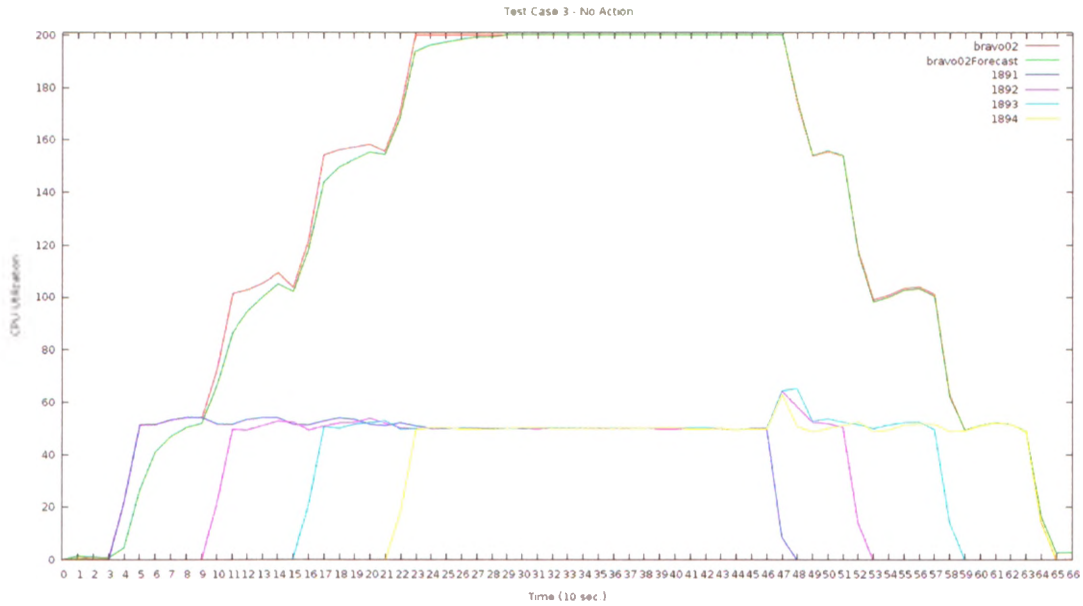

Figure 6.11: Experiment 3 - No Action

was at $t = 17$. Golondrina's resource stress detection mechanism signaled the problem at $t = 22$. Since no action was taken, the resource stress situation persisted and was signaled every single time until $t = 51$ (included).

Starting at $t = 23$ the CPU was equally shared between the four containers, using almost 50% each. However, the number of CPU cycles allocated to each container was not enough for the hosted web servers to process all requests. The web server **one.com**, hosted in **1891**, had 36 lost requests out of 300 (client-timo 36), resulting in an effectiveness of 88%. The web server **two.com**, hosted in **1892**, had 24 lost requests out of 300 (client-timo 24), resulting in an effectiveness of 92%. The web server **three.com**, hosted in **1893**, had 39 lost requests out of 300 (client-timo 39), resulting in an effectiveness of 87%. The web server **four.com**, hosted in **1894**, had 6 lost requests out of 300 (client-timo 6), resulting in an effectiveness of 98%.

It can be seen in Figure 6.11 that at $t = 47$, when **1891** saw a decrease in its CPU utilization, the remaining containers had a peak in their CPU utilization. This behaviour could be attributed to the hosted web server processing all the requests that could not be satisfied before due to a lack of CPU cycles.

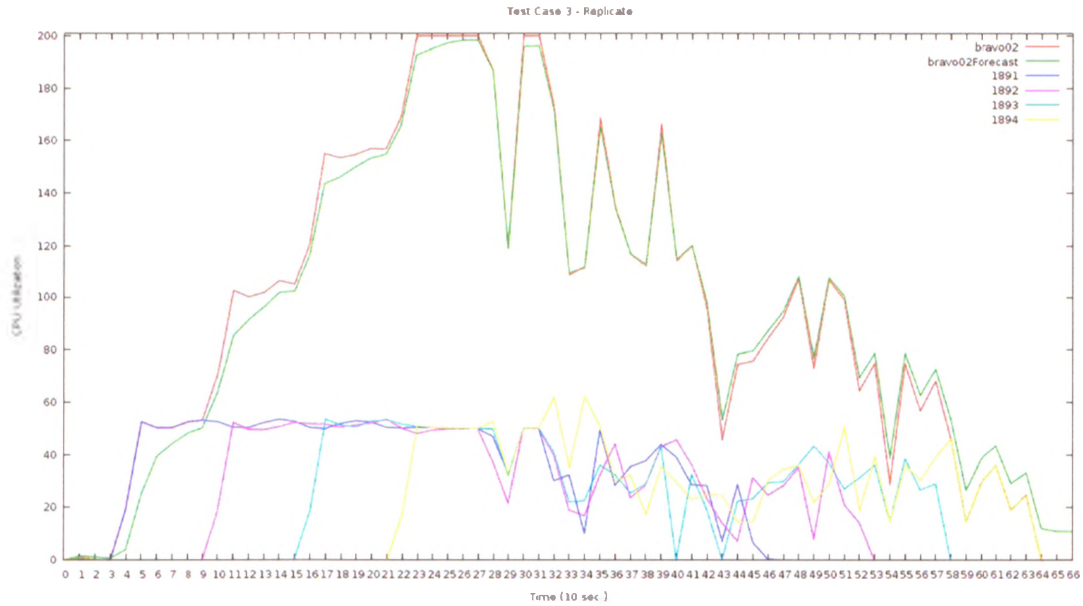

Figure 6.12: Experiment 3 - Replicate, bravo02

The web server **one.com** had connection times $min = 729.2$, $avg = 2916.9$ and $max = 9815.9$ milliseconds. The web server **two.com** had connection times $min = 702.4$, $avg = 2462.3$ and $max = 9737.5$ milliseconds. The web server **three.com** had connection times $min = 690.4$, $avg = 2537.6$ and $max = 9998.2$ milliseconds. The web server **four.com** had connection times $min = 688.5$, $avg = 2268.5$ and $max = 9375.6$ milliseconds.

Run 2: In the second run of the experiment, Golondrina was to search for possible replications if a resource stress situation was detected. Figure 6.12 shows the resource utilization of the containers **1891**, **1892**, **1893** and **1894**, and the resource utilization and predicted CPU utilization of the hardware node **bravo02**. Figure 6.13 shows the resource utilization of the replicas **91891**, **91892**, **91893** and **91894**, and the resource utilization and predicted CPU utilization of **bravo03**.

The first resource stress situation in **bravo02** was signaled at $t = 23$. Golondrina determined that the containers **1891**, **1892** and **1893** had to be replicated in **bravo03**. By $t = 27$ the containers **91891**, **91892** and **91893** had been created and the load balancer at the *gate of the cluster* was updated. The load balancer was

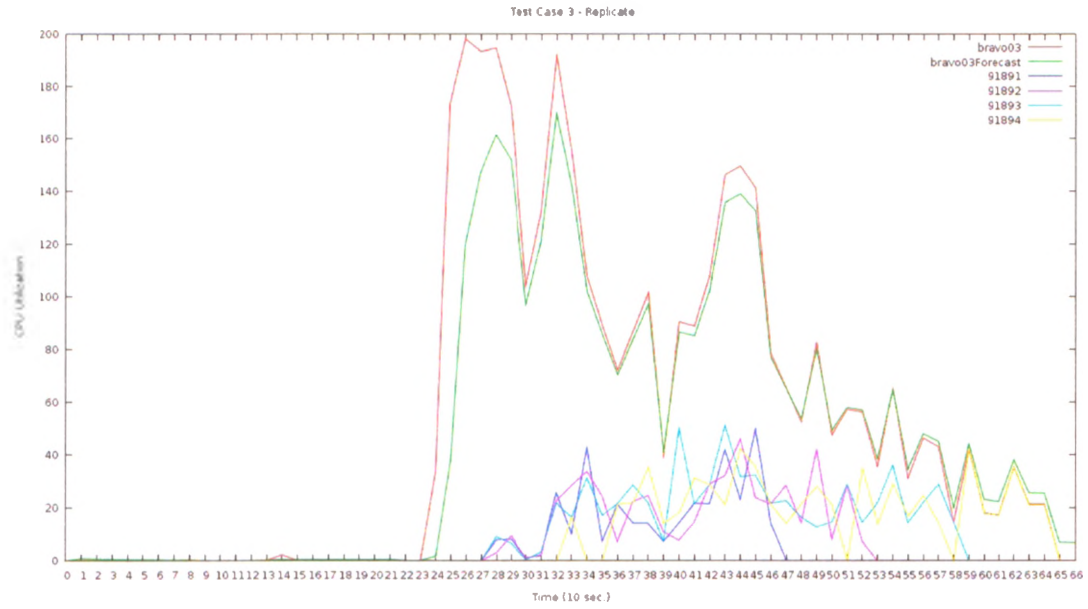

Figure 6.13: Experiment 3 - Replicate, bravo03

updated (and restarted) three times (one for each replication) in the period $(27, 28)$. During that period and the following one, it can be seen in Figure 6.12 that the CPU utilization of **1891**, **1892**, **1893** and **1894** decreased, due to connections that were refused or reset.

At $t = 28$, the containers **91891**, **91892** and **91893** had a low CPU utilization and remained with minimal CPU utilization until $t = 31$ (included). During those periods, the requests were handled by **1891**, **1892** and **1893**. As a consequence, an additional resource stress situation was signaled in **bravo02** at $t = 30$. Golondrina determined that **1894** had to be replicated in **bravo03**, indicating also that the action would not be enough to dissipate the resource stress situation in **bravo02**.

The web server **one.com**, hosted in **1891** and **91891**, had 7 failed requests (connrefused 4 connreset 3) and 2 lost requests (client-timo 2) out of 300, which resulted in an effectiveness of 97%. The web server **two.com**, hosted in **1892** and **91892**, had 6 failed requests (connrefused 5 connreset 1) and 5 lost requests (client-timo 5) out of 300, which resulted in an effectiveness of 96.33%. The web server **three.com**, hosted in **1893** and **91893**, had 10 failed requests (connrefused 5 connreset 5) and 1 lost

requests (client-timo 1) out of 300, which resulted in an effectiveness of 96.33%. The web server **four.com**, hosted in **1894** and **91894**, had 9 failed requests (connrefused 5 connreset 4) and 2 lost requests (client-timo 2) out of 300, which resulted in an effectiveness of 96.33%.

Compared with the first run of the experiment, where Golondrina took no action upon detection of a resource stress situation, the replications helped improved the effectiveness of web server **one.com** by 9%, of web server **two.com** by 4.33% and of web server **three.com** by 9.33%. The effectiveness of web server **four.com** suffered a degradation of 1.67%.

The web server **one.com** had connection times $min = 702.0$, $avg = 1151.3$ and $max = 6530.0$ milliseconds. The web server **two.com** had connection times $min = 694.0$, $avg = 1130.8$ and $max = 9882.7$ milliseconds. The web server **three.com** had connection times $min = 708.4$, $avg = 1281.0$ and $max = 9858.3$ milliseconds. The web server **four.com** had connection times $min = 689.7$, $avg = 1275.9$ and $max = 8468.4$ milliseconds.

In comparison with the web server's performance in the first run, **one.com** decreased the average connection time by about 60.53%, **two.com** decreased the average connection time by about 54.07%, **three.com** decreased the average connection time by about 49.51%, and **four.com** decreased the average connection time by about 43.75%.

Run 3: In the third run of the experiment, Golondrina was to look for migrations upon detection of a resource stress situation. Figure 6.14 shows the resource utilization of the containers **1891**, **1892**, **1893** and **1894**, and the resource utilization and predicted CPU utilization of the hardware node **bravo02**. Figure 6.15 shows the resource utilization of **1891** and **1892**, and the resource utilization and predicted CPU utilization of **bravo03**.

A resource stress situation was signaled at $t = 24$ in **bravo02**. Golondrina determined that the container **1891** was to be migrated to **bravo03**. The migration process was started, increasing the CPU utilization in **bravo03**. The CPU in **bravo02** was already exhausted, so the migration process competed for the CPU with the con-

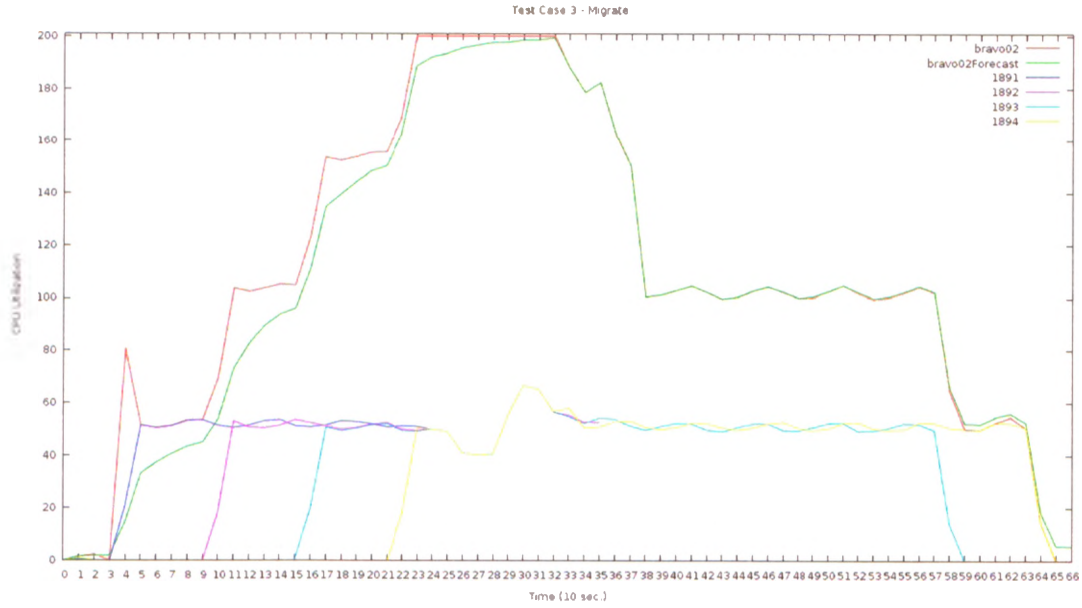

Figure 6.14: Experiment 3 - Migrate, bravo02

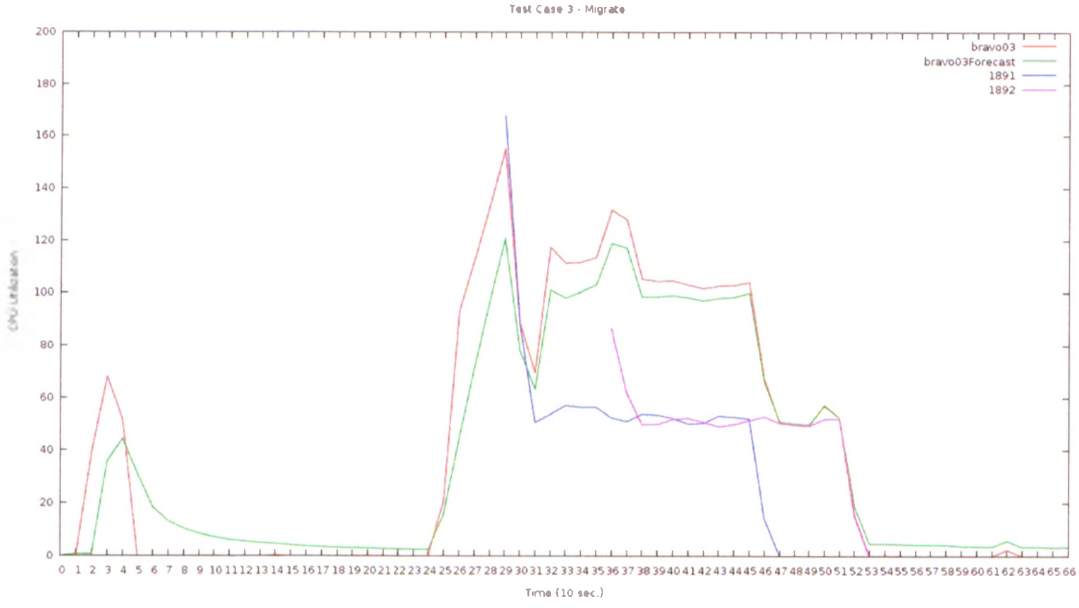

Figure 6.15: Experiment 3 - Migrate, bravo03

tainers. The four containers saw a reduction in their CPU allocation in the interval [26, 28] until the migration process ended in the period (28, 29).

At $t = 29$, **1891** had a peak of around 170% in CPU utilization. Taking advantage of the resource availability, the containers **1892**, **1893** and **1894** increased their CPU utilization during the interval [29, 33], what caused a resource stress situation to be signaled at $t = 30$. At that time, Golondrina decided to migrate **1892** to **bravo03**. The migration process ended in the period (35, 36) and **1892** reached a peak of around 85% in CPU utilization after being migrated.

A final resource stress situation was signaled at $t = 37$ with the CPU utilization of **bravo02** being 150.48% and becoming 100.5% at the following point in time. Golondrina found no solution to the situation.

The web servers **one.com** and **two.com**, hosted in **1891** and **1892** respectively, had 18 lost requests out of 300 (client-timo 18), which resulted in an effectiveness of 94%. The web server **three.com**, hosted in **1893**, had 14 lost requests out of 300 (client-timo 14), which resulted in an effectiveness of 95.33%. The web server **four.com**, hosted in **1894**, had 19 lost requests out of 300 (client-timo 19), which resulted in an effectiveness of 93.66%.

Compared with the first run of the experiment, where Golondrina took no action upon detection of a resource stress situation, migrations helped improved the effectiveness of web server **one.com** by 6%, of web server **two.com** by 2% and of web server **three.com** by 8.33%. The effectiveness of web server **four.com** suffered a degradation of 4.34%.

In comparison with the second run of the experiment, where Golondrina used the replication mechanism, migrations fell short in the effectiveness improvement by 3% in the case of the web server **one.com**, by 2.33% in the case of the web server **two.com**, by 1% in the case of the web server **three.com** and by 2.67% in the case of the web server **four.com**.

The web server **one.com** had connection times $min = 706.3$, $avg = 1051.0$ and $max = 9719.9.4$ milliseconds. The web server **two.com** had connection times $min = 691.3$, $avg = 1412.8$ and $max = 9966.3$ milliseconds. The web server **three.com**

Web Servers' Effectiveness			
Servers	Run 1	Run 2	Run 3
one.com	88%	97%	94%
two.com	92%	96.33%	94%
three.com	87%	96.33%	95.33%
four.com	98%	96.33%	93.66%

Table 6.5: Experiment 3 - Percentage of successful requests.

had connection times $min = 690.6$, $avg = 1343.0$ and $max = 9997.6$ milliseconds. The web server **four.com** had connection times $min = 698.0$, $avg = 1248.8$ and $max = 9589.1$ milliseconds.

In comparison with the web server's performance in the first run, **one.com** decreased the average connection time by about 63.96%, **two.com** decreased the average connection time by about 42.62%, **three.com** decreased the average connection time by about 47.07%, and **four.com** decreased the average connection time by about 44.95%.

The comparison with the web server's performance in the second run does not offer conclusive results.

Web Servers' Performance			
Servers	Run 1	Run 2	Run 3
one.com	2916.9	1151.3	1051.0
two.com	2462.3	1130.8	1412.8
three.com	2537.6	1281.0	1343.0
four.com	2268.5	1275.9	1248.8

Table 6.6: Experiment 3 - Web servers' average connection time in milliseconds.

As in the second experiment, container migration was an improvement over taking no action upon detection of a resource stress situation. Again, the benefits offered by the replication mechanism exceeded the benefits obtained through migration.

6.4 Summary

In this chapter, three experiments were studied. In all cases, two hardware nodes were managed by Golondrina. The number of hosted containers ranged from two to eight. The containers hosted an Apache web server and received HTTP requests for dynamic content. Each experiment was run three times, with Golondrina's configuration changed each time to react differently upon detection of a resource stress situation. The first time, Golondrina would take no action. The second time, Golondrina would search for replications. The third time, Golondrina would search for migrations. The results of the three runs of each experiment were compared with each other, based on the number of lost and failed requests and the performance of the web servers.

Chapter 7

Conclusion

This thesis presents a review of the research done in the area of resource management in virtualized environments. Golondrina, a resource management system, was designed and implemented to work with operating system-level virtualization. The system implements monitoring and resource stress detection mechanisms and uses mathematical models to predict resource utilization trends. The system relies on migration and replication mechanisms to deal with resource stress situations.

This work is one of the very few (if any at all) that proposes a resource management system for operating system-level virtualized environments. In addition, this is the first study that uses replication as an alternative to migration and compares both mechanisms. Others have proposed to do replication, but have not done it [33]. Others have implemented replication, but not migration [25].

Section 7.1 presents the conclusions that can be drawn from the experiments reported in Chapter 6. Section 7.2 discusses issues that were discovered during the implementation and evaluation of the system. Section 7.3 suggests directions for future work.

7.1 Conclusions

The experiments presented in Chapter 6 show that both relocation mechanisms offer an improvement over taking no action upon detection of a resource stress situation. Even if there are spare resources to allocate in the stressed hardware node, the mech-

anisms do not have a negative impact, which supports the use of the relocation mechanisms as preventive actions in case the resource utilization were to continue increasing in the stressed hardware node.

The replication mechanism offers a better improvement over the migration mechanism. One exception is the scenario where enough spare resources are available at the resource stressed hardware node for the migration process to use. In this case, the hosted containers see no performance degradation, so migration has the same benefits as replication. In other words, a policy could be defined to use replications whenever resources are exhausted and use migrations whenever there are spare resources.[1]

It is intentionally dismissed in this analysis a small performance hit that happens with replications due to the update of the load balancer. This problem does not belong to the replication process per se, but to the load balancing software used in the experiments. Pound does not allow for dynamic updates of its configuration, so a restart of the software is necessary. During the time that the restart process lasts, any request that arrives to the load balancer gets lost.

7.2 Discussion

As it was mentioned in the previous section, the selection of Pound as the load balancing software was not ideal. First, Pound does not allow for dynamic update of its configuration. This forces a restart of the software (and consequently an interruption in the service). Second, it uses a random algorithm for load balancing, selecting randomly the server to which forward each incoming request. This seems to contribute to an unstable balance of the load sent to the replicas.

It was stated in Section 4.1 that the system runs in a cluster of homogeneous physical servers. However, a certain degree of flexibility is possible. From the oper-

[1]This policy would be an oversimplification, since replication could still be used in scenarios where there are spare resources. However, replication would imply paying for two containers instead of one, which would suggest that the client would prefer doing migrations whenever possible. However, if the resource utilization of the container were to continue increasing, a replication could end up being the only solution if no hardware node had enough spare resources to host the migrated container. All this belongs to the problem domain of effective strategies for resource management, which can only be addressed once decision making support mechanisms are in place.

ating system perspective, OpenVZ is a modified Linux kernel that could work with different (GNU/Linux) distributions. CentOS was the distribution of choice for the hardware node's operating system, but Red Hat, Ubuntu or Debian could have been used. As for the containers, there is even more flexibility there and several templates from different distributions are available to use in the container creation process.

Golondrina was also designed to manage hardware nodes with different number of cores. As it is explained in Subsection 4.5.2, the Client component informs the Server component of the number of cores available in the hardware node. This information is used by the Server to determine the total computing power of the hardware node. However, for this feature to be fully functional, a mean has to be devised to express the load of a hardware node without losing perspective of its total computing power (50% spare computing power in a hardware node with one core is not the same as having 50% spare computing power in a hardware node with four cores). Also the CPU utilization threshold used by the resource stress detection mechanism would have to be tuned for each hardware node (a threshold of .75 for a single core is very different from the same threshold for four cores).

Section 4.2 states that the Server component runs in a non-virtualized physical server. However, it would be feasible for the Server to be hosted in a container. Precautions should be taken for the container not to run out of resources, since it would be hosting the software that manages the system and running out of resources would prevent the fulfillment of this responsibility. The container could be configured to have higher resource priority than the rest of the containers. This would mean that a dedicated server is not needed and would allow to take advantage of the migration mechanism for maintenance purposes. In any case, a careful study of the resource requirements of the Server component would be required.

Golondrina uses a black-box approach to resource monitoring. A grey-box approach would offer better benefits, but it requires additional software running inside the containers. There are two disadvantages: (i) Clients may not agree to have software that they neither need nor control installed in their container; (ii) Clients could potentially interfere with the operation of the software module and report unreal

statistics. This would increase the difficulty in managing the data center.

The resource utilization prediction model implemented in Golondrina was the Auto-regressive Model of Order 1 AR(1) (as explained in Subsection 4.5.3). The experiments showed that the prediction model ran behind the actual utilization level most of the time. Thus, it did not provide the expected benefit. Other prediction models could be implemented.

A profiling component with a historical policy was also used during the relocation process to predict container's future resource utilization. One argument against using a historical policy during the relocation process is that it could prevent a relocation from becoming effective at the moment (solving a current resource stress situation) based on events that took place in the past and might not be likely to happen again (and if they were to happen again, another relocation could be done later on).

A final issue to take into account with prediction models is what would happen with containers or hardware nodes that do not report statistics for a period of time or do it inconsistently. The absence of continuity in the reported statistics could handicap the prediction model effectiveness.

Regarding consolidation, Golondrina does not currently work towards that goal. However, some research groups have been working on the subject and have studied different policies and algorithms (see Section 3.2). Adding a consolidation mechanism with a historical policy to Golondrina would be interesting. Our current historical policy is rather primitive and limited. It considers the last hour of statistics and as a consequence any periodic increase in demand that happens in periods greater than an hour are ignored.

Golondrina's current relocation algorithm uses a greedy approach to the problem of finding a sequence of relocations. It is an effective approach, but it does not guarantee the best solution. The problem the relocation algorithms (as well as the consolidation algorithms) have to solve is complex, since it involves several dimensions. To begin with several resources have to be considered (CPU, memory, network). Moreover, multiple *bins* (the hardware nodes) are available to be filled. As it is noted in [20], the problem is similar to the NP-hard problem N-dimensional bin packing,

but with the additional restriction that the bins are loaded right from the beginning. It is also noted that each relocation entails CPU and network overhead, making the problem more complex.

Another issue is that the current relocation algorithm bases its decisions on the load of the containers, that is, the containers' resource demand. Another possibility to explore is considering the growing factor of the containers' demand. Instead of relocating the container with greater resource demand, relocate the container whose demand is growing.

Focusing on replication, the current process assumes that when a container is replicated, the load will be distributed equally between the two replicas. This would result in the predicted CPU utilization of the container being divided by 50% for each replica. Thus, allocating to each replica 60% of the resources indicated by the predicted CPU utilization would be enough for the replicas to handle their load. Not only is this assumption arbitrary, but it does not hold for the cases where one of the two replicas is replicated as well (that is, there are three replicas of the container). When the third replica is created, the load of the replicas does not decrease by a half, but by a third.

As mentioned before, migrations and replications impose a temporary overhead on the hardware nodes involved and the network. Both relocation processes could be studied in detail (as shown in [35]), so as to quantify the overhead and learn how to estimate it. This knowledge could be used to improve the resource stress detection mechanism (ignoring temporary overheads) and the relocation algorithm (selecting target hardware nodes that can handle the overhead). In addition, Golondrina currently triggers all relocations at the same time, resulting in a great overhead. It could be worthy to study triggering relocations in sequence.

7.3 Future Work

There are many directions in which Golondrina can be extended. One is to add memory as one of the managed resources. Work has already started in this direction.

Another extension is to add remote storage capabilities to OpenVZ. This would allow for a faster migration mechanism (and less overhead), since no filesystem would have to be copied between hardware nodes.

Regarding replications, the system should be extended to monitor the activity of replicas and determine when one or more of them are not necessary anymore. At that point, those unnecessary replicas should be stopped and removed.

Another interesting extension would be a resource *under-stress* detection mechanism. This would allow for the system to detect lightly loaded hardware nodes whose load could be moved somewhere else and the hardware node then suspended. This would result in improved server consolidation and reduced energy consumption.

The current CPU management model could be extended to integrate CPU reallocation [32]. The CPU reallocation mechanism could be useful in those cases where the whole managed system is resource stressed and no relocation is possible.

Another interesting possibility would be to redesign Golondrina, which is a centralized system. The Server component receives resource utilization statistics, analyzes data and makes relocation decisions. The Server component has the potential to become a bottleneck and represents a single point of failure. Golondrina could be redesigned with a distributed or hybrid approach in mind. The management could take place completely in the hardware nodes or be distributed between hardware nodes and a minimal central server.

Finally, other topics that could be studied include managing resources with the goal of minimizing energy consumption and the effect of relocations on containers with shared dependencies (that is, containers that host different tiers of an application).

7.4 Summary

This chapter presented the conclusions drawn from the system's evaluation. It also discussed issues discovered during the design, implementation and evaluation of the system, and possible improvements based on the research conducted by other groups and presented in Chapter 3. Finally, it suggested possible directions for future work.

Appendix A

Code Snippets

Listing A.1: gatherCtCpuStats method

```
def gatherCtCpuStats(self):
    newSet = {}
    stats = file(ctStats)

    # read the first line (version number) and the second one (parameter
        names)
    stats.readline()
    stats.readline()

    # now, for each line create a ContainerData instance
    while True:
        line = stats.readline()
        if len(line) == 0:
            break
        elems = line.split()

        # (parameter names) - second line - already read
        # VEID    user    nice    system    uptime    idle    strv    uptime
            used    maxlat    totlat    numsched
        ct = ContainerData(elems[0])
        ct.setUptime(int(elems[7]))
        ct.setUsedtime(int(elems[8]))
        newSet[ct.getCtid()] = ct
    stats.close()
```

```
    return newSet
```

Listing A.2: calculateCtCpuUsage method

```
def calculateCtCpuUsage(self):
    newCts = self.gatherCtCpuStats()
    for ctid, newCt in newCts.iteritems():
        if not ctid in self.containers:
            self.containers[ctid] = ContainerData(ctid)
        ct = self.containers[ctid]

        # CPU_USAGE_% = (NEW_USED - LAST_USED) / (NEW_UPTIME -
            LAST_UPTIME)
        cpuUsage = float(newCt.getUsedtime() - ct.getUsedtime()) / float
            (newCt.getUptime() - ct.getUptime())

        newCt.setCpuUsage(int(cpuUsage * 10000))
        newCt.setHostname(ct.getHostname())
        newCt.setIp(ct.getIp())
    self.containers.clear()
    for ctid, newCt in newCts.iteritems():
        self.containers[ctid] = newCt
```

Listing A.3: calculateHostCpuUsage method

```
def calculateHostCpuUsage(self):
    stat = file(hnStats)
    while True:
        line = stat.readline()
        if len(line) == 0:
            break
        elems = line.split()
        if len(elems) < 9:      # number of columns in entry _cpu_ since
            Linux 2.6.11
            continue
        if elems[0] == 'cpu':
            # columns in entry _cpu_ of /proc/stat - first line
```

```
            # cpu  user  nice  system  idle  iowait  irq  softirq
                  steal
            newVals = ( long(elems[1]), long(elems[2]), long(elems[3]),
                long(elems[4]), long(elems[5]), long(elems[6]), long(
                elems[7]), long(elems[8]) )
            newNonIdle = sum(newVals) - newVals[3]
            oldNonIdle = sum(self.oldVals) - self.oldVals[3]

            # CPU_USAGE% = (NEW_NON_IDLE - OLD_NON_IDLE) / (
                ALL_NEW_VALS - ALL_OLD_VALS)
            cpuUsage = int(10000 * (newNonIdle - oldNonIdle) / (sum(
                newVals) - sum(self.oldVals)))

            self.oldVals = newVals
            break
    stat.close()
    return cpuUsage
```

Listing A.4: getHostInfo method

```
def getHostInfo(self):
    info = {}
    info['hnid'] = os.uname()[1]
    info['cores'] = 0
    stat = file(hnStats)
    while True:
        line = stat.readline()
        if len(line) == 0:
            break
        elems = line.split()
        if elems[0][0:3] == 'cpu' and len(elems[0]) == 4:
            info['cores'] += 1
    stat.close()
    if info['cores'] == 0:
        info['cores'] = 1
    return info
```

Listing A.5: getCtsInfo method

```
def getCtsInfo(self, newCts):
    for ctid in newCts:
        filename = ctConfig % ctid
        config = file(filename)
        while True:
            line = config.readline()
            if len(line) == 0:
                break
            line = line.strip()
            if line[0:10] == 'IP_ADDRESS':
                self.containers[ctid].setIp(line[12:len(line) - 1])
            elif line[0:8] == 'HOSTNAME':
                self.containers[ctid].setHostname(line[10:len(line) -
                    1])
        config.close()
```

Listing A.6: migrateCt method

```
def migrateCt(self, ctid, destId):
    write('ACTUATOR: Migrating CT %s to HN %s .' % (ctid, destId))
    success = False
    cmd = ['vzmigrate', '--online', '-v', destId, ctid]
    try:
        retcode = subprocess.call(cmd)
    except OSError, e:
        msg = 'ACTUATOR: Execution failed: ' + str(e)
        write(msg)
        raise Exception, msg
    if retcode == 0:
        write('ACTUATOR: Successful migration of CT %d to HN %s .' % (
            ctid, destId))
        success = True
    elif retcode < 0:
        write('ACTUATOR: Child was terminated by signal %d .' % -retcode
            )
    else:
```

```
        write('ACTUATOR: Problem with cmd %s? Child returned %d .' % (
            cmd, retcode))
    return (success, ctid, destId)
```

Listing A.7: replicateCt method

```
def replicateCt(self, ctid, ip):
    write('ACTUATOR: Replicating CT %s .' % ctid)
    success = False
    try:
        if ctid[0] == '9':
            newCtid = ctid
            ctid = ctid[1:]
        else:
            newCtid = '9' + ctid
        path = self.requestEnvironment(ctid)
        self.unpackEnvironment(path)
        self.renameFile(private + ctid, private + newCtid)
        self.renameFile(private + newCtid + '/' + ctid + '.conf',
            ctConfig % newCtid)
        hostname = self.updateCtConfig(newCtid, ip)
        self.startCt(newCtid)
        success = True
    except Exception, e:
        write('ACTUATOR: Replication failed: ' + str(e))
        newCtid = None
        hostname = None
    return (success, newCtid, hostname, ip)
```

Listing A.8: (Container) init method

```
def __init__(self, ctid, hn):
    self['ctid'] = ctid
    self['hn'] = hn
    self['replicas'] = [self]
    self['migrating'] = False
    self['cpu'] = [0] * window
```

```
    self['timestamp'] = 0
    self['recCnt'] = 0
    self.CPUtheta = (1,)
    self.CPUmu = 0
```

Listing A.9: update method

```
def update(self,cpuUsage):
    self['cpu'].pop()
    if self['ctid'] == '0':
        self['cpu'].insert(0, cpuUsage * self['hn'].cores)
    else:
        self['cpu'].insert(0, cpuUsage)
    self['timestamp'] = time.time()
    self['recCnt'] += 1
```

Listing A.10: updateCpuModel method

```
def updateCpuModel(self):
    if self['recCnt'] < len(self['cpu']):        # _cpu_ has not been
        completed once yet
        tmpCpu = self['cpu'][0:self['recCnt']]  # duplicate useful part
            of array
    else:
        tmpCpu = self['cpu'][:]                  # duplicate complete
            array
    for i in range(len(tmpCpu)):
        tmpCpu[i] = tmpCpu[i] / 100.0

    mu = 0.0
    for i in range(len(tmpCpu)):
        mu += tmpCpu[i]
    mu = mu / len(tmpCpu)
    self.CPUmu = int(mu * 100)

    theta_num = 0.0
    theta_den = 0.0
```

```
    for i in range(0, len(tmpCpu) - 1):
        theta_num += (tmpCpu[i] - mu) * (tmpCpu[i+1] - mu)
        theta_den += pow(tmpCpu[i] - mu, 2)
    theta_den = max(1, theta_den)
    CPUtheta = theta_num / theta_den
    self.CPUtheta = (int(CPUtheta * 100),)
```

Listing A.11: cpu method

```
def cpu(self):
    cpu = self.CPUmu / 100.0
    for i in range(len(self.CPUtheta)):
        cpu += (self.CPUtheta[i] / 100.0) * ((self['cpu'][i] - self.
            CPUmu) / 100.0)
    return min(cpu, self['hn'].maxCpu / 100.0)
```

Listing A.12: profileCpu method

```
def profileCpu(self, percentile = 0.9):
    if self['recCnt'] < 10:
        return self.cpu()
    if self['recCnt'] < len(self['cpu']):        # _cpu_ has not been
        completed once yet
        tmpCpu = self['cpu'][0:self['recCnt']]  # duplicate useful part
            of array
    else:
        tmpCpu = self['cpu'][:]                  # duplicate complete
            array
    tmpCpu.sort()
    p = percentile * len(tmpCpu)
    return max(self.cpu() + 5, tmpCpu[int(p-1)] / 100.0)
```

Listing A.13: (HardwareNode) init method

```
def __init__(self, hnid, cores, pipe):
    self.hnid = hnid
    self.cores = cores
    self.pipe = pipe
```

```
    self.ct0 = None
    self.containers = {}
    self.maxCpu = 10000 * self.cores
    self.migCpu = 0
    self.overChecks = [0] * n
    self.overCnt = -1
    self.migrating = 0
```

Listing A.14: overloaded method

```
def overloaded(self):
    overloaded = False
    self.overCnt = (self.overCnt + 1) % n
    if self.ct0.cpu() / (self.maxCpu / 100.0) > OVERLOAD['cpu']:
        if sum(self.overChecks) >= k:
            overloaded = True
        self.overChecks[self.overCnt] = 1
    else:
        self.overChecks[self.overCnt] = 0
    return overloaded
```

Listing A.15: willFitRep method

```
def willFitRep(self, ct):
    share = 0.6
    cpu = (self.ct0.cpu() + self.migCpu / 100.0 + ct.profileCpu() *
       share) / (self.maxCpu / 100.0)
    if cpu > OVERLOAD['cpu']:
        return False
    else:
        return True
```

Listing A.16: recordMigration method

```
def recordMigration(self, ct, srcHN):
    self.migrating += 1
    self.migCpu += int(100 * ct.cpu())
    if srcHN:
```

```
        ct['migrating'] = True
    else:
        self.containers[ct.getCtid()] = ct
```

Listing A.17: recordReplication method

```
def recordReplication(self, ct, srcHN):
    self.migrating += 1
    share = 0.6
    if srcHN:
        # srcHN will lose 40% of the CT's load
        self.migCpu += int(100 * ct.cpu() * (1 - share))
    else:
        # destHN will receive 60% of the CT's load
        self.migCpu += int(100 * ct.cpu() * share)
        self.containers[ct.getCtid()] = ct
```

Listing A.18: migCompleted method

```
def migCompleted(self, ct, srcHN):
    self.migrating -= 1
    if self.migrating == 0:
        self.migCpu = 0
    if srcHN:
        self.overChecks = [1] * (k-1)
        self.overChecks.extend([0] * min(n-k+1, n))
        self.overCnt = k-2
        del self.containers[ct.getCtid()]
    else:
        ct['migrating'] = False
```

Listing A.19: repCompleted method

```
def repCompleted(self, ct, srcHN):
    self.migrating -= 1
    if self.migrating == 0:
        self.migCpu = 0
    if srcHN:
```

```
        self.overChecks = [1] * (k-1)
        self.overChecks.extend([0] * min(n-k+1, n))
        self.overCnt = k-2
```

Listing A.20: addNewObservation method

```
def addNewObservation(self, hnid, cores, ctid, cpuUsage, _time, pipe):
    if hnid in self.hns:
        hn = self.hns[hnid]
    else:
        hn = HardwareNode(hnid, cores, pipe)
        self.hns[hnid] = hn
        write('REGISTER: Added %s to list of HNs.' %(hnid))
    if ctid == '0':
        if hn.ct0 == None:
            hn.ct0 = Container(ctid, hn)
            write('REGISTER: Added distinguished CT ( CT0 ) to %s.' % (
                hnid))
        hn.ct0.update(cpuUsage)
    else:
        if ctid in hn.containers:
            ct = hn.containers[ctid]

            # check if CT was migrated and this is the first
            # observation gather at target HN
            if ct.migInProcess() and ct['hn'] != hn:
                ct['hn'].migCompleted(ct, True)
                hn.migCompleted(ct, False)
                ct['hn'] = hn

            # check if CT was replicated and this is the
            # first observation gather at target HN
            if ctid[0] == '9' and ct['hn'] != hn:   # replica of replica
                hn.removeCt(ctid)
                ct = ct.getCopy()
                ct['hn'].repCompleted(ct, True)
                hn.repCompleted(ct, False)
```

```
                ct['hn'] = hn
                ct['replicas'].append(ct)
                hn.addCt(ct)
        else:
            # check if CT was replicated and this is the
            # first observation gather at target HN
            if ctid[0] == '9':          # replica of original CT
                ct = hn.containers[ctid[1:len(ctid)]]
                hn.removeCt(ctid[1:len(ctid)])
                ct = ct.getCopy()
                ct['hn'].repCompleted(ct, True)
                hn.repCompleted(ct, False)
                ct['hn'] = hn
                ct['replicas'].append(ct)
                ct['ctid'] = ctid
            else:
                ct = Container(ctid, hn)
            hn.addCt(ct)
            write('REGISTER: Added CT %s to list of CTs on %s.' % (ctid,
                hnid))
        ct.update(cpuUsage)
```

Listing A.21: monitor method

```
def monitor(self):
    write('MONITOR: Starting...')
    self.register.processStats()
    overloaded = []
    underloaded = []
    for hn in self.register.getHns():
        for ct in hn.getCts():
            if time.time() - ct['timestamp'] > TIMEOUT and not ct.
               migInProcess():
                write('MONITOR: CT %s has reached its timeout.' % ct.
                   getCtid())
        if (time.time() - hn.ct0['timestamp'] > TIMEOUT):
```

```
            write('MONITOR: HN %s has reached its timeout.' % hn.getHnid
                ())
        else:
            if hn.migInProcess():
                write('MONITOR: HN %s is already involved in a
                    relocation.' % hn.getHnid())
            else:
                if hn.overloaded():
                    overloaded.append(hn)
                    write('MONITOR: HN %s is overloaded: %s' % (hn.
                        getHnid(), hn.resourceUsage()))
                else:
                    underloaded.append(hn)
                    write('MONITOR: HN %s is underloaded: %s' % (hn.
                        getHnid(), hn.resourceUsage()))
    if len(overloaded) > 0 and len(underloaded) > 0:
        write('MONITOR: Calling Relocator...')
        rels = self.relocator.relocations(overloaded, underloaded)
        if len(rels) == 0:
            write('MONITOR: No relocations could be found.')
        else:
            write('MONITOR: List of relocations:')
            for rel in rels:
                if rel[0] == 'mig':
                    write(str((rel[1][0].getHnid(), rel[1][1].getCtid(),
                        rel[1][2].getHnid())))
                else:
                    write(str((rel[1][0].getCtid(), rel[1][1].getHnid())
                        ))
            self.execRelocations(rels)
    write('MONITOR: Check finished.')
```

Listing A.22: migrations method

```
def migrations(self, overloaded, underloaded):
    write('RELOCATOR: Overloaded HNs: %d' % len(overloaded))
    write('RELOCATOR: Underloaded HNs: %d' % len(underloaded))
```

```
    migs = []
    overloaded.sort(key = lambda hn: hn.load())
    overloaded.reverse()
    underloaded.sort(key = lambda hn: hn.load())
    for hn in overloaded:
        containers = hn.getCts()
        self.decreasingLoadPolicy(containers)    # modifies list
            following policy
        for ct in containers:
            target = None
            for hnode in underloaded:
                if hnode.willFitMig(ct):
                    target = hnode
                    break
            if target:
                migs.append(('mig', (hn, ct, target)))
                hn.recordMigration(ct, True)
                target.recordMigration(ct, False)
                if not hn.stillOverloaded():
                    break
        if hn.stillOverloaded():
            write('RELOCATOR: Overload situation unsolved in HN %s .' %
                hn.getHnid())
        else:
            write('RELOCATOR: Overload situation solved in HN %s .' % hn
                .getHnid())
    return migs
```

Listing A.23: decreasingLoadPolicy method

```
def decreasingLoadPolicy(self, cts):
    threshold = 20              # threshold set to 20%
    i = len(cts) - 1
    while i >= 0:
        if cts[i].cpu() < threshold:
            del cts[i]
        i -= 1
```

```
    cts.sort(key = lambda ct: ct.load())
    cts.reverse()
```

Listing A.24: twoReplicasInHnPolicy method

```
def twoReplicasInHnPolicy(self, ct, hn):
    for rep in ct.getReplicas():
        if rep['hn'] == hn:
            return True
    return False
```

Listing A.25: replications method

```
def replications(self, overloaded, underloaded):
        write('RELOCATOR: Overloaded HNs: %d' % len(overloaded))
        write('RELOCATOR: Underloaded HNs: %d' % len(underloaded))
        overloaded.sort(key = lambda hn: hn.load())
        overloaded.reverse()
        underloaded.sort(key = lambda hn: hn.load())
        for hn in overloaded:
                containers = hn.getCts()
                self.decreasingLoadPolicy(containers)
                for ct in containers:
                        target = None
                        for hnode in underloaded:
                                if hnode.willFitRep(ct) and not self.
                                   twoReplicasInHnPolicy(ct, hnode):
                                        target = hnode
                                        break
                        if target:
                                reps.append(('rep', (ct, target)))
                                hn.recordReplication(ct, True)
                                target.recordReplication(ct, False)
                                if not hn.stillOverloaded():
                                        break
                if hn.stillOverloaded():
```

```
                    write('RELOCATOR: Overload situation unsolved in
                        HN %s .' % hn.getHnid())
                else:
                    write('RELOCATOR: Overload situation solved in
                        HN %s .' % hn.getHnid())
        return reps
```

Listing A.26: addToProxy method

```
def addToProxy(self, hostname, ip):
    write('ACTUATOR: Adding BackEnd %s for Service %s in load balancer\'
        s configuration file.' % (ip, hostname))
    hostnameLine = '\t\tHeadRequire "Host:.*'+ hostname +'.*"\n'
    newBackEnd = '\t\tBackEnd\n'+'\t\t\tAddress '+ ip +'\n'+'\t\t\tPort
        80\n'+'\t\tEnd\n'
    try:
        try:
            config = file(proxyConfig)
            lines = config.readlines()
            config.close()

            pos = lines.index(hostnameLine)
            lines[pos] = lines[pos] + newBackEnd

            config = file(proxyConfig, 'w')
            config.writelines(lines)
            config.close()
        except ValueError, e:
            write('ACTUATOR: Failure during load balancer\'s
                configuration update. Service %s not found.' % hostname)
        except Exception, e:
            write('ACTUATOR: Execution failed: ' + str(e))
        except:
            write('ACTUATOR: Execution failed: ' + sys.exc_info()[0])
        else:
            self.restartProxy()
    finally:
```

```
        config.close()
```

Bibliography

[1] VMware site - About Us. http://www.vmware.com/company/. Accessed August 2009.

[2] KVM site. http://www.linux-kvm.org/. Accessed August 2009.

[3] OpenVZ Project. http://openvz.org/. Accessed August 2009.

[4] CFQ - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/CFQ. Accessed August 2009.

[5] Linux: Fair Queuing Disk Schedulers - KernelTrap. http://kerneltrap.org/node/580. Accessed August 2009.

[6] Twisted. http://twistedmatrix.com/. Accessed August 2009.

[7] The Community ENTerprise Operating System. http://www.centos.org/. Accessed August 2009.

[8] Pound - Reverse Proxy and Load Balancer. http://www.apsis.ch/pound/. Accessed August 2009.

[9] The Apache HTTP Server Project. http://httpd.apache.org/. Accessed August 2009.

[10] Httperf site. http://www.hpl.hp.com/research/linux/httperf/. Accessed August 2009.

[11] Don Quijote by Miguel de Cervantes Saavedra - Project Gutenberg. http://www.gutenberg.org/etext/2000. Accessed August 2009.

[12] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[14] K. Begnum, M. Disney, A. Frisch, and I. Mevåg. Decision support for virtual machine re-provisioning in production environments. In *LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 1–10, Berkeley, CA, USA, 2007. USENIX Association.

[15] N. Bhatia and J. S. Vetter. Virtual Cluster Management with Xen. In *Euro-Par Workshops*, pages 185–194, 2007.

[16] G. Box, G. M. Jenkins, and G. Reinsel. *Time Series Analysis: Forecasting And Control, 3/E.* Prentice Hall, 1994.

[17] P. Emelianov, D. Lunev, and K. Korotaev. Resource Management: Beancounters. In *Proceedings of the Linux Symposium*, pages 285–292, June 2007.

[18] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and A. Kemper. An Integrated Approach to Resource Pool Management: Policies, Efficiency and Quality Metrics. Technical Report HPL-2008-89, HP Laboratories Palo Alto, Palo Alto, CA, USA, 2008.

[19] N. Holmes. The Turning of the Wheel. *Computer*, 38(7):100–99, 2005.

[20] C. Hyser, B. McKee, R. Gardner, and B. J. Watson. Autonomic Virtual Machine Placement in the Data Center. Technical Report HPL-2007-189, HP Laboratories Palo Alto, Palo Alto, CA, USA, 2007.

[21] A. Kochut and K. Beaty. On Strategies for Dynamic Resource Management in Virtualized Server Environments. In *MASCOTS '07: Proceedings of the 2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 193–200, Washington, DC, USA, 2007. IEEE Computer Society.

[22] K. Kolyshkin. Virtualization in Linux, September 2006. Documentation on OpenVZ.

[23] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo. VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 7, Washington, DC, USA, 2004. IEEE Computer Society.

[24] D. Marinescu and R. Kroeger. Towards a Framework for the Autonomic Management of Virtualization-Based Environments. 1. GI/ITG KuVS Fachgespraech Virtualisierung, Paderborn, February 2008.

[25] G. Munasinghe and P. Anderson. FlexiScale - Next Generation Data Centre Management. In *UKUUG Spring Conference*, 2008.

[26] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. *SIGOPS Oper. Syst. Rev.*, 41(3):289–302, 2007.

[27] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. G. Shin. Performance Evaluation of Virtualization Technologies for Server Consolidation. Technical Report HPL-2007-59, HP Laboratories Palo Alto, Palo Alto, CA, USA, 2007.

[28] K. K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. Live data center migration across WANs: a robust cooperative context aware approach. In *INM '07: Proceedings of the 2007 SIGCOMM workshop on Internet network management*, pages 262–267, New York, NY, USA, 2007. ACM.

[29] J. E. Smith and R. Nair. The Architecture of Virtual Machines. *Computer*, 38(5):32–38, 2005.

[30] A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[31] G. Vallee, T. Naughton, and S. L. Scott. System management software for virtual environments. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 153–160, New York, NY, USA, 2007. ACM.

[32] D. Weng, R. M. Bahati, and M. A. Bauer. Policy-Based Autonomic Management in Virtual Machine Systems. In *Proceedings of the 2009 International Conference on Grid Computing and Applications*, July 2009.

[33] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI)*, pages 229–242, 2007.

[34] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 2–12, New York, NY, USA, 2005. ACM.

[35] M. Zhao and R. J. Figueiredo. Experimental study of virtual machine migration in support of reservation of cluster resources. In *VTDC '07: Proceedings of the 3rd international workshop on Virtualization technology in distributed computing*, pages 1–8, New York, NY, USA, 2007. ACM.

[36] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova. 1000 Islands: Integrated Capacity and Workload Management for the Next Generation Data Center. *Autonomic Computing, International Conference on*, 0:172–181, 2008.