Managing Enterprise Services in Virtual Environment

Sebastian Schneider, Johannes Kirschnick, Nigel Edwards, Daniel Gmach
HP Laboratories
HPL-2008-85

Keyword(s):
Virtualization, autonomic management, closed loop management, resource control

Abstract:
Virtualization is becoming an increasingly commonplace commodity. The Service Lifecycle Management project at HP Labs is investigating how it can be applied to a more flexible provisioning of enterprise applications as a service by providing an automated delivery framework, enabling traditionally heavy-weight enterprise applications to become more agile.

This work presents a closed loop control framework that leverages the benefits of this new technology. It consists of abstract components responsible for measuring relevant data from the infrastructure as well as determining and effecting the most appropriate infrastructure changes to remedy possible adverse performance situations. Care has been taken, that each component is independent from the other.

To evaluate the framework for the SLiM project in a case study, a feedback controller and all other necessary components have been devised to autonomously manage the resource allocation to an SAP system in a virtual environment.

External Posting Date: July 6, 2008 [Fulltext] Approved for External Publication

Internal Posting Date: July 6, 2008 [Fulltext]

© Copyright 2008 Hewlett-Packard Development Company, L.P.

Contents

Chapter 1.

Introduction

The concept of virtualization has already been explored in the 1960s, to better utilize the expensive computing power of large mainframe computers. As mainframes were becoming more and more obsolete and computing power cheaper and decentralised, virtualization did not have such an important role to play any more. It was only in the mid-1990s, that companies like VMware reverted to the topic once more, as the x86 mainstream computer architecture provided more and more power that was increasingly running idle. In recent years, this development has gained further momentum, to the point where virtualization is supported natively by most modern processors. Still, the technology has by no means reached its peak. Further leveraging its benefits while exploring its implications on software operation and performance remains an exciting area of research, which this work delves into. It presents research of the applicability of virtualization for the delivery of enterprise applications by service providers.

1.1. Motivation

Traditionally, Enterprise Applications have been delivered as a software package that is deployed on hardware on the premises of the customer, either by the customers own IT department, the software company or an external agent. Once the application is operational, both the software as well as the hardware running it need regular upgrades, maintenance and troubleshooting. Even short downtimes of business critical applications can cost the enterprise hundreds of thousands of Dollars. So, to take advantage of sophisticated enterprise applications, companies have to face multiple financial burdens, most of which are not fully foreseeable when deciding on the application. In the end, they are confronted with upfront hardware and licensing costs and subsequent maintenance costs, that can be several times higher than the initial investment. It is estimated, that companies spend 75% of their IT budget on managing existing systems [7].

This uncertainty has deterred many small and medium sized businesses (SMBs) from using more sophisticated enterprise applications. Faced with increasing competition in their traditional heavy-weight-customer markets, developers of such enterprise applications are now trying to enter into the market of SMBs, since they make up 99% of all businesses in the EU [9] and represent a largely untapped resource for them. To offer these potential customers the flexible and straightforward operation and clear pricing for the software they really need and use, they are switching to a new business model, called *Software as a Service* or SaaS.

After a difficult start in the early 2000s the SaaS concept is now widely seen as one of the strongest motors of the software industry. Analysts of IDC predict an annual growth in the order of 30% until the year 2011 [21].

Hardware vendors like HP perceive this development as a threat, because the user is lost as a direct customer, since computing power is provided independently of the hardware by service providers. These service providers will most probably go for the solution offering the lowest total cost of ownership and be less inclined to buy from a big, reputable hardware vendor if it means paying a heavy premium. HP must consider the possibility to orient to higher levels of the supply chain. This means to support customers as a service provider or to offer services as a service to service providers, which means offering them the infrastructure they need to provide their service to their customers. Generally this comes down to computing power, storage and Internet access.

But since the service provider is oblivious to where the resources he is consuming come from, it is less efficient to provide them with dedicated hardware that would run idle or be overloaded, depending on the cycles in workload which the service provider experiences. Fortunately, virtualization enables ways to provide computing power much more flexibly, since workloads in a virtual machine can be scaled up or down on the fly, stacked on hosts and relocated between them as needed. This allows for the flexible division of a fixed pool of hardware between different customers, giving them just the proportion needed to fulfill the performance they have been contractually granted.

In this context we are talking about Service Level Agreements (SLAs), that define exactly what level of service or performance the customer is paying for and penalties for the provider for cases in which he does not deliver this performance. The same performance can now be sold at different prices, depending on the penalty and thus the importance the fulfillment of the contract has to the provider.

To fully benefit from the aforementioned advantages of virtualization technology, a sophisticated datacenter management is necessary. It needs to ensure complete performance isolation between the various customers, unknowingly sharing a single host computer. Security issues and data isolation also play an important role here.

At the same time datacenter management must optimally utilize the infrastructure, to make it possible to run a maximum number of customers on a minimal computing infrastructure, thus economising on the cost of hardware, maintenance and energy. Especially the latter is gaining significance, considering rising energy costs and increasing environmental awareness.

1.2. The Adaptive IT Project

Over the years, HP has invested in research and development of applications in the area of datacenter management. Aware of the potential of SaaS in association with virtualization technology and eager to further strengthen HP's position as a provider of hardware as well as software components for datacenters, the Adaptive IT (AIT) project of HP Labs is striving to create a management solution which can wield the potential of virtualization technology. A close cooperation with SAP Research offers the project the possibility to understand the motives and intentions of an application provider and to put their solutions to the test with one of the premier enterprise softwares, SAP R/3.

The project focuses on automating large parts of the process involved in managing the lifecycle of an enterprise application, aiming to design a system based on customer specifications. It proposes a model-based approach, to manage the complexity. The model is created out of a template and refined in a series of steps, each one creating an intermediate model and adding information. This information "flows" over into the subsequent refinement phases until the components identified in the outcome of the last phase can be deployed on hardware in a datacenter. These steps form the *Model Information Flow* (MIF), depicted in Figure 1.1.

Its name stands for the information that flows from the left hand side to the right hand side of the process. During this process, new information is added and joins the flow. In the first step from general to custom model, a general process is customised according to the user's business process. The second step takes into account specifications of the application, how it is packaged and what constraints it might imply for the later phases of deployment. Additionally, the application performance requirements are estimated, depending on the components that are to be used in the solution. Going from unbound to grounded model assigns abstract resource classes to the model, concerning the infrastructure needed as well as which and how many execution components of the application are required. When the model finally takes the step to become a bound model, specific instances of the infrastructure and application components identified earlier are assigned to it onto which it is deployed in the next phase, when the system is operational.

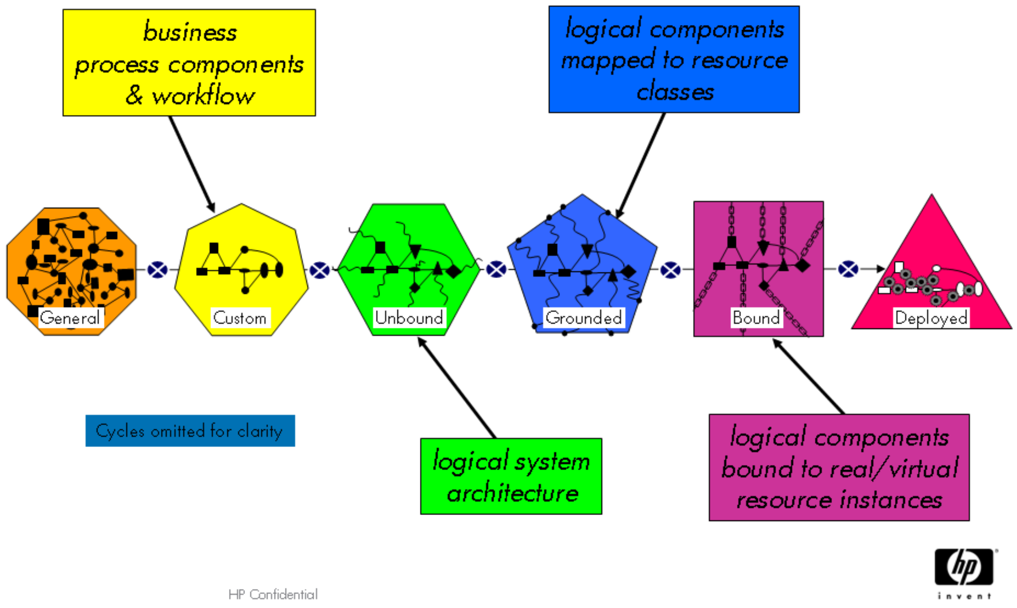

business process components & workflow
logical components mapped to resource classes
General
Custom
Unbound
Grounded
Bound
Deployed
Cycles omitted for clarity
logical system architecture
logical components bound to real/virtual resource instances
HP Confidential
invent

Figure 1.1.: The Model Information Flow - from [6]

The flow does not stop, once the model is deployed. A great advantage of the model-based approach is that, if changes need to be made at a later stage, parts of the MIF can be re-evaluated.

While not all parts of the MIF can be automated, especially the rightmost model iterations can be automated at this stage already. One piece still missing in the link was the automated monitoring of infrastructure and application performance and maintaining the infrastructure configuration accordingly. This is addressed by this work and the Closed Loop Management Framework is designated to work on the deployed model, when changing deployment details such as resource allocation, and the bound model, when relocating applications to another host.

A more detailed introduction to the MIF and the work of the AIT project at HP Labs can be found in [5].

1.3. The Solution Proposed in This Work

The solution proposed herein successfully tackles the problem of maintaining the performance of SAP R/3 running in a virtual machine, while at the same time allowing multiple virtual machines to be stacked on the same host. This is achieved by monitoring CPU usage in the virtual machine and adjusting the resources, in particular memory and CPU allocation, of the virtual machine accordingly. A framework was created to enable exchanging the mechanisms needed to adapt to the particular infrastructure. This comprises provisioning of information on virtual machine performance; effecting changes to resource allocation and the infrastructure; as well as finding optimal solutions to critical situations through a controller logic. So, even though the implementation of the framework is currently adapted to work in the environment of the AIT project, it can be adapted to other situations with minimal effort.

The framework is called the Closed Loop Management Framework, the corresponding controller the HP Labs Bristol or HPLB Local Controller.

1.4. Outline

The work described in this document was created as a Diploma Thesis of Technische Universität München in cooperation with HP Labs Bristol. It was developed on HP premises in Bristol, UK, as part of the Adaptive IT project.

The following chapters introduce some of the background on the subject of the work before detailing its contents and showing the possible improvements in practice.

Chapter 2 defines the problem space in which the solution tries to provide a contribution. It introduces basic concepts of Enterprise Applications, and virtualization technology, such as Xen in Section 2.1. Furthermore, Section 2.2 gives an insight into the concept of Software as a Service and the service provider model and suggests how this work might be applied in that area. Finally, it briefly introduces the theory of closed loop management in Section 2.3 and defines what the work's environment looks like in Section 2.4.

Chapter 3 describes the main work of this thesis, which is the Closed Loop Management Framework, and its components. First, it gives an overview over the structure, coherences and functioning of the framework in Section 3.1. Then it goes into more detail on the components of the framework in Sections 3.2 through 3.4, before showing how to use the framework and what has to be considered when deploying it in in Section 3.5.

Chapter 4 looks more closely on feedback controllers, in particular the HPLB Local Controller in Section 4.1, that was developed in the course of this work. Another feedback controller is introduced briefly in Section 4.2, the AutoGlobe Migration Controller, that will be used for the case study later on. Finally, a glance at the Virtual Machine Management API in Section 4.3 provides some insight into the concept of an API for this purpose, what methods and mechanisms it uses and how this is integrated into the framework.

Chapter 5 brings the concepts and constructs described earlier to life. We con-front the framework with situations that typically occur in an enterprise application setting, presented in Section 5.1. Section 5.2 details, how exactly the environment used for the tests looks like and how it is configured. We finally show how the framework and the controllers cope with the proposed scenarios in Section 5.3 and analyse the results in Sections 5.4, discussing the effectiveness of the solution and the limitations it might currently still have.

Chapter 6 concludes the work with a summary of its results in Section 6.1 and gives an outlook onto what we identify as future challenges and opportunities that might be encountered by the framework in Section 6.2.

Chapter 7 mentions some of the other approaches that exist to the subject of automated datacenter management.

Additional information can be found in the Appendix. It is particularly important to have a general idea of the concepts of hardware monitoring and virtualization, as is provided by Appendix A.

Chapter 2. Background

2.1. Enterprise Applications and Virtualization

As the term suggests, an enterprise application is providing a solution or service to an enterprise, which usually involves multiple employees simultaneously accessing the application, entering or querying data and performing calculations. However, not all the employees will be active all the time, but systems have to be provisioned with the resources to sustain a certain number of users. As long as the number of users is lower, acceptable responsiveness of the application is expected. The amount of users that a system can support is generally determined by the amount of memory installed and the power of the CPUs in the system. Often, it is also possible to run enterprise applications distributed among a number of servers. But the more powerful a server is and the more servers are provided, the more expensive is the initial investment and the long-term maintenance and energy cost. This is a Catch-22 situation: Providing enough computing resources to provide for the maximum number of users possible under any circumstances means overprovisioning the system for normal operations and thus wasting money. Limiting the resources, and at the same time the maximum number of users supported, can lead to shortage, should a peak demand occur. Then users are either not able to connect to the system or experience slow performance of the application or, in the worst case, lose data.

At the same time, the usage of enterprise applications is highly alternating. If an application is used interactively by employees, there might be peak usage on workdays during mid-morning and especially mid-afternoon hours, whereas there is virtually no usage at night or at lunch time, for instance. However, it is not always possible or advisable to shut down the application and the hardware it is running on during idle times, since it might interrupt user sessions and put data at risk.

The disadvantages of running enterprise applications on dedicated hardware are obvious. An apparent solution has emerged recently with the advent of virtualization technology. It allows applications to run inside containers, so called *virtual machines*. While they still need physical hardware to operate, they are independent from it in the sense that they can be moved to run on another physical host system in a matter of seconds to minutes, even without interrupting the operation of the application. They can also be provided with resources much more flexibly, which allows to distribute the resources of a host freely between the virtual machines running on it.

But the new technology is posing new challenges: How should the resource distribution among virtual machines be controlled? Will heavy-weight, traditional enterprise applications run stably in the new environment? How will they react to changes of previously unchangeable resources like memory and the number of CPUs? Will all user sessions still be available after the application's virtual machine has moved to another host in mid-operation?

This thesis will explore the aforementioned challenges, by providing a framework to monitor application performance and control resource distribution and virtual machine relocation. In a subsequent case study, the applicability of this approach is investigated, specifically including resource changing and relocation while the application is operational and under heavy load.

2.2. Software-As-A-Service and The Service Provider Model

The concept of "software as a service" has been circulating for a couple of years. In an article for ASPNews.com in March 2001, consulting analyst Phil Wainewright speaks about "Engines (...) that will drive software as a service" [27], one of the first occurrences of the term. He criticises the ignorance of this trend of large parts of the software industry but at the same time spots individual companies that have silently adopted this paradigm, something that even today is only slowly beginning to change. According to his analysis, application developers have to increase the flexibility of their application, building a core functionality base open to service providers that can use it to create a unique service for their customers. This allows independent software vendors developing for niche markets to use the provided foundation to build applications specialized for the needs of their market segment. At the same time, the application developer is able to levy at least parts of the revenue made in market segments that have previously been too small for it to focus on them.

Actually, the article omits a third and fourth party playing a part in the supply of software services: Those providing the computing infrastructure to run the new services and the hardware manufacturers, producing the actual fabric on which the whole service is based. But of course, a company is not limited to doing business on one of these layers only. In fact, it might be beneficial to cover multiple layers at the same time. On the one hand, the challenges and the complexity, and thus the amount of possible revenues, are moving toward the top layers. On the other hand, new business opportunities emerge as the intermediate layers are opened up. If a company can provide services on multiple layers, it can combine the possible revenues of all the layers. For example, a hardware manufacturer that is also active as a provider of computing services, will of course use its own hardware to deliver the computing services, while an independent computing service provider has the choice of competing hardware manufacturers. At the same time, the manufacturer can still sell hardware to other computing service providers.

Computing services consist of providing CPU power, memory, disk space and network bandwidth. The customer buying this service does not actually care, where and how it is provided, as long as it complies with certain contractual performance guarantees. This enables the provider to choose the most cost-effective way of providing the service. Since cost is driven mostly by maintenance and operational costs, the service provider will try to maximize the service output while minimizing the hardware necessary to provide it. One way of doing so is to use virtualization. As mentioned in the previous section and shown in Chapter 5, it makes better utilization of hardware possible by detaching virtual machines from actual physical hardware. For a computing service provider this means, that the computing infrastructure at his disposal can be shared between its customers much more flexibly. At times of low demand, customer workloads can be consolidated on fewer host computers while the rest of the infrastructure is shut down, saving energy and prolonging the absolute lifetime of the hardware.

Another benefit of adopting virtualization is, it allows to strongly isolate customers from each other, something that is also called *single tenancy*. Each customer has his own isolated instance of the enterprise application, a separate database and is also sealed off from other customers in terms of performance and security. In contrast, *multi tenancy* means having one scalable instance of an application that is shared by many or all customers, who are restricted to accessing their own data only by means of the application. This bears the risk that bugs in the application could allow malicious customers to gain access to third-party data. It is also much harder to ensure performance isolation, since all customers share the same computing base. Hence, if the application supports it at all, multi-tenancy can be easier to set up, but in the long run incorporates more risks. Single-tenancy is more versatile, since it can be applied to almost any application, and is structurally superior.

2.3. Closed Loop Management Theory

We use the term “closed loop management” to describe a key characteristic of our framework. The term is derived from *closed loop control* in control theory, an interdisciplinary field of applied mathematics. Control theory deals with influencing the behavior of dynamical systems and controlling their output. It is applied in many fields of engineering and has evolved to be used by psychological and social sciences as well.

The theory behind closed loop control is that the output of a dynamical system can usually not be influenced directly. The system provides certain “levers” of control, that can influence the output of the system, but it is also influenced by factors that can not be controlled and often are external. A good example of such a system is a cruise control system in a car. It is set to control the speed of the car, but what it can actually influence is the engine throttle, determining the fuel input to the engine, to put it simply. However, the same fuel input can result in different speeds, depending on external factors such as the inclination of the road or wind speeds, but also internal factors such as fuel quality, oil levels or tire pressure, if we count this as internal. Therefore, it is not sufficient to keep the engine throttle at a given output for a given speed. The speed has to be constantly monitored and the throttle adjusted accordingly. This is called *closed loop control.* The concept is illustrated in Figure 2.1.

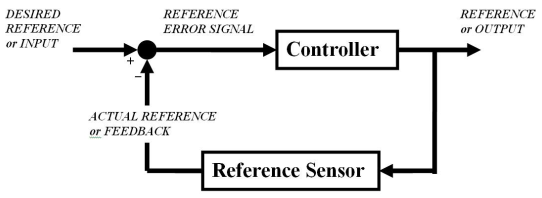


Figure 2.1.: Principle of closed loop feedback control

On the contrary, *open loop control* works with a static translation of desired output of the system to a corresponding input. It does not depend on any data to be monitored, but a deviation of the output, so called *translation error*, is not detected, not to mention corrected.

The mode of operation of our Closed Loop Management Framework is another classical example of closed loop control. It is able to monitor certain performance values on the systems it controls, but is not necessarily capable of directly controlling these values. It is provided with certain levers to influence the behavior of the system and thereby indirectly influence the output values. The settings of these levers have to be constantly adjusted, according to the monitoring values received.

What the performance values and control levers are, depends on the implementation of the framework, as it can be adapted to the needs of the environment at hand. Our implementation, which is described in the following chapters, provides CPU and Memory usage as performance values and gives the framework levers to control aspects of the virtual infrastructure, such as resource allocation and the placement of the virtual machines containing the application.

A more detailed introduction to the topic of control theory can be found in [18], but is not necessary for the understanding of the remainder of this document.

2.4. Environment of the Work

The environment in which the Closed Loop Management Framework is eventually run, plays an important role in its design. This section will briefly clarify, what we mean when we speak of an "environment" and how the environment we are running the Closed Loop Management Framework in looks like.

The framework has a very open design and is adaptable to a wide variety of environments. However, the term *environment* in this text stands for an environment consisting of a number of physical computers or *hosts*, that are connected to a network. Typically, they are located in a datacenter. These host computers are configured with an operating system supporting virtualization. The purpose of this setup is to run applications inside the virtual machines, that are accessed and used by remote clients.

The whole of the host computers together is also referred to as the *infrastructure* available for the use of the applications. Each host disposes of various *resources*, such as memory and CPUs, which can be allocated to the virtual machines running on this host. The usage of these resources by the application is monitored by *sensors* and reported to *controllers* who then can perform changes to the configuration and placement of virtual machines in the infrastructure through *actuators*. In this context, we use the term *metric* to refer to one of these performance indicators and any data that is monitored in the infrastructure. A metric can be something dynamic, such as CPU load as well as static such as the amount of memory installed in a physical machine.

Chapter 3.

Closed Loop Management Framework

The Closed Loop Management Framework was developed as part of the Adaptive IT project at HP Labs Bristol, which aims to provide a model-based structure for highly automated provisioning, deployment and operation of large scale applications. The envisioned contribution of the framework to the project is to automatically adapt the infrastructure once it is deployed, maintaining application performance according to the agreements with the customer, without the need for human intervention. It is configured and deployed as part of the Model Information Flow that was presented in Section 1.2. Hence it is designed to be highly flexible, robust and can be configured programmatically or through configuration files, without any part of the core program having to be changed.

The following section presents an overview of the Closed Loop Management Framework, before Sections 3.2 through 3.4 go into more detail on its individual components. The steps necessary to deploy and use the framework are discussed in Section 3.5.

3.1. Overview

A closed loop controller mechanism generally needs to be provided with information on the current state of the system it controls. It evaluates this information and finds an appropriate response or feedback and feeds it back into the system. (See Chapter 2 for some details on the concept of Closed Loop Management)

In our case, the system is a computing infrastructure, consisting of one or more physical host computers that run workloads; the situation is the load of the hosts and the current performance of the workloads; and the feedback are changes to the infrastructure, such as relocating workloads or varying the allocation of computing resources to the workloads. However, this is just an example of an application of

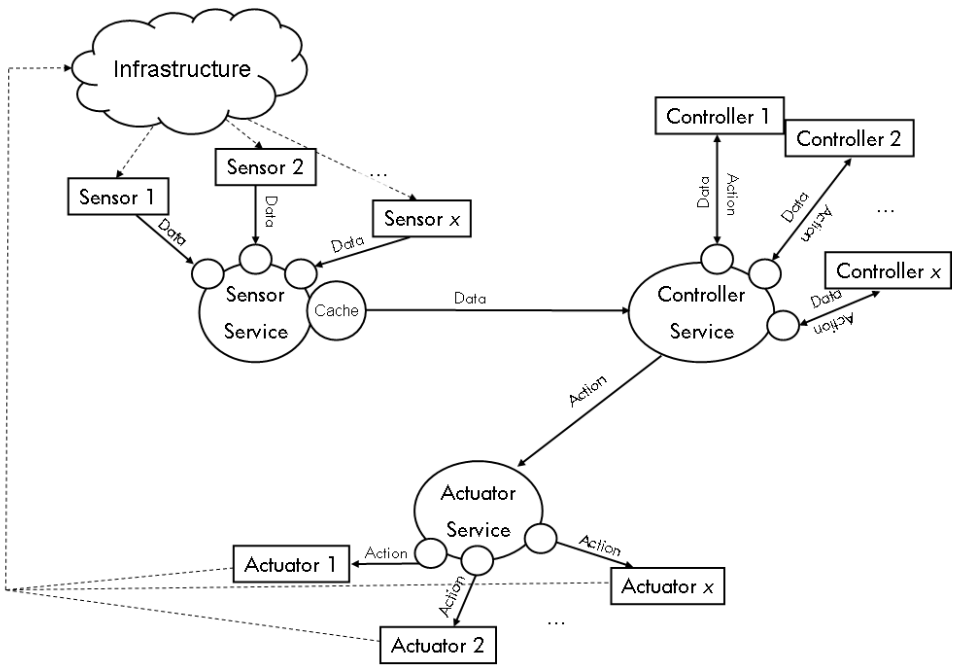


Figure 3.1.: Overview of the architecture of the framework

the framework, albeit probably the most common one.

Hence, to perform what is mentioned above, the three fundamental tasks that our Closed Loop Management Framework has to handle are:

1. Measuring data
2. Evaluating this data
3. Performing changes to the environment

Thus, the solution consists of three main components: 1. the SensorService, 2. the ControllerService and 3. the ActuatorService. This structure is illustrated in Figure 3.1.

From a high-level perspective, the SensorService is responsible for obtaining monitoring data from the hosts and virtual machines in the environment. It implements various different ways to access this data for the rest of the framework, in particular for the ControllerService. The ControllerService is the part of the framework that processes the monitoring data to detect, whether the system is in an acceptable state and if not, to trigger changes to the environment in order to improve the current state. To perform these changes, the ControllerService makes use of the ActuatorService, which is capable of carrying out changes to the infrastructure.

However, since flexibility is a key feature of the Closed Loop Management Framework, none of these components is designed to actually perform their respective task by themselves. Every component of the framework has to be easily interchangeable to simplify adaptation of the Framework to many different areas of application. This is especially important for the parts that are directly connected to the environment, since their functionality highly depends on the particular environment.

The task of measuring the monitoring data is carried out by *sensors*. These sensors have to be provided by the architect of the environment and be able to measure the relevant data and provide it to the SensorService in a specific way. The logic that is driving the management decisions is found in *controllers*, which are plugged into the framework via the ControllerService. It offers them a way to obtain monitoring data and to trigger actions in the environment. These actions are performed by the ActuatorService through *actuators*. They have to be able to understand and carry out any action that the feedback controllers used by the framework rely on to apply changes to the environment.

To give a general idea of the framework as a whole, each of its components is briefly introduced here, before going into more detail on how they were implemented in the respective sections later in this chapter.

The sensors, shown on the upper left hand side of Figure 3.1, can have any form, ranging from physical sensors, for instance temperature sensors mounted to a server rack, to any data that is simply polled, such as the current network bandwidth or CPU clock. However, the framework is rarely able to communicate with a sensor directly, given that the mechanisms and data formats of a particular sensor are usually unknown and different from those used in the framework. Here, the connectors, represented by the small circles adjacent to the big circle of the SensorService come into play. These connectors have to implement anything necessary to obtain the requested data from the sensor and translate it to an intermediate format provided and understood by the framework.

The SensorService as seen from the outside works as a black box that is, ideally, able to provide all data as requested by the rest of the framework. In it are hidden implementation details, such as how many sensors exist, how data is acquired and at what intervals. The performance impact of the monitoring activities is another concern. Frequent data requests served directly by the sensors will generally put a certain level of additional stress on the environment. In order to hide the sensor structure and to take advantage of the performance benefits, the SensorService uses a cache to buffer data it receives from the sensors and serves all requests from there.

Connected to the SensorService is the ControllerService, shown on the upper right hand side. Attached to it are the controllers that receive monitoring data from and transfer actions back to the ControllerService. Again, the communication of the controllers to the framework is provided by connectors to the ControllerService, which translate the framework's intermediate format for monitoring data to the way this data has to be provided to the respective controller. They also transpose the management decisions of the controller to another intermediate format provided by the framework to describe actions.

The ControllerService decides, whether the action will be forwarded to the ActuatorService or ignored. This is necessary in situations where multiple controllers simultaneously work on different architectural levels and it does not always make sense to carry out every action suggested, as they might conflict and have contrary effects. To keep the logic necessary for these decisions out of the framework's core codebase, we propose the use of a meta-controller, in cases where multiple controllers are used that can negatively influence each other. This meta-controller would act as another layer in between the controllers and the ControllerService, so that the ControllerService would be oblivious to the presence of multiple controllers, who would communicate to the framework through the meta-controller. It should contain rules that filter the actions initiated by the individual controllers to eliminate interferences.

The lower part of the figure shows the ActuatorService and corresponding actuators, who are also linked to the framework through connectors. Technically speaking, an actuator can have any form and use an arbitrary mechanism to perform its functionality. Possible mechanisms range from operating physical machinery or powering on additional parts of the infrastructure to merely sending an email or writing an entry to a log. It is essential, however, that, as a combined force, the actuators can perform the complete set set of actions used by the controllers to effect changes to the environment. Otherwise, the Closed Loop Management might not work correctly.

As we can see, there is a strong separation between the different components of the framework. There are basically three layers of abstraction:

1. The boundary layer, which is connected directly to the environment and consists of the sensors, controllers and actuators

2. The interface layer, connecting the implementation specific objects of the boundary layer to the layer below it. It consists of the connectors

3. The core layer, the main foundation of the framework, consisting of the SensorService, ControllerService and ActuatorService

Each of the components is interchangeable as long as it provides the functionality expected by its adjacent neighbours on the abstraction levels it is adjoined to them.

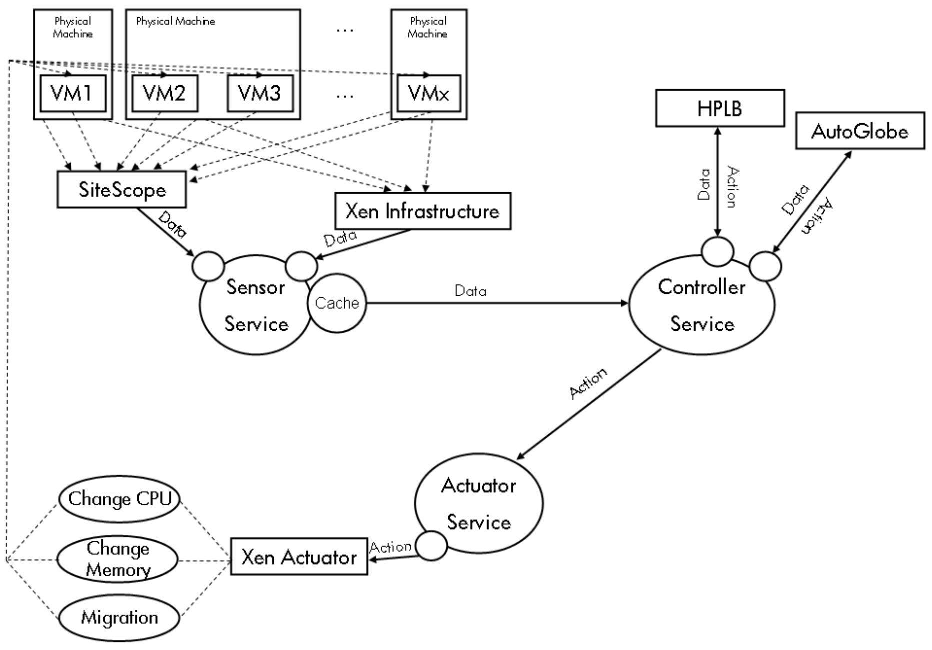


Figure 3.2.: Implementation of the framework for the Adaptive IT project

This decoupled structure facilitates the adaptation of the framework to different environments. An example of an implementation of the framework that has been realised as part of this work to suit the needs of the Adaptive IT project is shown in Figure 3.2, where all of the concepts from the abstract concepts presented so far have been put into place. A short description of this implementation follows, before we go into specific details of the individual components of the framework.

The environment the project is situated in consists of a pool of host computers and instances of SAP R/3 systems running inside virtual machines, which are distributed among the host computers. The monitoring is mainly carried out by HP SiteScope, a datacenter monitoring application, that can be configured to obtain a variety of data from systems using network connections. A more detailed introduction to SiteScope can be found in A.2. Additional information is gathered from the virtualization layer, Xen. These are the two sensors shown in Figure 3.2. Note, that SiteScope monitors both the virtual and physical machines, whereas the XenInfrastructure sensor reads its information only from the physical machines.

The goal of this scenario is to maintain the performance of the SAP R/3 systems while allowing multiple of the systems to be stacked on each host. But since the workload of each SAP system can change over time, depending on the number of users on the system, corrective control is necessary, as the systems resource demands change as well, both up and down. This is provided by the two controllers shown, the HP Labs Bristol Local controller and the AutoGlobe Migration Controller.

Thanks to virtualization, the infrastructure is highly adaptable. We take advantage of this fact by allowing the controllers to change resource allocation to the virtual machines and also relocate them to another host, which is called migration. All of this is realised by the XenActuator, which implements all the actions on Xen that the controllers use to maintain performance levels.

The following sections provide a detailed look at the implementation of the various parts of the framework, as well as how it is used. The effect of Closed Loop Management control and the effectiveness of the framework are examined in Chapter 5, using test runs on real infrastructure.

3.2. Sensor Service

The main task of the SensorService is to provide the feedback controllers with data from the managed environment on request, without making any assumptions as to what kind of data is to be transmitted nor how it can be collected. To make this kind of flexibility possible, the actual task of acquiring the data is carried out by *sensors*. The SensorService can not acquire this data on its own. Everything needed to connect to a particular data source has to be provided through a corresponding implementation of the sensor interface, which is shown in Appendix C.1. The interface ensures, that a custom sensor implementation can work with the rest of the framework. In the remainder of this document, the term *sensor* refers to the compound of sensor and interface implementation.

Due to the diversity of possible data sources, no general description exists as to how such an implementation might look like or work. It has to implement the mechanism to connect to the data source, that is the actual sensor, e.g. through a remote connection such as telnet or ssh, read the data, e.g. by parsing an output string or searching through log-files or reading status flags, and provide it in the data format used in the framework for monitoring data. This format is called *SensorData*.

SensorData acts as a container for monitoring data. It is initialised with the name of a single system and a list of metrics. For each of the metrics, a value can be stored as a String, which makes it applicable to many different data formats. However, integer or floating point values have to be converted first. Sensors will create a SensorData object for each of the systems they monitor, containing all the metrics available for that particular system. As we will see later, SensorData is also used by the controllers to request monitoring data from the SensorService.

Resource	Metrics	Method
	HP SiteScope Sensor	
CPU	Load, Idle, Wait, Steal	Linux system monitor `top`
Memory	Total, Free	Xen information tool `xm info` (for physical host)
		Linux resource monitor `cat /proc/meminfo` (for virtual machine)
Network	Bytes sent and received	`/bin/netstat -i -n`
	Xen Infrastructure Sensor	
CPU	weight assigned in the CPU Credit Scheduler (vm)	Xen configuration tool `xm sched-credit -d [vm name]` (run on host)
	number of virtual CPUs (vm)	Xen information tool `xm list --long` (run on host)
	CPU clock (vm and host)	`xm info` (run on host)
Memory	maximum memory allowance (vm)	`xm list --long` (run on host)

Table 3.1.: Sensors and Metrics used in our implementation of the framework

Usually, a single sensor can monitor a whole group of similar systems or even metrics. As an example, the sensors used in our implementation of the framework are listed in Table 3.1 along with the metrics they provide. One of the sensors is SiteScope, a datacenter monitoring solution, briefly detailed in Appendix A.2. It monitors most of the metrics relevant in our setup, on all parts of the environment. The second sensor we depend on is the Xen infrastructure itself, which provides information on the configuration of the virtual machine as well as the host. Here, the implementation of the sensor interface consists of different methods to obtain various data directly on each host, by connecting to the Xen hypervisor running there. The hypervisor is the core component of virtualization, controlling resources and mediating between physical hardware and virtual machines. For more information on the structure of Xen and how it is used in the project, please refer to Appendix A.1.

The association between a sensor and the metrics it monitors is made by the identifiers of the metric and is defined at the time of deployment of the framework. This is discussed in Section 3.5.3 later in this chapter. It is important to note at this point, that identifiers have to be unique throughout the framework and sensors and controllers have to refer to the same metrics by the same identifier.

The process of gathering the data from a sensor's source is a potential performance bottleneck, as it possibly involves using a network connection or other communication link. Often, operations necessary for measuring data can cause a significant amount of stress on a system, if performed excessively, thus creating additional resource consumption and influencing the data quality and overall performance negatively. In other scenarios, the monitoring data could hypothetically be available at certain points in time only or the occurrence could be driven by irregular external events. On the other hand, feedback controllers often operate with control loops in fixed intervals that do not necessarily coincide with the intervals at which monitoring data is measured. Additionally, certain controllers might need to be provided with data originating from specific periods or points in time.

All of this can be easily dealt with, if the SensorService features some kind of buffer or historical cache, storing temporal data per machine per metric. As the intervals at which each single value is generated should be regarded as completely arbitrary and independent from one another, it makes sense for the cache to hide this complexity and provide interpolated values for any requested point in time.

Consequently, the SensorService has been equipped with a cache based on time series, denoted SensorDataService. At its heart lies a multi-layer data structure that stores for each machine each metric's value with the time it has actually been measured. Thus, the SensorDataService has a time series of each metric on each machine at its disposal and is able to calculate an interpolated value for any point in time a feedback controller needs a value for. It provides four basic methods to request data:

1. `SensorData getData(SensorData sd)`
2. `SensorData getData(SensorData sd, Date newerThan)`
3. `SensorData getData(SensorData sd, Date start, Date end)`
4. `SensorData[] getData(SensorData sd, Date start,`
   `Date end, int measuringPoints)`

Method number 1 just provides the last data that has been measured, irrespective of when this happened. This can be limited in method number 2, which provides the last data, if it has been measured after the given date. Method 3 provides a single value for each metric requested, averaged out over the period defined by the two given dates, whereas method 4 returns an array of as many values as is specified in `measuringPoints`, equally spaced over the period between `start` and `end`. This way, a controller can request data for one or more specific points in time. The data is linearly interpolated between the measuring points stored by the SensorDataService.

These methods should comfortably accommodate most controllers or enable an adaptor to calculate the requested values otherwise.

So, the consequence of interposing the SensorDataService between the sensors and the controllers is that the sensors can push data to the cache at their own intervals or whenever it occurs and the controllers can request data whenever they need it, without the need of any synchronisation between the two. The controller does not need to worry about identifying the sensor that is collecting the requested data, since the cache accumulates all data irrespective of the source. This also means, however, that the controller can not tell, whether certain values will be available and if they are not, the controller does not know if this is a misconfiguration of the framework or the data is just not available, even though a sensor is actively trying to retrieve it. In the end, it comes down to a problem that is usually rooted out of the range of Closed Loop Management, so at present, no special treatment of the many different causes for unavailable data is considered useful. The SensorDataService simply returns an “Undefined” string, telling the controller that this data is not available.

The way the transmission of monitoring data works is illustrated in Figure 3.3. First of all, the SensorDataService signals to the sensors in which of the data they provide it is interested in by subscribing to this specific data at the corresponding

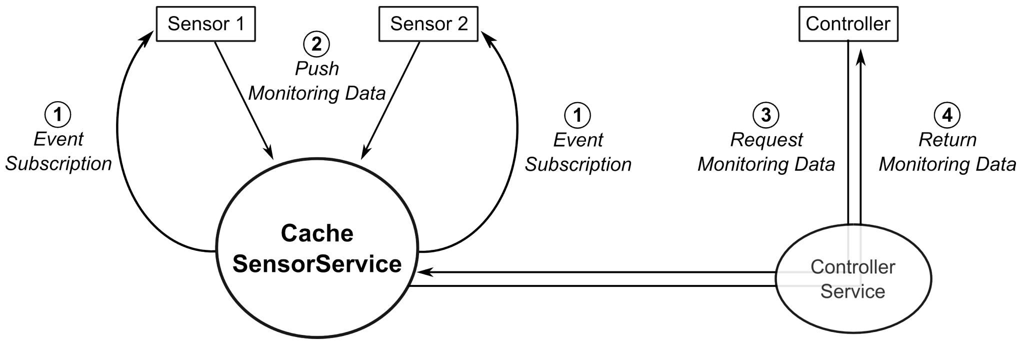


Figure 3.3.: Data exchange between a feedback controller and various sensors via a cache in the SensorService

sensor. Then, whenever the sensor has new data available, it “fires” an event by creating an instance of the *SensorEvent* class containing the new data. This is then passed to each subscriber. In our current implementation, the only subscriber is the SensorDataService, which then extracts the data and stores it in its internal datastructure. When a controller requires data, it sends a request via the ControllerService to the SensorService which hands it over to the SensorDataService. The returned data is passed on vice versa. Obviously, this highly decouples the controllers from the mechanism used to obtain the monitoring data. A further application could be to periodically store the monitoring data contained in the cache in an external database for further analysis of the cumulative data and to detect long-term trends, as an example.

3.3. Controller Service

The ControllerService is in charge of receiving data requests by the feedback controllers and passing them on to the SensorService. The same applies to the actions that the controllers might trigger after evaluating the data. These actions are defined in *ActionItem* objects and handed over to the ActuatorService, both described in the next section. If a feedback controller has not been specifically developed to work with the Closed Loop Management Framework, which will probably be the case most of the time, some form of interface or connector needs to be provided that translates between SensorData and the form of representation of monitoring data used in the controller, as well as the way the controller initiates actions and the ActionItem mechanism used in the framework.

If multiple independent controllers are used, they are likely to implement conflicting management strategies. It may nevertheless be necessary to rely on more than one controller, as they might take into account different aspects or hierarchical levels of the management domain. One controller could for instance consider the distribution of resources between workloads on a single machine only and work on a very fine grained level, whereas other controllers take into account multiple machines and try to distribute load evenly amongst them, without considering the fine grained local control of resources. While they seem to work independently, some form of synchronization is necessary, otherwise the actions of one controller might be nullified by actions of another.

Hence, if multiple controllers are to be used and interferences are likely, we propose the use of a meta- or proxy-controller. It is interposed in the communication between the controllers and the framework and has the ability to filter out actions triggered by the controllers according to rules that it implements. Technically speaking, it has to act as a proxy between the controllers and the ControllerService. This way a hierarchy can be imposed on the controllers as has been done in the implementation at hand, where both a controller exerting local control (the HPLB Local controller) and a controller distributing workloads among several hosts (the AutoGlobe Migration controller) are active at the same time.

Both controllers are constantly monitoring the situation of the hosts and virtual machines in the environment. Whenever the AutoGlobe Migration Controller initiates a migration, the actions of the HPLB Local Controller are ignored, until the migration is completed. This is to prevent interference with the migration. So, in our hierarchy of controllers, preference is given to the AutoGlobe Migration Controller. As we will see later, when the controller is presented in Section 4.2 of Chapter 4, it is configured to only becoming active when the total load on one of the hosts is becoming too high, limiting the applicability of local control anyway.

3.4. Actuator Service

To carry out actions indicated by the controllers, the framework relies on actuators which are managed by the ActuatorService. Its main function is to relay actions to the appropriate actuator and to make it possible to carry out actions asynchronously.

An actuator can implement any form of action mechanism, as long as it comes with a connector that is able to interpret the information in an *ActionItem* object and offers a method to perform actions as defined by the actuator interface, which is shown in Appendix C.2.

The ActionItem class enables the encapsulation of an action description in an object. It contains an identifier, the target system and parameters, if needed. Additionally it is possible to keep track of the progress of the action through an

Resource	Action	Method (run on host)
CPU	*CPU/Set/Number* (Set number of virtual CPUs)	`xm vcpu-set [vm name] [quantity]`
CPU	*CPU/Set/Weight* (Set CPU scheduler weight)	`xm sched-credit -d [vm name] -w [weight]`
Memory	*Memory/Set* (Set memory allocation)	`xm mem-set [vm name] [amount]`
Host	*Migrate* (Live migration to another host)	`xm migrate -l [vm name] [target]`

Table 3.2.: Actions provided by the Xen Actuator

integrated feedback mechanism. The actuator can indicate the state of the action in the object and store an arbitrary result (e.g. return values or error messages) that the issuer can then retrieve if it keeps a reference to the object. The actions are identified in a similar fashion as metrics and systems, see Section 3.5.3 for details.

Table 3.2 shows which actions the Xen Actuator, as part of our implementation, can carry out, their identifier and what Xen commands they result in. The commands have to be issued on the host of the virtual machine they are targeted at. In our implementation, this is done via SSH.

The actuations are performed asynchronously, that is, the ActionItems are queued up in the ActuatorService and performed one at a time in a separate program thread. Thus it is not possible to tell beforehand exactly when the actuation will take place. Additionally, to perform the actual task of the actuation can be time consuming. A migration, for example, can take up to a minute or more, depending on the memory footprint of the virtual machine and the network bandwidth available, whereas changing the memory allocation still can take a couple of seconds. This is the reason why it is necessary to carry out the actions asynchronously in the first place. It would not be very efficient to have parts of the framework waiting seconds to minutes for an action to take place.

Unfortunately, there is no direct way to verify that an actuation has successfully been carried out in Xen, as the actual command will always report success as long as it is valid. Whether the action is successful depends on the availability of resources or boundaries set at the time of setting up a virtual machine. So, if the controllers decisions depend on the success of previous actions, it has to collect this information through the sensor mechanism. Nevertheless, the Xen Actuator uses the feedback mechanism of the ActionItem class to inform the originator of the action when the command has finished. Equally, if there are any problems carrying out the command, the originator is informed and the error output of the command along with its exit code is provided. Thus, the controller triggering the action does know, when it has been issued to the Xen layer and can then verify it by consulting the data it receives through the sensors.

At present all actuations are performed directly on the Xen layer through one Xen Actuator, as shown in Table 3.2. They include changing memory allocation and the number of virtual CPUs, which roughly translates to the number of physical CPUs if there is no overbooking, i.e. more virtual CPUs have been assigned than actual physical CPUs exist. Additionally, the weight and capping mechanisms of the Xen credit scheduler can be configured, basically determining the share of CPU time a virtual machine is allocated. Finally, the Xen Actuator enables triggering live migrations, moving a virtual machine from one host to another without interrupting its work. Future work will include the integration of model based changes through the Request For Change mechanism used by the Adaptive IT project.

3.5. Using the Closed Loop Management Framework

Environments that can benefit from the use of the Closed Loop Management Framework typically consist of a number of servers or hosts. These are physical computers that run a virtualization technology such as Xen. Furthermore, there are a certain number of virtual machines inside of which applications are running. The number of virtual machines is not linked to the number of hosts at all, there can either be more or fewer, or the same number, of virtual machines than hosts. These virtual machines would be distributed amongst the hosts or maybe consolidated on one host in the beginning.

3.5.1. General Setup

One host should be set aside for running the framework as well as any auxiliary application, such as a monitor for the infrastructure. Of course, this can also be nested in virtual machines. Whatever method of monitoring is used, the framework needs to be supplied with a suitable adaptor. The same holds true for the actuations and the controllers that are intended to be used. Any adaptor has to be a piece of Java code that implements a translation between the data formats used in the framework, SensorData and ActionItem, and the data representation used in the external code, see Section 3.2 and 3.4 for details on this topic. Additionally, any sensor class needs to implement the sensor interface, to be able to interact with the cache in the SensorDataService and to provide methods to add and remove specific probes for specific systems. The actuator interface consists of one method only, defining the method called by the ActuatorService to perform an actuation described by an ActionItem object. These interfaces are shown in Appendix C.

To run the framework, a series of steps have to be followed:

1. instantiate the SensorService and the sensors and register the sensors with the SensorService — this will subscribe the SensorDataService as an event listener with every sensor and tell the SensorService, which sensor is responsible for which metrics

2. install the individual probes by passing to the SensorService a SensorData object with the corresponding metrics for every system to be monitored — this will let the individual sensors know which metrics they have to measure on which system

3. instantiate the ActuatorService and the actuators and register the actuators with their corresponding action

4. instantiate the ControllerService and pass it references to the SensorService and the ActuatorService

5. instantiate the controllers, providing them with a reference to the ControllerService and start them — it is sometimes necessary to wait a few seconds before starting the controllers to give the sensors time to fill the cache in the SensorDataService with some initial data

After these steps have been performed, the framework is ready and monitoring the environment. Once applications run inside the virtual machines, the controllers will be able to detect load levels and application performance, as specified by the probes deployed, and react accordingly, performing necessary changes to the environment.

3.5.2. AIT Implementation

The implementation of the framework for the Adaptive IT project uses HP SiteScope and the Xen virtualization technology. In addition to the steps above, the current version 8.5 of SiteScope requires that a SiteScope server has to be set up and be preconfigured for the monitoring of the systems in the environment. Future versions of the software will probably allow automating the configuration of SiteScope as well. Another sensor we use is a Xen infrastructure sensor, measuring virtual machine configuration and current resource allocation, which can be used out of the box.

As controller we employ the HPLB Local Controller, which has to be instantiated individually for every virtual machine that should be managed. If desired, the path and filename of a management profile XML-file can be provided to the constructor of the controller, which it will then import. Additionally, the AutoGlobe Migration Controller can be used, which requires our implementation of the Virtual Machine Management API to be set up first, providing a specific access layer to the monitoring and management environment. The AutoGlobe Migration Controller works on a pool of hosts, including empty ones, and relocates virtual machines, if necessary, to spread the load.

The information which hosts are available in the infrastructure and which virtual machines are running on them is maintained in a resource repository. It has to be set up with the initial situation and made aware of any changes, e.g. through a migration.

The actuator we use is the Xen Actuator, which can perform a number of changes to the environment by connecting to the host in question via SSH and calling the corresponding Xen commands. It also takes care of keeping the resource repository up to date when performing a migration.

3.5.3. Identification of Metrics, Actions and Systems

The boundaries between the various sensors or actuators are flexible. In our case some of the Xen infrastructure metrics were also gathered by SiteScope, as it can run any command specified by the user on a host and parse its output. The decision, which sensor monitors which metric is taken by the architect of the environment and is configured at the time of deployment of the framework by registering a sensor with one or more identifiers.

An identifier can either be identifying a single metric by its full name, or identify a whole group of metrics by a common prefix, that then also has to be used as a prefix for their individual full name. In any case, the full name has to be a unique identifier. When the SensorService is told to install a specific metric on a specific system, it can associate this to a particular sensor on the basis of the prefix to the full name of each metric. Furthermore, it is possible to connect prefixes to create a hierarchical tree structure, much like fully qualified domain names on the Internet.

This has been implemented in a general *Namespace* class, that defines how identifiers are structured for metrics and actions of the framework as well as the systems in the environment. It provides methods to build these identifiers out of Strings, extract the prefixes to an array and extract the name from the identifier. It uses the forward slash "/" as a separator between the individual prefixes and the name and sorts the prefixes hierarchically according to their position in the resulting full name. In addition, it defines constants for the names and prefixes that should be used for naming to prevent matching problems due to misspelling. To modify this list of constants or to change the mechanisms underlying the namespace creation, the Namespace class can be modified or replaced by a custom implementation, as long as it offers the same functionality.

The SensorService and ActuatorService use the methods offered by the Namespace class to split the hierarchical levels up again and find the sensor, respectively actuator, that is most specifically registered for the metric or action at hand. That is, it will go down a level at a time, first the full name including the metric's name, then removing the last part of the String and trying again to match the result to an association that has been previously registered. This procedure is organised in two nested loops, the inner loop working on the metric's identifier, the outer loop on the system's (see below). Thus, preference is given to the system level and if a sensor is specifically registered for that part of the environment's infrastructure, it will be picked before a sensor specifically registered for that metric.

For instance, SiteScope is first registered as sensor for metrics with the prefixes `CPU` and `Memory`. So, if the SensorService is then told to install a probe, that is, establish monitoring, of `CPU/Load` or `Memory/total` on a system X, it now knows, by looking at the prefixes (`CPU` and `Memory` respectively) of the two probe's metrics, that these two metrics are handled by the SiteScope Sensor. It hands down the installation request to the instance of the SiteScope Sensor, that was given by the installation routine in the first step, at the time of registering.

If another sensor had been registered for `CPU/Load` specifically, it would have been chosen over the SiteScope Sensor. However, if a third sensor had been registered for `CPU` Metrics on system X, it would have been chosen over both the other two sensors, as preference is given to system-specific sensors.

In short, implementing the naming scheme as explained above makes it possible to define general sensors which work for the vast majority of metrics on most machines. At the same time, it allows specific sensors to be provided for a certain machine or machines for some or all of the metrics monitored on them.

The same generally applies to the actuators, especially when using multiple actuators. We were only dealing with one actuator in our implementation, but the ActuatorService implements the same strategy for actions to be identified, structured and assigned to an actuator by their identifier.

Another issue is the identification of systems in the environment. While their hostname or IP-address on the network usually is a unique identifier, it may not be easy to distinguish between virtual and physical machines based on that information. As it can be vital for certain sensors or actuators to know, whether they are dealing with a virtual or a physical machine, we use the prefix mechanism to easily distinguish between the two. Identifiers of Virtual machines consist of a `VM Level` prefix, followed by the hostname of the virtual machine. The same applies to physical hosts, which have a `Host Level` prefix. This information can also be used to register sensors and actuators for a part of the environment's infrastructure only, identified by a certain prefix, enabling a very fine grained association between systems and actuators or sensors. However, the sensors and actuators have to be aware, that this identifier does not work on the network level and use the mechanism to extract the network hostname provided by the Namespace class.

For more information on the use of the Closed Loop Management Framework, please refer to the case study in Chapter 5. Further information on the controllers mentioned above and the Virtual Machine Management API can be found in Chapter 4.

Chapter 4.

Feedback Controllers

At the heart of any Closed Loop Management implementation lies one or more feedback controllers. They form the logic that determines appropriate actions for any given situation encountered by the environment and they are the main basis for the effectiveness of the implementation. Given their importance and complexity, it is essential that the Closed Loop Management Framework can harness the power of as many different controllers as possible, to enable employing the controller or controllers that work best in the present environment. Chapter 3 described, how this is achieved through the design of the Closed Loop Management Framework.

The following sections of this chapter provide an insight into two instances of feedback controllers that have been integrated during the development of the Closed Loop Management Framework for the Adaptive IT project. They are on the one hand the HPLB Local Controller, that has been developed during this work from the experience with SAP R/3 in virtual environments. It is designed to locally preserve the performance of a centralized SAP instance in a single virtual machine on its current host. On the other hand, the AutoGlobe Migration Controller monitors load levels on a pool of hosts and relieves overload situations on single hosts by migrating, e.g. moving, virtual machines to hosts having more free capacity available, and so effectively balancing load.

Increasing interest in datacenter management through virtualization has led to the development of various other feedback controllers. At HP Labs, the Computational Fluidity project and the ACTS team are also working on this topic. Their integration was beyond the scope of this work at the present stage, more detailed information can be found in [16] and [20] respectively. Nevertheless, the integration of these controllers will be the target of future work of the Adaptive IT project.

4.1. HPLB Local Controller

While it would have been possible to use an existing controller to drive the Closed Loop Management Framework for our particular setup, it was particularly interesting for us to work out our own attempt at a feedback controller, implementing the ideas and experiences of SAP testing in virtual environments. This has not only increased our insight into this area of datacenter management, but might eventually provide an alternative to existing developments.

Generally, the provisioning of the data that the controller works with can be problematic. Not all the metrics are always easily available. By building our own controller we are trying to minimize this problem, as we designed it to rely only on system-level metrics that are available independently of the particular application that is monitored. Furthermore, measuring the data at system-level is easier and generally has less impact on the performance of the managed system when compared to application internal monitoring.

The architecture and implementation of the controller presented herein is likely to change in future versions. At the moment it should be regarded as a first attempt at casting into a controller mechanism the empiric testing experience of the AIT project. As the results of our case study in Chapter 5 suggest, it is already capable of handling shifting load patterns that typically occur in enterprise applications. Possible future enhancements will be discussed towards the end of this section, after presenting in the following paragraphs what the current state of the HPLB Local Controller looks like.

4.1.1. Architecture

The controller was developed as a loop based condition/action system. The way this is implemented is shown in Figure 4.1 on page 34. It works with a predefined set of metrics, for each of which default upper and lower thresholds are given that can be customised by the user. The metrics are checked in a fixed order, representing their impact on the system performance and the anticipated effect of the corresponding remedy. Each metric is associated with a certain action that is most likely to affect it positively.

As the diagram shows, the controller will check so called *Defer Counters* before triggering an action. Two separate counters exist globally for increasing and decreasing of allocations. Their initial values are defined in the management profile and prevent the controller from changing resource allocations too quickly, giving the application time to adjust to the changes. In our example, the controller will instantly increase any resources but will wait four loop cycles of 15 seconds before decreasing. Whenever an action has been taken or none of the metrics is neither too high nor too low, both counters are reset to their initial values.

More on the metrics, their thresholds and the actions they are associated with will follow in the next section on the implementation of the controller.

4.1.2. Implementation

The metrics considered by the controller are the four different states of a CPU in virtual environments, as measured by the Linux CPU monitor *top* [1], run inside the virtual machine. This only provides meaningful results with version 3.2.6 and above, since top has to be aware that it is running in a virtual environment. See [25] for a discussion of this phenomenon. The program distinguishes between CPU Idle, Load, Wait and Steal and shows the proportion (in percent) of the time the CPU spends in either one of them. CPU Idle and Load are defined the same way as for a physical CPU. CPU Wait occurs when the virtual machine has to wait for an I/O request to finish. CPU Steal on the other hand occurs when the CPU is in use by another virtual machine.

Each metric has a maximum and a minimum threshold. The maximum threshold should not be exceeded, whereas continuously undercutting the minimum threshold indicates potential for removing resources. The opposite is the case for CPU idle time, naturally. Falling below its minimum value indicates a critical situation. If on the other hand enough idle CPU time is available, it still can make sense to treat exceptionally high proportions of the other states, as indicated by the maximum thresholds, to prevent future problems, when the CPU demands of the application could rise all of a sudden. In the case of an underload situation, where one or more of the metrics fall below their minimum threshold, the controller will free up the associated resource for other virtual machines.

The default threshold values shown in Table 4.1 on page 35 are integrated into the Controller. They can be customized if needed, by providing a management profile in an XML file. An example can be found in Appendix B.1. The management profile also contains the increase and decrease factors for memory and CPU allocation and defines the responsiveness and aggressiveness of the controller. Aggressiveness indicates, whether the controller should increase allocation of CPU time at the expense of other virtual machines. The responsiveness is given as a delay, measured in control loop iterations, for how long the controller should wait when thresholds are exceeded to perform the corresponding action. This can be defined independently for increasing and decreasing resources, as it makes much more sense to react fast when increasing, whereas the situation should be observed a little while before taking away resources, to achieve a stable state and avoid errat-

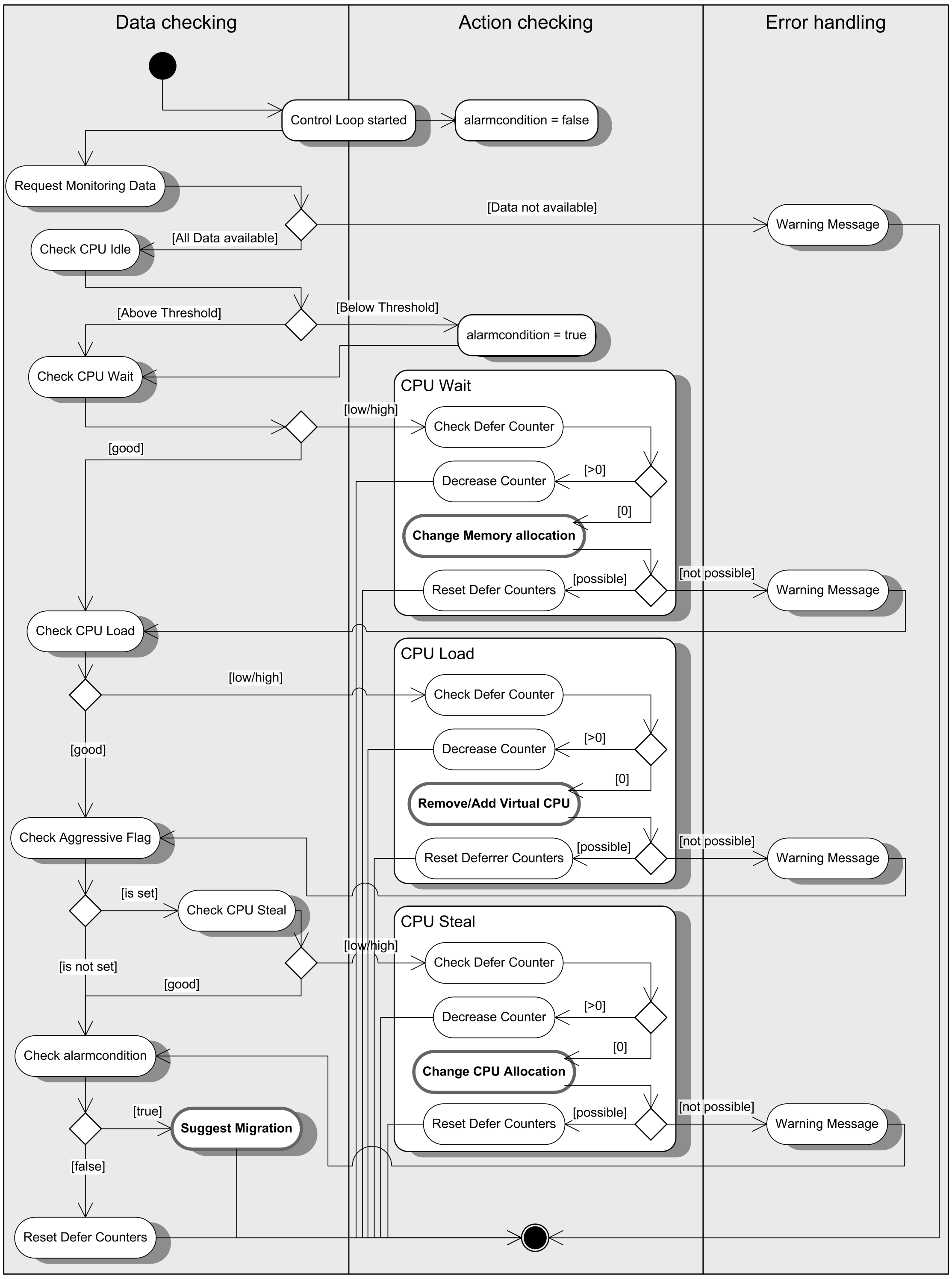


Figure 4.1.: Activity Diagram showing the decision process of the HPLB Local Controller

Metric	Default Threshold Max / Min	Problem Assumption	Controller Action
CPU Idle	80% / 35%	Alarming situation in general	Minimize CPU wait, load or steal time (see below)
CPU Wait	40% / 10%	Heavy swapping activity due to memory lack	Increase memory allocation if possible
CPU Load	65% / 25%	Not enough CPU power available	Allocate another CPU
CPU Steal	40% / 15%	Other virtual machines are active on the same host	Increase CPU time allocation
No more resources available to the vm		The host is overloaded or not powerful enough	Suggest migration to another host

Table 4.1.: Situations and Actions of the HPLB Local Controller

ically changing resource allocation. The associated action is only performed, when the threshold is exceeded for as many iterations as is defined in the management profile. The interval of these iterations can be defined at the time the controller is instantiated, along with the location of the management profile, if desired. By default the controller will perform an update iteration every 15 seconds.

The diagram in Figure 4.1 illustrates the control loop of the HPLB Local Controller. It first requests monitoring data for the four metrics, CPU Idle, Load, Wait and Steal. Then it checks one by one the associated thresholds from the management profile to trigger the most appropriate action, printed in bold. The order in which they are checked is relevant and reflects observations made previously on the impact of the different CPU states.

Our experience showed that performance degradation can generally be narrowed down to a low share of idle CPU time. Hence, this is checked first and if it falls below a critical threshold, 35% in our example, application performance and stability may be impacted. In that case, the controller sets an *alarmcondition* state variable to be *true*. When alarmcondition is set to true, the minimum thresholds from Table 4.1 are taken as maximum thresholds for the rest of the metrics, as the controller is trying to free up additional CPU time for the application at all costs.

The reasons for low CPU Idle time vary and so do the possible remedies: A lack of memory can force the system to swap out data heavily. This is indicated by a high proportion of CPU Wait times, which the controller consequently tackles by increasing the memory allocation of the virtual machine. Another reason for low CPU Idle time can simply be that the amount of CPU power available to the virtual machine is insufficient, which is indicated by high levels of CPU Load times. To improve this, the controller can add another virtual CPU to the virtual machine. However, it does not make sense to assign more virtual CPUs than the host has physical CPUs. This has to be considered when creating the management profile. Furthermore, the host can be overloaded by other virtual machines or the Domain-0, which is indicated by a high proportion of CPU Steal time. The only way to counteract this situation, is to increase the proportion of CPU time the scheduler assigns to the virtual machine. The problem is, that this adversely affects other virtual machines or the Domain-0 on the same host. Thus, this is not done by default, but can be enabled by setting an *Aggressive* Flag in the Management Profile to 1. In case of an underload situation, i.e. continuously undercutting the respective minimum threshold of one of the metrics, the controller triggers a deallocation of the associated resource.

It has to be mentioned that we can not simply consult the amount of free memory in the virtual machine to know when additional memory is needed. SAP gradually occupies all available memory and does not release it again, not even after users log off.

In its present form, the HPLB Local Controller provides local control for a single virtual machine. So, every virtual machine has to be managed by its own controller, which independently tries to optimise the performance of "its" machine. This inevitably leads to competition if two or more virtual machines have high resource demands and at a certain point, one or more of the controllers may find themselves unable to perform additional changes. They then signal this to the ControllerService, who has a separate controller at his disposal that is able to decide if and where to migrate a virtual machine. Such a controller could be the AutoGlobe Migration Controller, that is used in our case study in Chapter 5 and is presented briefly in Section 4.2.

4.1.3. Application

To use the controller with the framework, it is sufficient to provide it with a reference to the ControllerService and the name of the virtual machine to control via its constructor. It will then assume default values for the management profile and a default interval of 15 seconds between the control loop iterations. Alternatively, a different interval and/or the full path to a custom management profile can be provided.

In each control loop, the controller queries the data it needs for the virtual machine via the ControllerService and passes on actions when needed. Strictly speaking, it does not need a particular connector to the framework. Since it was developed at the same time, it was designed to work with the data formats of the framework, namely SensorData and ActionItem.

4.1.4. Future Enhancements

Due to time constraints, not all of the ideas for this controller could be implemented in the present version. Here, we present some of them.

Future enhancements can be aimed at making the controller more flexible. The static consideration of the four CPU metrics could be replaced by defining the relevant metrics in the management profile. For each metric the profile would contain which action it is associated with and what thresholds it has to maintain.

Additionally, the order in which the metrics are checked could be made more adaptive. While the current order is highly effective in the situations we consider to use the controller in at present, it might make more sense to implement a mechanism for choosing the metric, and with it action, most relevant at each given moment. Most relevant might be the metric, whose value is furthest from the threshold, for example.

The intelligence of the controller could also provide potential for improvement. It could incorporate a state-machine model, where it is aware of its last action and how long ago it has been taken. So, if a further increase is necessary immediately after an increase, the next increase could be greater. This would help react faster to sharp rises or drops, where the normal rate of resource change might be insufficient. The controller could even calculate a gradient for the increase of a value and scale the resource adjustment accordingly, anticipating how load levels could evolve from the current point on.

Furthermore, better allocation decisions could presumably be made, if one controller would maintain all virtual machines. It would then be aware of how many virtual machines are active on one host, how the total load of the host is and how to better distribute the resources among the virtual machines.

4.2. AutoGlobe Migration Controller

The AutoGlobe Migration Controller has been developed as part of the AutoGlobe project at the Technische Universität München [10]. The project is currently working in close cooperation with HP Labs Bristol on the topics of adaptive infrastructures and distributed systems. Their research focuses on load prediction, pattern extraction and the enforcement of Service Level Agreements.

4.2.1. Architecture

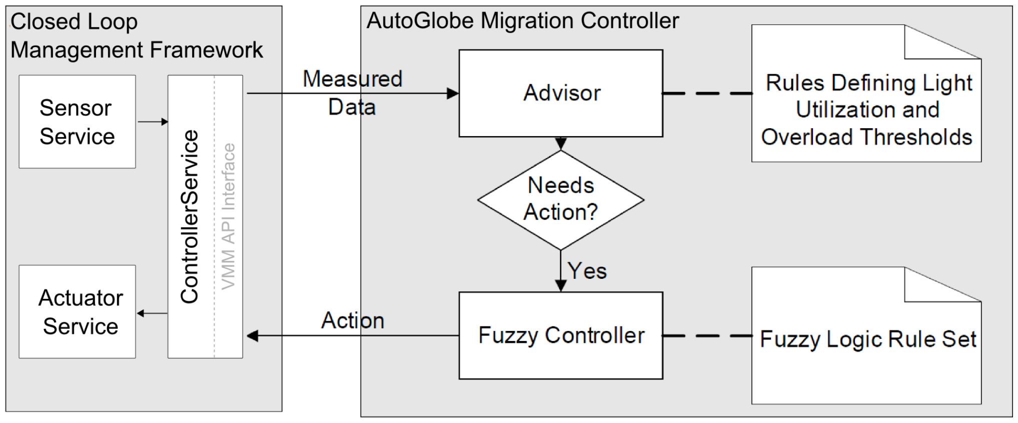


Figure 4.2.: Architecture of the AutoGlobe Migration Controller (adapted from [10])

The AutoGlobe Migration Controller is a reactive resource pool controller working with fuzzy logic. As can be seen in Figure 4.2, it consists of two separate controllers: The Advisor module and the actual Fuzzy Controller. The Advisor assesses the measured data it is receiving by applying simple thresholds. In case one of the thresholds is exceeded respectively under-run, it invokes the FuzzyController to find a possible response.

Configuration files in an XML format define the Advisor's thresholds for over- and underload situations as well as the rule set the Fuzzy Controller uses to find a suitable solution once a critical situation has been identified. The rule set defines a number of different states the input variables can be in, depending on their value, along with rules that combine these states to deduct the most suitable solution. Since the controller is based on fuzzy logic, there are no strict thresholds associating values of input variables to states. Instead, *membership functions* determine to what degree a certain variable is a member of any of the states, also called *linguistic terms*. It is very common for an input variable to be the member of more than one linguistic term and the degree of membership of a certain subset of variables to certain linguistic terms determines, which of the rules in the rule set apply. Please see Appendix B.2 for an example of a rule set and [8, 11, 19, 29] for more information on fuzzy logic and how it is distinguished from classical predicate logic.

The controller receives data and communicates with the infrastructure through the Virtual Machine Management API, or VMM API. It provides methods for obtaining performance and configuration data and triggering actions on the infrastructure, irrespective of the type of components it consists of. The left hand side of the figure shows, what a possible connection to an infrastructure can look like.

The functionality of the generic interface of the VMM API has to be implemented by a concrete tool that is able to retrieve the necessary data from the infrastructure and carry out the actions triggered by the controller. For our purposes, we have implemented the VMM API interface as an extension to our Closed Loop Management Framework, to be able to run it with the AutoGlobe Migration Controller. For further details on the VMM API and our implementation of it, please refer to Section 4.3.

4.2.2. Application

Our implementation of the Closed Loop Management Framework relies on the AutoGlobe Migration Controller to find a suitable new placement for virtual machines, when a host is overloaded and local control is no longer sufficient to sustain performance. We provide the controller with the CPU and memory load values of the hosts and virtual machines in the infrastructure and use a rule set that takes these values into account to tackle overload situations. These situations are resolved by migrating workloads away from overloaded machines.

If the CPU or memory load of one of the hosts exceeds the thresholds defined in the configuration file, the Advisor module will invoke the Fuzzy Controller to check whether a migration is applicable and to find a suitable target. Therefore, the Fuzzy Controller first identifies the virtual machine most suitable for migration away from the overloaded host. This is the “smallest” one, in terms of CPU load and memory footprint, that still is significant enough to make a difference to the overload situation, once it resides on a different host. The controller then evaluates the load of all other hosts, to see whether they have enough free capacity to accommodate the virtual machine. If a match can be found, the appendant migration is triggered.

The rule set shown in Appendix B.2 is the full rulebase of the controller used in our case study in Chapter 5. It contains all the variables, membership functions and rules relevant for our case study in Chapter 5. The rule set also makes it possible to consolidate virtual machines onto fewer hosts in an underload situation and the subsequent shutdown of idle hosts. This will be supported by future enhancements of our VMM API implementation, to enable shutting down hosts, and the HPLB Local Controller or possibly a meta-controller, one of which has to be enabled to identify such an underload situation. More information on the AutoGlobe Migration Controller and the AutoGlobe project can be found in [10, 24].

4.3. The Virtual Machine Management API

As part of HP's portfolio of datacenter management solutions, the Virtual Machine Management Pack VMM is offered as an extension to the HP Systems Insight Manager, SIM, a comprehensive monitoring and management solution for large networks and datacenters. The extension adds features and capabilities necessary to use HP SIM in a virtual environment, providing a generic way to perform management tasks and access status and performance information across a virtual infrastructure, regardless of the virtualization technology used. Mainly, it offers a graphical user interface for system administrators. Additionally it supports the use of external applications through an application programming interface (API). More information on these products can be obtained from [12, 15].

We used neither HP SIM nor the VMM Pack, however. Our interest focused solely on the specification of the API that the VMM Pack provides to external application developers. It allows external management tools to retrieve monitoring data from the infrastructure and to perform a set of standard changes to it through HP SIM. Our objective was to mimic this functionality, using the Closed Loop Management Framework. This enables us to use tools with our Framework that have originally been developed to work with the VMM Pack. However, the API is quite extensive, so for now we have only implemented those methods that are actively used by the tools the Adaptive IT project works with, which currently are CompFlu and the AutoGlobe Framework. The part of the interface that has been implemented can be found in Appendix C.3.

The methods the VMM API provides for obtaining performance data of hosts and virtual machines are:

- `PerfData getCurrentVmPerformance(String token,`

  `    String vmId, int avgPeriod)`

- `PerfData[] getCurrentHostPerformance(String token,`

  `    String hostAddr, int avgPeriod)`

The data format used to represent monitoring data is the *PerfData* class. It encapsulates CPU load (but not the other states a CPU can have in a virtual environment, see the description of the HPLB Local Controller in Section 4.1 for details), the number of CPUs (either physical or virtual), information on total and free memory as well as current Network and Harddisk read and write speeds. Hence, we provide sensors for all of these metrics in our framework, where they did not already exist, and extract the corresponding values from the SensorData object and insert them in a PerfData object. Data requests for a host return an array containing the performance data of the virtual machines running on that host as well.

As far as actuations on the infrastructure are concerned, our implementation of the VMM API is limited to migration. It provides the method:

- `int migrateVm(String token, String vmId, String hostAddr)`

The *vmId* identifies the virtual machine; our implementation relies on the controllers using the VMM API to use the same identifier as the rest of the framework. The *hostAddr* represents the target host for the migration. To track the status of the operation, a *JobId* is returned as an Integer value. It can be used to retrieve information on the job. Job management is done through various methods of the API:

- `Job getJobDetails(String token, int jobId)`
- `void cancelJob(String token, int jobId)`
- `void waitFor(String token, String jobId, String[] answers)`
- `Job[] getJobs(String token)`

Details of a job, such as its status or optional error messages, are encapsulated in a *Job* object, which can be retrieved through the *getJobDetails* and *getJobs* methods. Since the jobs themselves are performed asynchronously in the background, it is used to obtain information on the state, progress, end time and optional error messages of the task. Jobs correspond to ActionItems in the Closed Loop Management Framework. We assign a JobId to each ActionItem object generated when an action is requested through the VMM API. This association is stored, to retrieve status information through the ActionItem's feedback methods, when a Job object is later requested through any of the above methods.

The implementation contains various other methods, which can be consulted in Appendix C.3. For the sake of completeness it is to be mentioned that the *token* parameter seen in all of the above methods represents the authentication mechanism of the VMM API. A *login* method starts a new session and provides a token, that has to be passed upon calling any method to prove authentication. Since our implementation currently works with a single session only, the VMM API uses a static token-String containing just the word `token` and does not verify user credentials in the login method.

Any compatible controller can call any method of the VMM API at any time. The necessary data is requested through the framework on the spot and translated to the corresponding data format.

Chapter 5.

Case Studies

The following sections detail how we put the solution presented in the preceding chapters to the test, in a series of scenarios that are very likely to occur frequently in a real enterprise situation. The questions we will investigate are how the SAP system reacts to the changes to its environment, in terms of application response time, and how the performance and stability of the test runs changes when resources are assigned statically.

The next section presents the three scenarios we used and why we think they are representative for an enterprise application. Section 5.2 goes into more detail on the testing environment. This includes what hard- and software we used and how it was set up. In Section 5.3 we show and discuss the results of the test runs we conducted for each scenario, before returning to the initial questions in the last section, 5.4, of this chapter and analysing to what extent the solution is applicable and what lessons can be learned from the results.

5.1. Enterprise Application Scenarios

Traditionally, systems running enterprise applications have been heavily overprovisioned to avoid performance problems. Still, exceptional situations could cause the system to choke, triggering further difficulties. Service providers of the future will face fierce competition concerning both the price they can charge customers as well as the quality and availability of the service they are able to provide. Hence the dilemma is aggravated: They can not afford to waste money on overprovisioning, yet have to maintain a guaranteed level of service at all times. A possible solution to this dilemma is to use virtualization, as it enables maintaining a cost-effective resource pool, that can be flexibly divided between different customers. Resources left idle by one customer can be used to absorb peak load of another customer. Parts of the infrastructure can be powered down when not needed and be kept in reserve for situations where all customers require peak resource allocation.

Our implementation of the Closed Loop Management Framework, as presented in the earlier chapters of this document, aims to find the optimal resource allocation for a virtual machine running an enterprise application, or in fact any other kind of workload. If a host is overloaded, it can also decide on relocating the virtual machine to another host. This can help improving the utilization of a computing infrastructure by absorbing the effects of shifting load patterns of enterprise applications. Whenever the term "Closed Loop Management Framework" or just "framework" is used in the remainder of this chapter, we are referring to this implementation of the framework.

To examine how effective this solution works, we are creating simulations of several situations we believe to be typical challenges and propose the following scenarios:

1. The Gradual Ramp-Up Scenario

2. The Repeated Ramp-Up Scenario

3. The Host Overload Scenario

Each of these scenarios incorporates two virtual machines, which are initially active on the same host. They run separate installations of SAP R/3 that experience increased load as users gradually log into the system. At the same time, the Closed Loop Management Framework provides resource and relocation management. We have conducted a separate test run without closed loop control for the first scenario, where the amount of resources allocated to the virtual machines is set statically to the averaged value of allocations during the controlled test run of the same scenario.

The users are simulated using HP LoadRunner. This is one of the leading application performance benchmarking suites. It simulates the users through an installation of the SAPGUI, the SAP client-software that a human user would also employ to interact with the SAP server. Each user continuously performs a cycle of SAP transactions that has to be defined beforehand in a LoadRunner script. We chose the sales and distribution (SD) module, as it is also used in SAP's own, widely recognised Sales & Distribution Benchmark for the performance of an SAP system [23].

It consists of a series of transactions that are typically used in a sales and distribution environment. A single transaction is usually composed of multiple steps, like filling data into fields and tables and pushing a button to submit the data. The transactions of the SD benchmark involve composing an order, assembling the corresponding shipment and creating the invoice. Each cycle of the SD process is made up of these transactions and think times, which are pauses in between steps to simulate time a human would spend thinking or entering data. The think times of a whole cycle amount to about 2:20 minutes and the application response time typically to a total of around 10 seconds, making the duration of one iteration 2:30 minutes. Thus in the course of a test lasting about 50 minutes, each user will perform up to 20 iterations of the SD process defined in the script, depending on when the user enters the test.

5.1.1. Gradual Ramp-Up Scenario

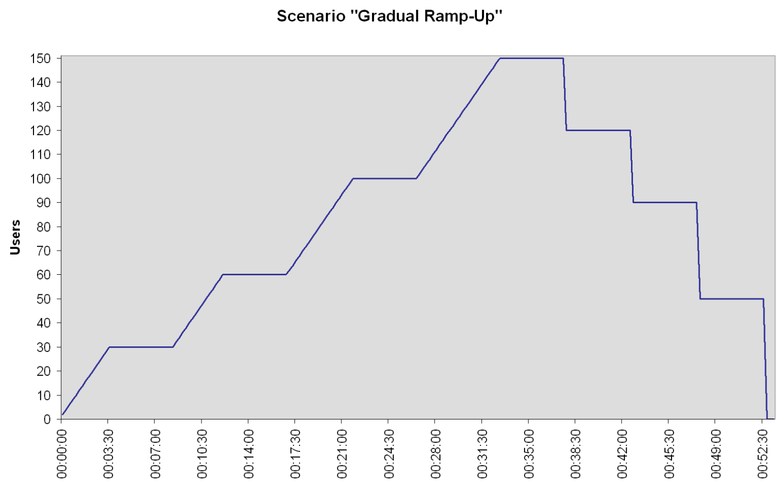


Figure 5.1.: User chart for Scenario 1 “Gradual Ramp-Up”

The *Gradual Ramp-Up Scenario* will see a total of 150 users logging into the system. Every 15 seconds two users will enter the scenario until 30 users are present. The ramp-up then pauses for five minutes, before an additional 30 users start to log in. This is repeated with 40 and 50 users until the total number of 150 users is reached. Hence, we have users logging into the system over a period of 32:45 minutes. The maximum amount of users runs for 5 minutes, until the users are gradually logging out of the system, once they have finished their current iteration of the SD cycle, in the order they have logged in. The amount of active users is plotted in Figure 5.1.

This scenario features a relatively mild increase in the load on the system. Nevertheless, it is very unlikely in real terms that human users log onto the system exactly at the same time, giving this scenario very real significance. In the next scenario, we show that the framework is also capable of handling a steeper ramp-up.

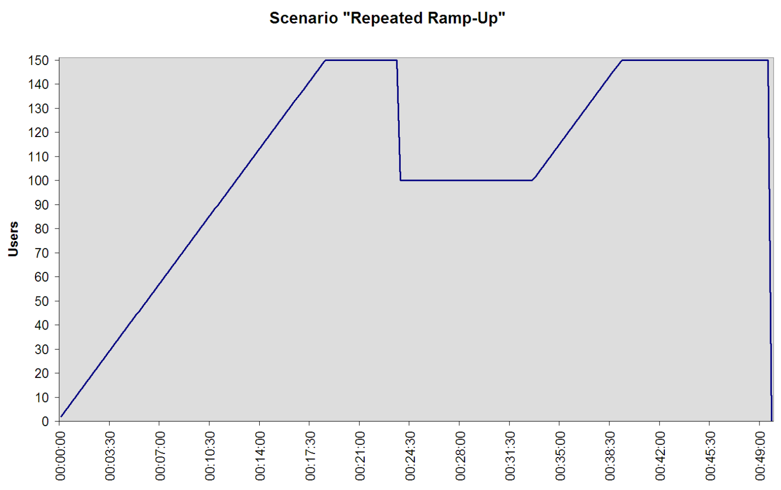


Figure 5.2.: User chart for Scenario 2 "Repeated Ramp-Up"

5.1.2. Repeated Ramp-Up Scenario

The *Repeated Ramp-Up Scenario* starts off with a linear increase in the number of users. Every 15 seconds two users will enter the scenario until a total number of 150 users has been reached after 18:30 minutes. After 5 minutes of running the maximum amount of users, 50 users finish their cycle and log off, reducing the number of active users to 100. After a further 10 minutes of running the reduced amount of users, the 50 users log on again, two every 15 seconds. See Figure 5.2 for a plot of the amount of active users.

This scenario is more challenging, featuring not only a steep linear increase of load on the system, but also a sudden drop in load, which is followed by a repeated increase. This allows us to show that the framework is capable of handling faster increases as well as how quickly it adapts to changing load patterns.

Both scenarios presented so far produce load that can be handled very well by one host alone. To show how the framework resolves an overload situation on a host, we increase the load further in the next scenario.

5.1.3. Host Overload Scenario

Although the Closed Loop Management Framework includes a Migration Controller to relocate virtual machines if a host is overloaded, both the scenarios presented so far have been designed to be sustained by a single host, even if two virtual machines are used. In the *Host Overload Scenario*, the amount of users is increased to a level exceeding the capacity of one host. At the same time, a second host in the environment is idle and is designated as target for a possible migration.

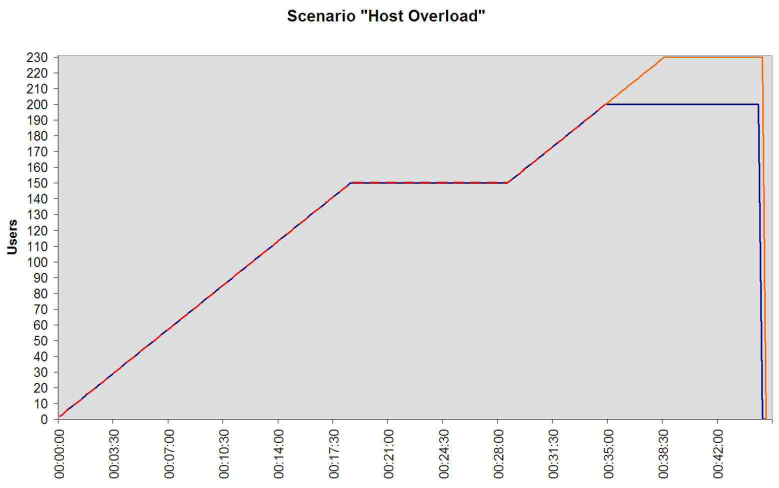


Figure 5.3.: User chart for Scenario 3 “Host Overload”

The number of users in the scenario increases by two every 15 seconds on both VMs until it reaches 150. It remains on this level for 10 minutes, until further users log onto the VMs, until the number of users reaches a maximum of 200 respectively 230. The number of users per virtual machine is plotted in Figure 5.3.

After 36 minutes, a total of 430 users are active in both SAP systems, which is well beyond the capacity of just one of the physical machines used in our study. Given that a second host is available and running idle, it is reasonable to perform a live migration of one of the two virtual machines to the other host at some point, so that each host is operating within its limits again. Ideally this should occur before performance problems arise. We demonstrate when and on what conditions the framework performs such a migration.

5.2. Testing Environment

The term “environment” has already been loosely defined in Chapter 2.4. Here, we want to be more specific about which components exactly have been used for our case study. As can be seen in Figure 5.4, the environment consists of three servers, two virtual machines, the HP LoadRunner benchmarking suite and the SiteScope server application. The two servers to the left are the ones that run the virtual machines containing an installation of SAP R/3 4.72. Their specification is detailed in Table 5.1. The third server, on the right, has a similar specification and serves as a host to the SiteScope server application that is responsible for parts of the load data monitoring for the Closed Loop Framework. The data is also used later in the chapter to present and analyse the test runs.

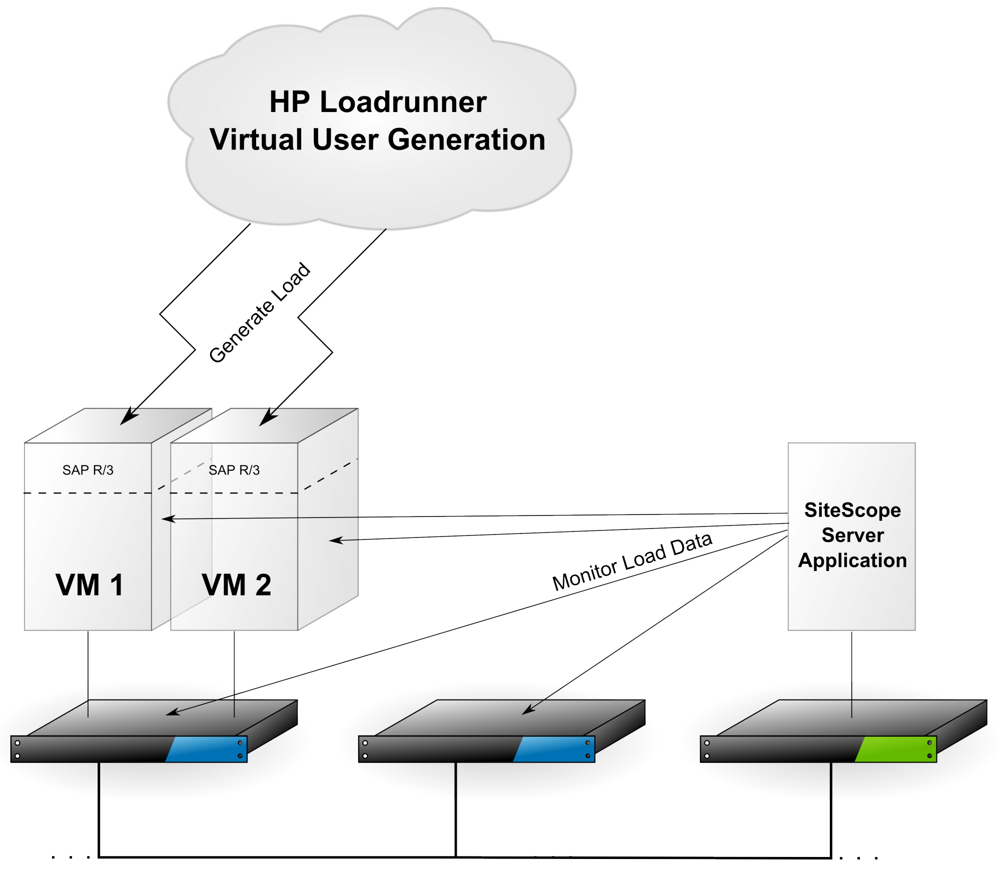


Figure 5.4.: The testing environment for our case study

	Physical Hosts	Virtual Machines	
		Minimum	Maximum
Model	HP ProLiant Bl25p	–	–
Processors	2x AMD Opteron 280 2.4 GHz Dual Core	1 virtual CPU	4 virtual CPUs
Memory	8 GB	800 MB	3 GB
Network	Gigabit Ethernet		
Operating System	SuSe Linux Enterprise Server 10.1 64 bit		
Virtualization Layer	Xen 3.0.3 x64		
SAP Version	–	SAP R/3 Enterprise 4.72 on Max/DB 7.5.0 (Two-Tier Central Instance)	

Table 5.1.: Configuration of the components of our case study

To simplify matters, the Closed Loop Management Framework is not depicted. At the moment it is run from a workstation attached to the same network, but it could as well run on a separate server or share a host with the SiteScope server application. It connects to the SiteScope server application and to the physical host computers to read load monitoring data, but also to trigger changes to the virtual machines. The use of the framework is optional, as the virtual machines can also be configured manually, by connecting to their respective physical host. There is of course no automatic control of resources, if the framework is not active.

Details of the LoadRunner package have been omitted as well. It has a complex architecture, consisting of two controller instances, each running in virtual machines on yet another host. Additional servers provide the computing power needed to simulate the amount of virtual users in our tests. In total five blade servers are involved in our tests. Further information on LoadRunner can be found in Appendix A.3 and [13]. For our purposes it is only important to know that HP LoadRunner is capable of simulating user activity on the SAP systems, that is very similar to the activities of human users.

5.3. Test Results

In total, we have conducted four test runs. One for each scenario and an additional comparison test run of the first scenario with static resource allocation and no closed loop control. Table 5.2 presents an overview of these runs, listing each one's key details. It gives an impression of how well the test ran by listing how many users failed and how many transactions could not be completed during the test. These numbers are linked, since one of the reasons why a user fails is when he can not complete a transaction. This happens when the simulated user encounters a response from the SAP application that is different from what is defined in the script. This can be an error message or an information showing up, basically anything that requires user input not defined in the script. Given that, a failure rate of up to 15% of the users is still acceptable, with lower being better, of course.

The table also shows some key data on the response time of one of the transactions, VA01. It is used to create orders, involving a step to choose the order type and another step to provide information on the buyer and the items the order includes. The other transactions included in the script are VA03, VA05, VF01, VL01n and VL02. They are less complex than VA01, which is why the latter serves as reference mark to us. The table lists the average and maximum response time of the SAP system to the user entries during this transaction and the standard deviation of the response times. The standard deviation is a measure for the variance of

Number of VMs		2	2	2	2
Resource Allocation		static	dynamic	dynamic	dynamic
Virtual CPUs	VM 1	2	1 - 4	1 - 2	1 - 4
	VM 2	2	1 - 3	1 - 2	1 - 4
Memory VM 1 (MB)	VM 1	1550	800 - 2450	800 - 1900	800 - 3000
	VM 2	1550	800 - 1950	800 - 2400	800 - 2500
Errors					
Failed Users	VM 1	7	4	21	30
	VM 2	7	8	11	52
Failed Transactions	VM 1	7	4	22	27
	VM 2	8	7	11	62
Response Time VA01 (in seconds)					
Average	VM 1	50.18	3.967	1.975	2.348
	VM 2	48.135	1.865	1.895	7.165
Maximum	VM 1	130.06	47.28	11.057	10.223
	VM 2	106.183	7.337	11.650	65.135
Standard Deviation	VM 1	21.226	7.715	1.112	0.951
	VM 2	14.915	0.734	0.642	9.036
90% are within	VM 1	79.583	6.090	3.169	3.593
	VM 2	69.219	2.557	2.537	13.817
Duration (Min:Sec)	VM 1	55:48	59:00	54:45	55:42
	VM 2	58:22	59:17	55:22	55:27

Table 5.2.: Overview Scenario Test Runs

the actual values from the average. A lower value means less variance and generally around 65 to 75% of the values will lie within the average value ± the standard deviation.

In particular the maximum response time can be misleading, however. That is why we also provide the upper boundary of the response time for 90% of the transactions. This is much more meaningful than just the maximum value, since a single delayed transaction is less critical than constant high values. Additionally there are a multitude of reasons that can delay some or just one transaction. They can be SAP internal or related to background processes running on the hosts or in the VMs. Of course it can also stem from inadequate resource allocation, as is especially visible in the the case of the test run with static resource allocation.

To visualize the results of the tests, we provide graphs plotting the resource allocation and usage during the test. The y-axis to the left is hereby relevant for CPU usage and number of (virtual) CPUs, whereas the axis to the right is relevant for the amount of total and free memory. Please note that, for better visibility, the values shown in the graphs for the number of CPUs are the actual values multiplied by 10.

Additional graphs show the application response time of the SAP transactions mentioned earlier as well as the number of users that were currently active in the SAP system. As a guideline, a response time of around two seconds for a transaction is acceptable. In practice, much higher response times can occur, but are not regarded as a severe problem when they appear isolated and go back to normal quickly.

In the following section, we first present the results of the test run with static resource allocation and show the drastic effects of resource shortage. This is then compared to a test run of the same scenario, in which the Closed Loop Management Framework controls the resource allocation. In Section 5.3.2 and 5.3.3 the results for the other two scenarios are presented.

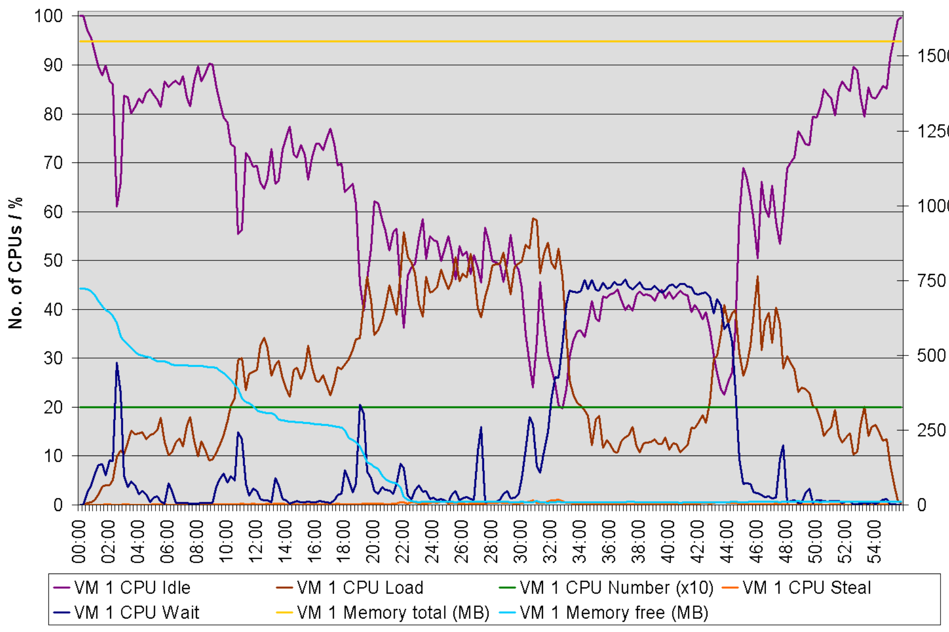
5.3.1. Gradual Ramp-Up Scenario

Static Resource Allocation

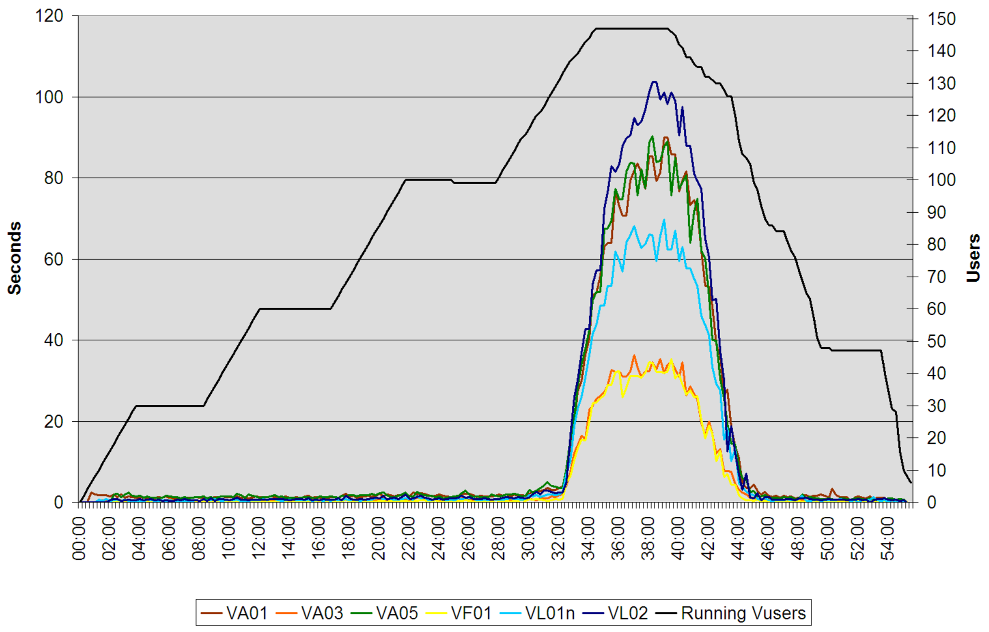
To be able to better assess the results of the later test runs which include the Closed Loop Management Framework, we first conducted a test without closed loop control where resources have been assigned statically. The scenario of this test is the Gradual Ramp-Up Scenario, as presented in Section 5.1.1, and both virtual machines were assigned 1550 MB of memory and 2 virtual CPUs. This corresponds to the average resource allocation that the framework would assign during a run of the same scenario. This is reasonable, given that in a sufficiently big environment with a large number of independent SAP installations, we can arrange the corresponding virtual machines in such a way, that the resource "waste", i.e. resources that are not used by any virtual machine because of the varying resource requirements they have, is minimal.

The results of the test show that the resources are sufficient up to a certain amount of users, in our case around 135. Once this number is exceeded at 32:30 minutes into the test, the system practically comes to a halt, with response times almost instantly surging to 40 seconds and later climbing up to over 100 seconds for VM 1 and 75 seconds for VM 2, see the corresponding Graphs 5.5b for VM 1 and 5.6b for VM 2. The values stay high for over 10 minutes until the number of users drops again. At 45:30 response times have dropped to a level of around two seconds again, when the number of users is at 92. The same happens for VM 2 about two minutes earlier, when 118 users are active.

Looking at the same period in the graphs plotting the resource usage, 5.5a for VM 1 and 5.6a for VM 2, an equally sharp increase in CPU Wait can be observed.

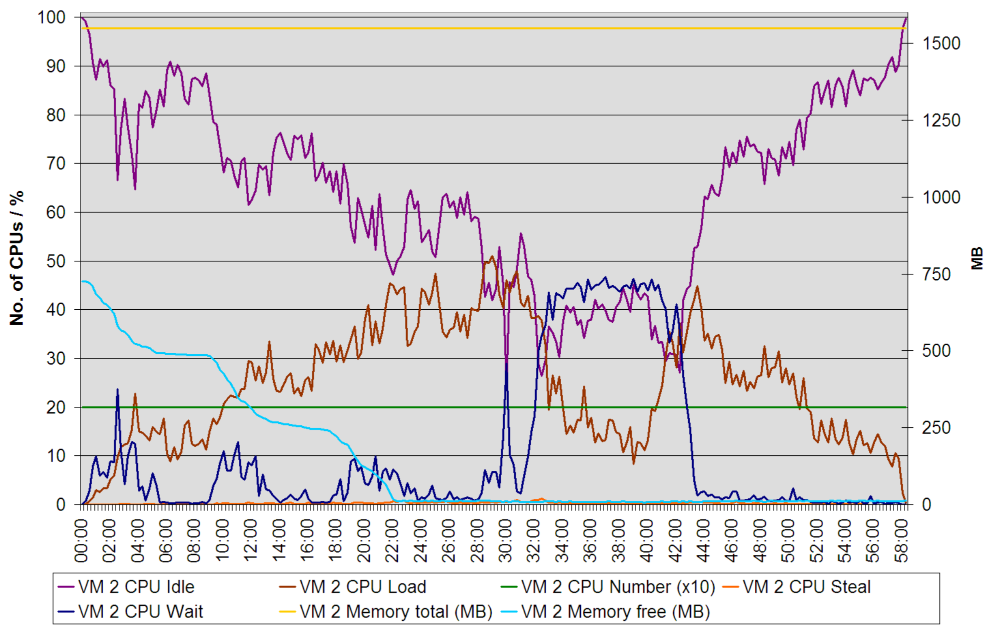


5.5a: Resource Usage

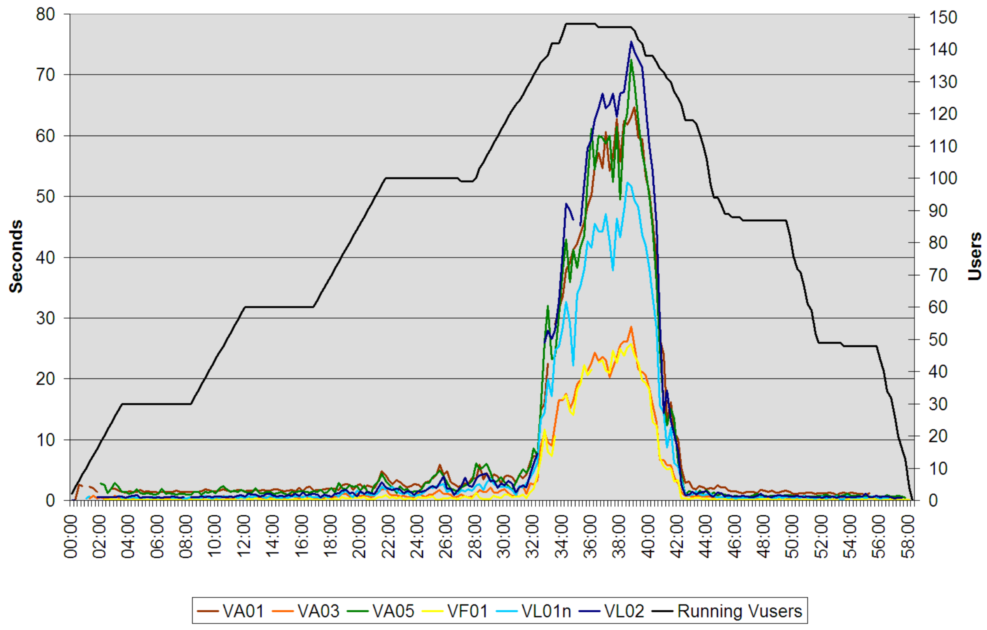


5.5b: Response Times

Figure 5.5.: Scenario 1 – Static Resource Run – Virtual Machine 1



5.6a: Resource Usage



5.6b: Response Times

Figure 5.6.: Scenario 1 – Static Resource Run – Virtual Machine 2

In both VMs, the value reaches 40% at 32:45 which is precisely the moment when response times start to climb in both VMs. At this moment, the available memory is not sufficient any more to simultaneously maintain the data for the sessions of all active users in memory and some of this data is swapped out. It is interesting to note that this happens about 10 minutes *after* the curve for free memory reaches zero. This underlines our previous statement in Chapter 4, that the amount of free memory in the virtual machine is not a valid indicator for the allocation of additional memory. Even after the number of users drops, no memory is freed by SAP. This behaviour is normal and can be observed throughout the other tests as well, shown later, which is why we rely on the percentage of CPU Wait to indicate when to increase and decrease memory.

Dynamic Resource Allocation

After we have seen that a static resource allocation only works well up to a certain point and performance decreases dramatically once more users become active, we will now have a look at the same scenario in a test run where resource allocation is controlled by the Closed Loop Management Framework.

The Gradual Ramp Up Scenario allows us to demonstrate how the framework reacts to increasing and decreasing load. The two Graphs 5.7a and 5.8a on page 56 et sqq. show the CPU usage and resource allocation for both virtual machines. The corresponding application response times in seconds and the amount of active users in the same period is plotted in the Graphs below, 5.7b and 5.8b.

The graphs show how the increasing number of users results in occasional surges in CPU Wait. At these moments, the amount of data for the user sessions can not be stored entirely in the main memory any more and the system begins to swap out parts of its memory. This is counteracted by the Local Controller by increasing the memory of the virtual machine by 200 MB in each case. Should the CPU Wait not fall as a result of this measure, the increase is repeated. When the condition of the system is back to normal again, and the value for CPU Wait drops below the minimum threshold, the controller will start to slowly decrease the amount of memory by 50 MB at a time, at most once per minute.

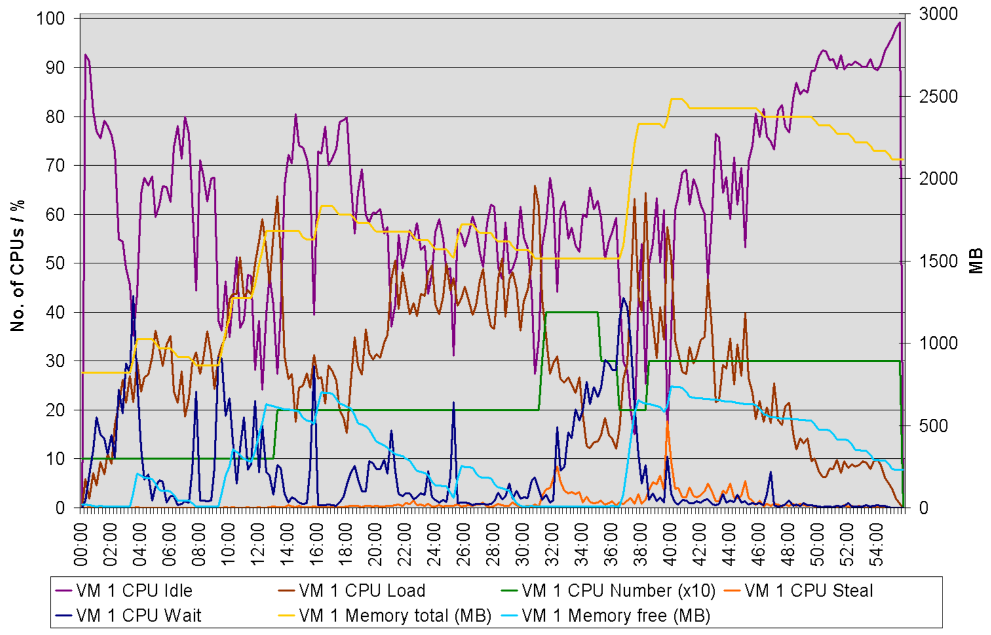
Since the system starts off with a minimal configuration of only one virtual CPU assigned, there will be a point where the CPU power becomes the bottleneck for the application performance. Once the CPU Load rises above its maximum threshold of 65%, the controller allocates another CPU to the virtual machine. As seen in the graph in Figure 5.7a this first happens at 11:45 for VM 2, when 60 users have become active, and a little later for the other virtual machine, at 13:15 and 60 users, in the graph in Figure 5.8a. This results in a drop in CPU Load to approximately half its previous value, since it is measured in percent used of the absolute CPU power available, thus ranging from 0 to 100% irrespective of the number of CPUs.

Toward the middle of the test, when the maximum number of active users is approached, the CPU Load in VM 1 rises faster and higher than in VM 2, even though the amount of users is the same. This could stem from processes active inside the application or on the virtual machine that are beyond the reach of the framework. Looking at the graph in Figure 5.7a, the controller reacts at 31:15 by increasing the CPU allocation to up to four virtual CPUs. Since the other VM already has two virtual CPUs assigned, there now is an overbooking situation, where physical CPUs are hosting more than one virtual CPU. Consequently, some, however limited, CPU Steal occurs. Increasing the number of virtual CPUs for VM 1 succeeds in bringing down the CPU Load, but unfortunately, as a glance at the graph for the response times for VM 1 in Figure 5.7b shows, it does not prevent an exorbitant surge in response time for all transactions during the period from 34:00 to 37:15. Although this is accompanied by a slow but steady increase in CPU Wait, the controller is misled by the relatively high CPU Idle percentage and fails to react until 36:30. A lower share of CPU Idle, below 35%, would have caused the controller to react to a lower CPU Wait of 10% already. See Chapter 4 Section 4.1.2 for an explanation of the Local Controller and the management profile.

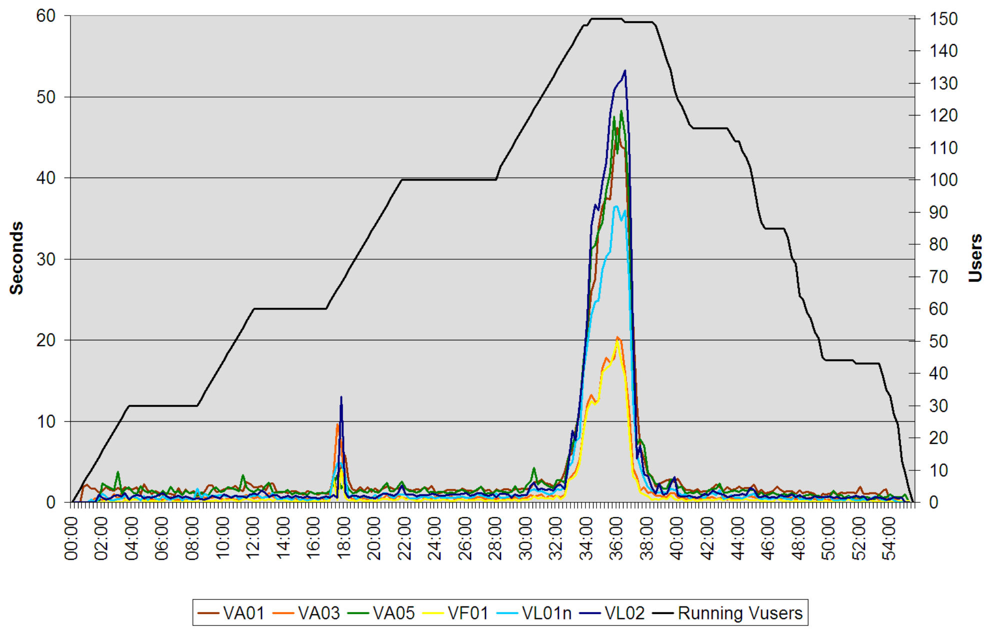
The other virtual machine does not experience such extreme difficulties. In the graph in Figure 5.8b an isolated spike of up to 12 seconds response time appears at 20:00 and lasts until 21:15. This is the apparent reaction to a sharp increase in CPU Wait, which can be observed in the corresponding resource graph in Figure 5.8a. Once it exceeds its threshold the controller increases memory multiple times and the CPU Wait quickly drops, along with the response times.

After the users start logging off, there is a steady decrease of the memory allocation of both VMs by the controllers. This is intentionally much slower than the increase, since removing memory puts a strain on the system. This is due to the fact that SAP usually occupies all available memory, so that it has to be swapped out, when memory is reduced, even though it most probably only contains old user data. To prevent such problems, the Local Controller is configured to remove only 50 MB every minute. Reaching the minimum allocation of 800 MB from the 2100 MB VM 1 still has after the end of the test would consequently require additional 13 minutes. Virtual CPUs are deallocated only after memory has reached its minimum. This is no problem, however, since CPU allocation is non-blocking, i.e. if one virtual machine leaves the CPU idle, other virtual machines can still use it.

Comparing both results of the static test run and the dynamic test run reveals a drastic improvement in the application performance during the dynamic run. Even though a seemingly similar spike in response times is visible in the graph in Figure

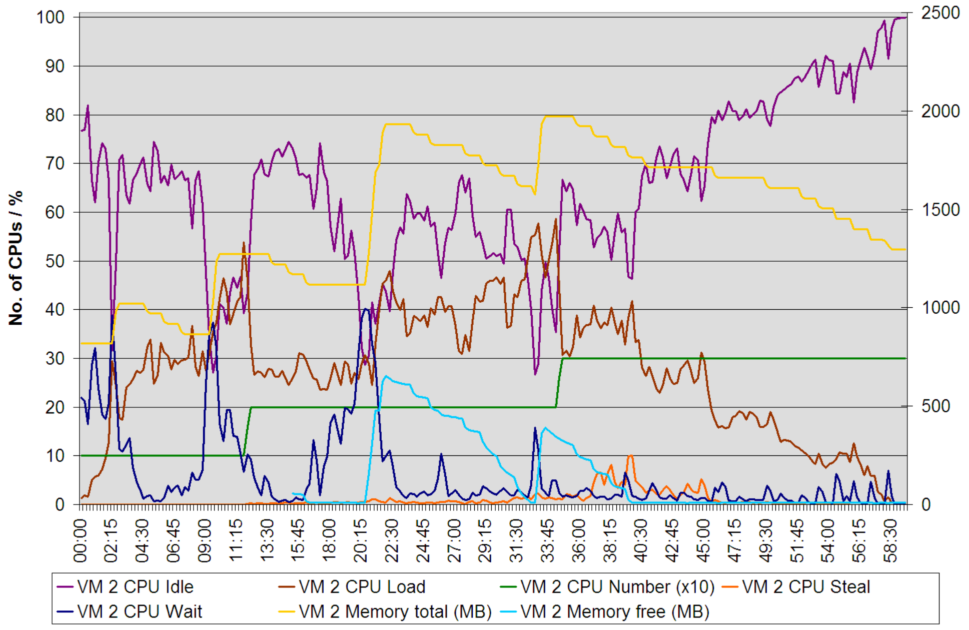


5.7a: Resource Usage

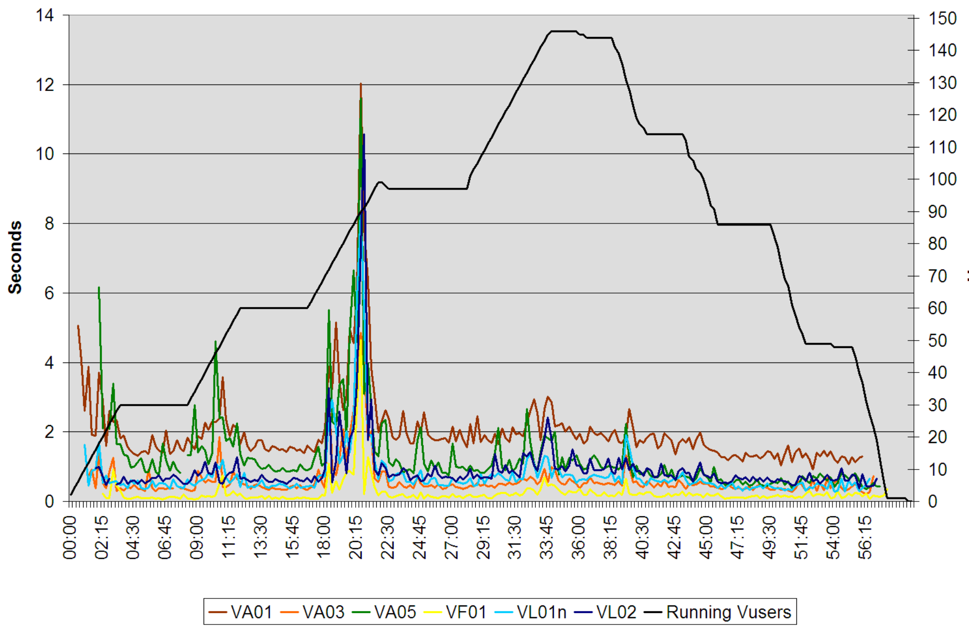


5.7b: Response Times

Figure 5.7.: Scenario 1 – Gradual Ramp-Up – Virtual Machine 1



5.8a: Resource Usage



5.8b: Response Times

Figure 5.8.: Scenario 1 – Gradual Ramp-Up – Virtual Machine 2

5.7b of the dynamic run as in the graph in Figure 5.5b of the static run, they are very different. Not only is the spike in the static run twice as high and lasts almost three times as long, it is also an intrinsic problem of the static setup. As we have identified earlier, the spike in the dynamic run stems from an inappropriate configuration of the Local Controller. Hence an optimized management profile or even a further refinement of the controller's algorithm will possibly minimize such spikes even more in the future.

As a general remark it has to be mentioned that looking closely at the curve plotting the number of active users shows that the maximum number of 150 users for the scenario, as stated in Section 5.3.1, is never actually reached. This is correct, since a user, once it has failed to proceed with its test cycle, drops out of the test run. The total number of failed users has been indicated in Table 5.2.

5.3.2. Repeated Ramp-Up Scenario

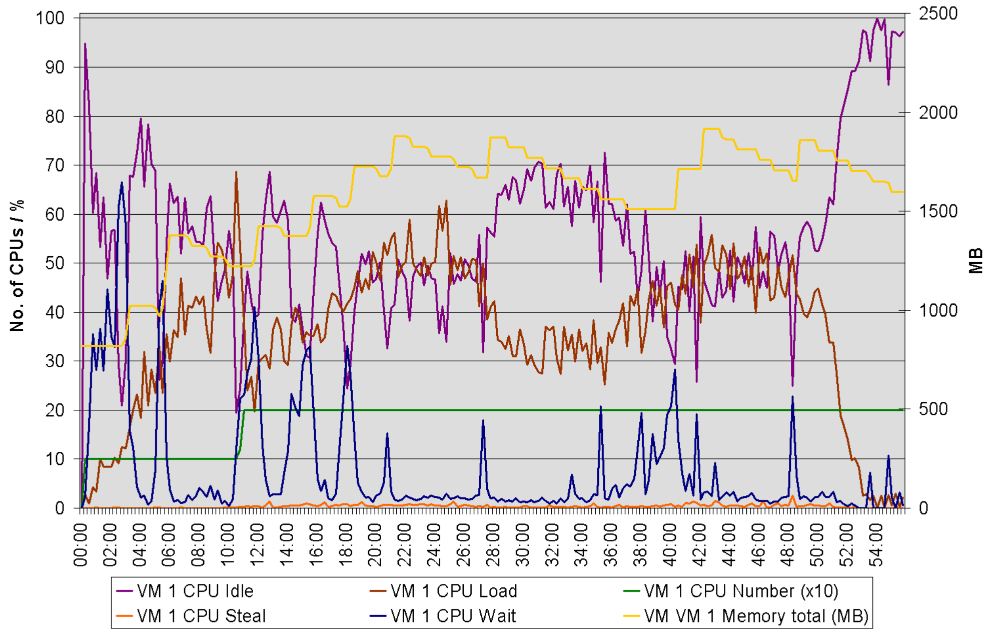
In addition to the two comparative test runs of Scenario 1 we performed further testing in two other scenarios, serving to demonstrate the broader applicability of the framework.

In the Repeated Ramp-Up Scenario the Closed Loop Management Framework is confronted with a faster increase in the number of users, followed by a drop and subsequent rise. We are able to prove, whether the framework can cope with the speedy incline of load and how well it handles changes to the load throughout the test.

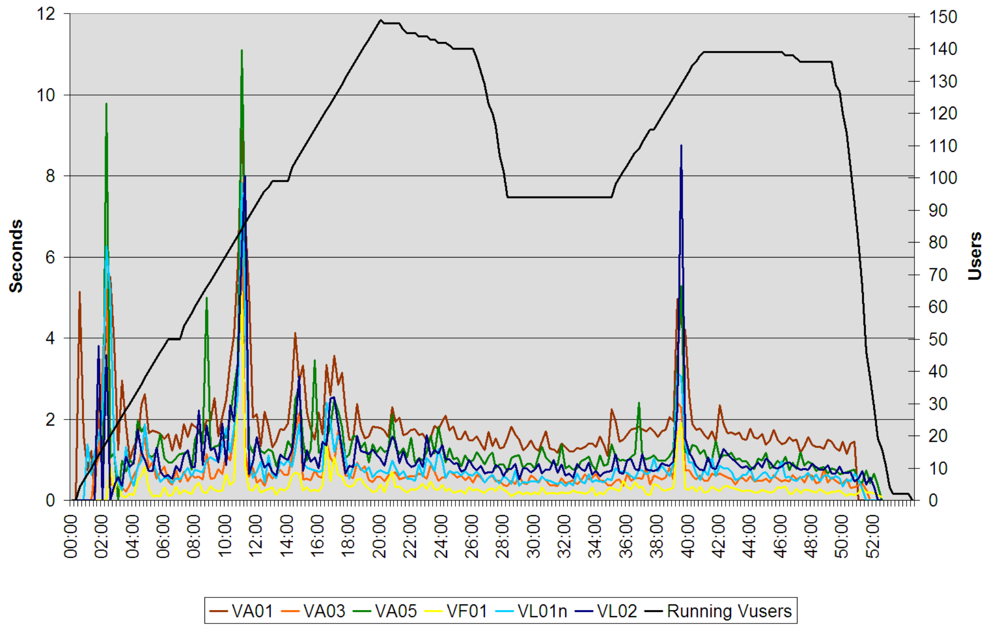
Comparing the figures in Table 5.2 to the previous scenario, reveals that a higher number of users fail the test. However, according to our experience, a failure rate of 10-15% in a load test is acceptable. This is due to the way the tests are run and that the simulated users have no possibility of dynamically reacting to unexpected output by the SAP system or their SAPGUI client. If the same number of human users were active on the system, there would not be such a high number of failures, since the human user could most likely have handled error messages presented by the system.

A closer look at the resource graphs 5.9a and 5.10a reveals that the framework has managed to contain levels of CPU Load and Wait, whereas CPU Steal hardly ever occurs. This is due to the fact, that only two virtual CPUs have been assigned at most to each VM, so that the Xen virtual machine hypervisor on the host machine can always find a mapping of virtual to physical CPUs, that does not involve overbooking.

In the first few minutes of the test VM 1 shows a high CPU Wait right from the start, which leads to rising response times in the graph in Figure 5.9b before

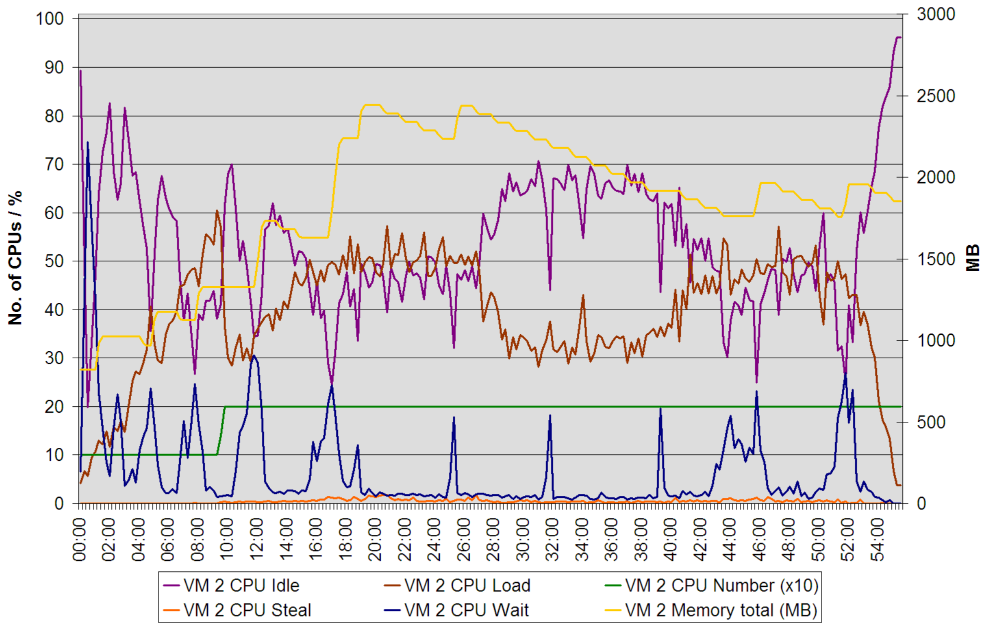


5.9a: Resource Usage

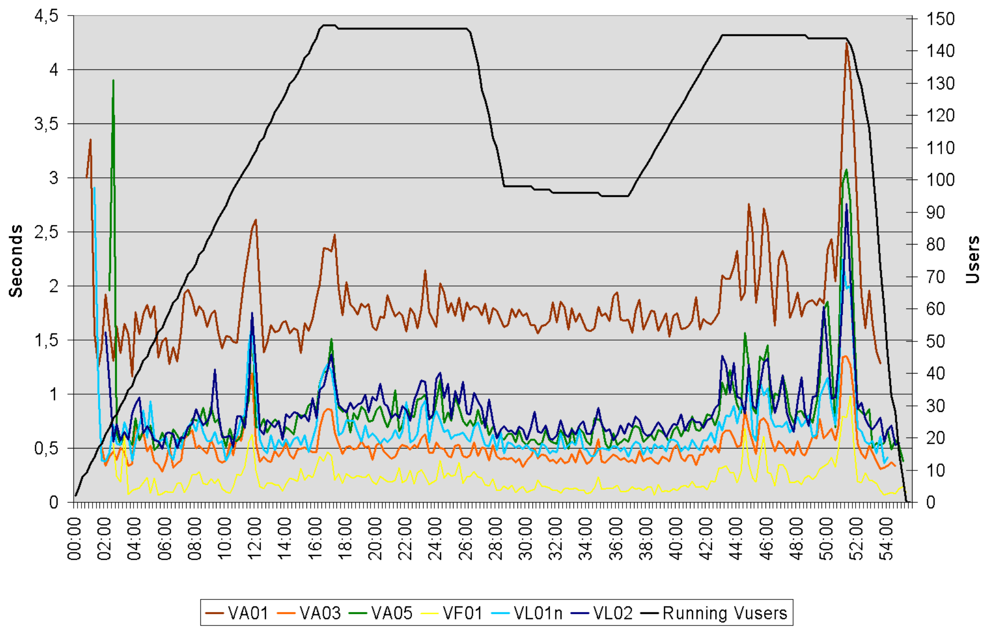


5.9b: Response Times

Figure 5.9.: Scenario 2 – Repeated Ramp-Up – Virtual Machine 1



5.10a: Resource Usage



5.10b: Response Times

Figure 5.10.: Scenario 2 – Repeated Ramp-Up – Virtual Machine 2

the framework is able to correct the situation by increasing the memory allocation. This could be caused by swapping activity already going on in the system before the test was started. When the virtual machine is created, its resource allocation is always at the maximum level (see Table 5.1). To prepare the machine for the test, the allocation is manually reduced to the minimum level. Since the operating system has already occupied parts of the memory, this leads to swapping activity, writing inactive contents of the memory to the disk.

There is a further, smaller spike at 40:00 minutes in the same virtual machine, corresponding to an increase in CPU Wait in the resource graph in Figure 5.9a, which is intercepted by the framework, once it has reached the associated thresholds.

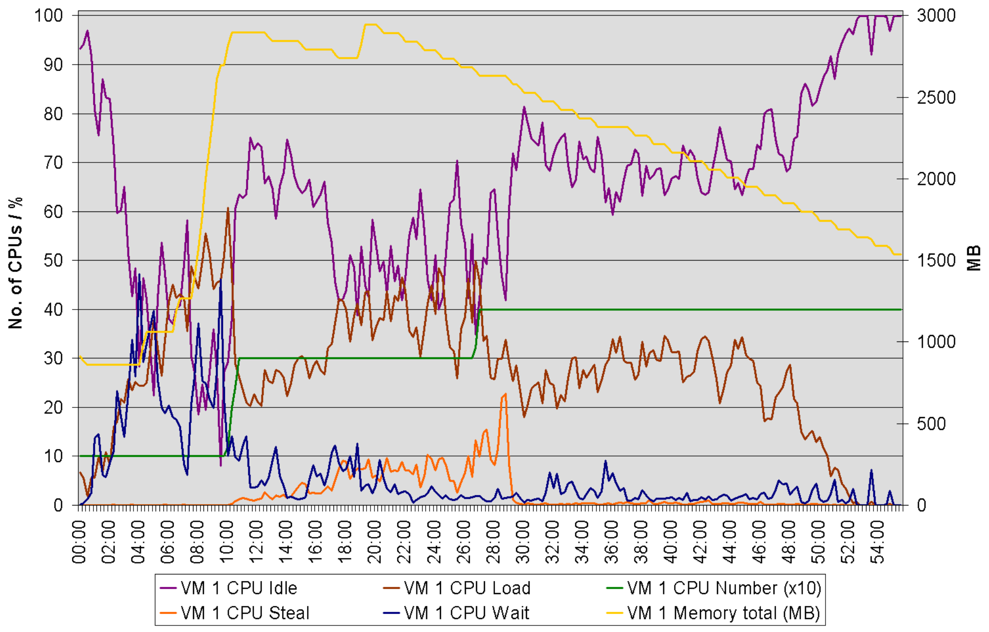
The graphs for the second VM suggest that the framework had little trouble maintaining application performance, since there is no real spike in response time visible in the graph in Figure 5.10b. While it is optimal to have a response time at around two seconds, the values displayed here are still perfectly acceptable. There is a marginal increase toward the end of the test, where the framework seems to have removed slightly too much memory, given that the peak amount of users is still active in the system.

5.3.3. Host Overload Scenario

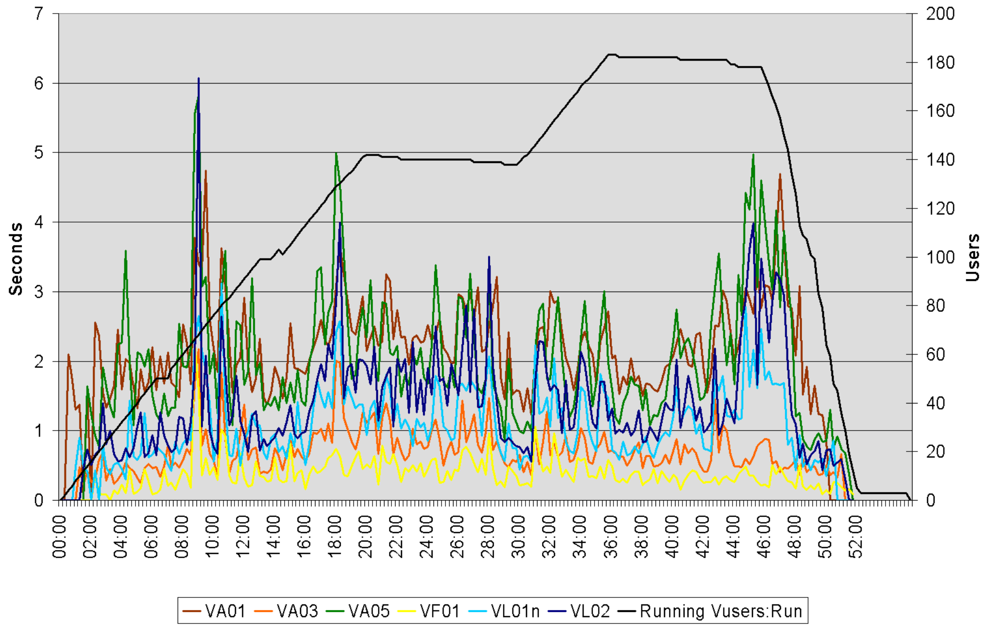
The final scenario, Host Overload, is the most challenging one. It involves a total of 430 users gradually starting on the two SAP systems involved, making sure that at some point during the test the host is overloaded. Such a situation can not be resolved by performing local resource control alone, which is why the Closed Loop Management Framework has been equipped with the AutoGlobe Migration Controller. It monitors the total CPU and memory load on the host and triggers a migration of one of the virtual machines on the host to a less loaded physical machine, in case it detects an overload situation.

As has been described in Section 4.2 of Chapter 4, the AutoGlobe Migration Controller can be configured by providing a configuration file for the advisor module, the controller's gatekeeper, and a rule set for the Fuzzy Controller module. We provide a rule set to identify overload situations and configure the gatekeeper to invoke the Fuzzy Controller when the average CPU load of the previous five minutes on the host exceeds 60%. This value might seem rather low, but it has to be kept in mind that migration produces some additional CPU load and takes a considerable amount of time, depending on the size of the virtual machine's memory and the available network bandwidth, in which the load could increase even further.

Our test run revealed that 60% was a sound choice. The graphs in Figure 5.11 and

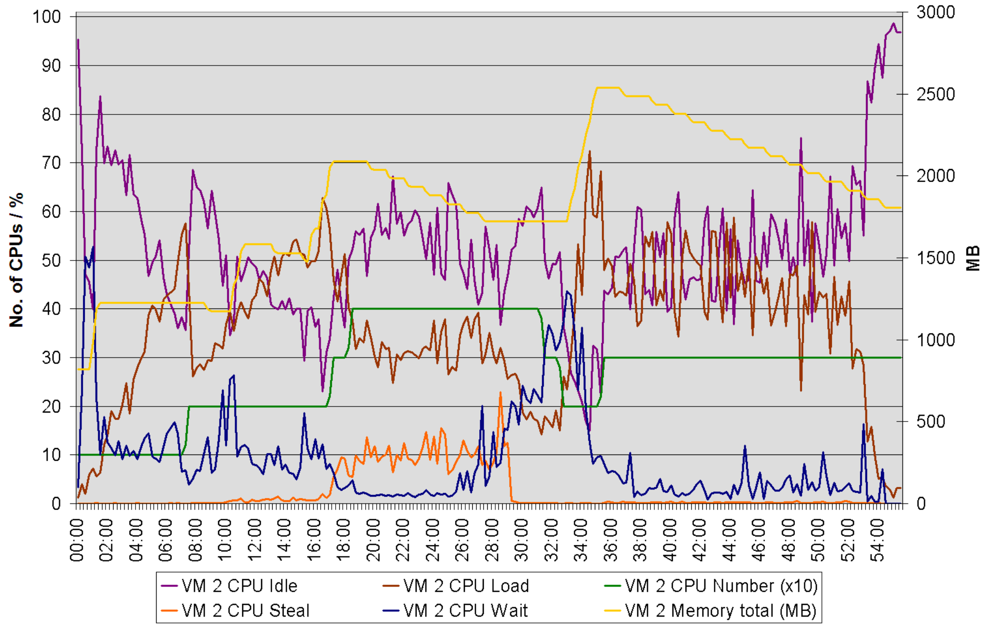


5.11a: Resource Usage

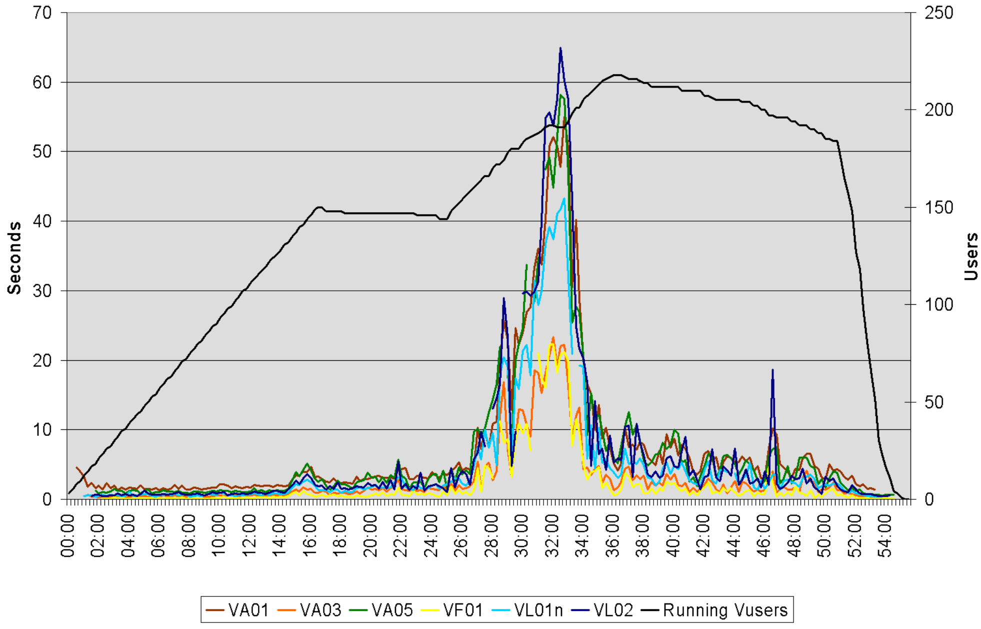


5.11b: Response Times

Figure 5.11.: Scenario 3 – Host Overload – Virtual Machine 1



5.12a: Resource Usage



5.12b: Response Times

Figure 5.12.: Scenario 3 – Host Overload – Virtual Machine 2

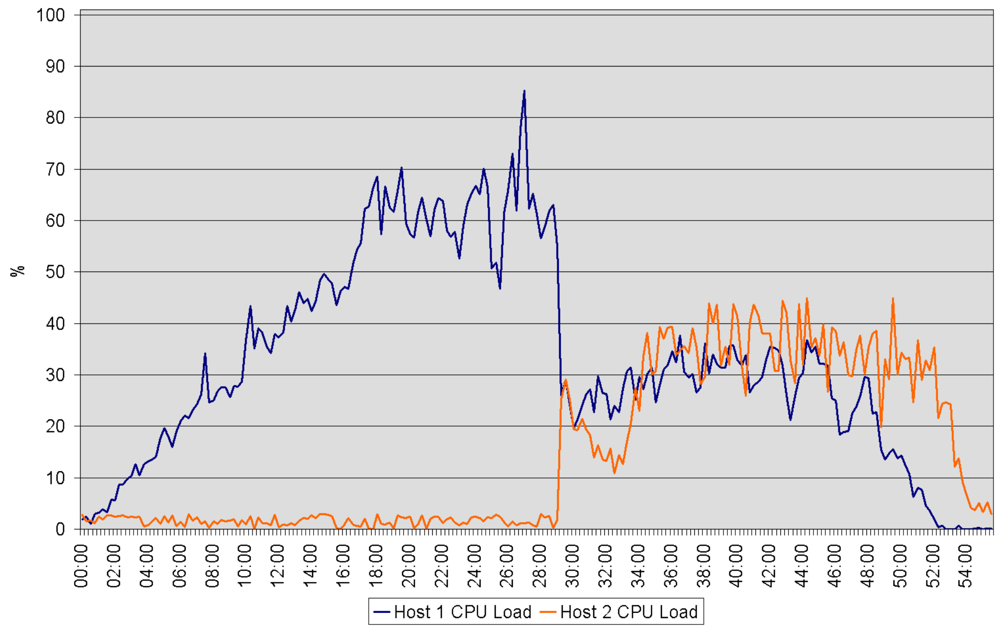


Figure 5.13.: Scenario 3 – Host Overload – Host CPU Load

5.12 show the resource usage and response times for VM1 and VM2 respectively. Since it is of particular interest in this test, the graph in Figure 5.13 shows the load for both hosts involved in the test. Initially, both virtual machines run on Host 1. The critical load on this host is reached 28:15 minutes after the beginning of the test, when 139 users are active in VM 1 and 172 users in VM 2. At the same time, a significant amount of CPU Steal can be observed in both virtual machines, indicating that the virtual machines take turns in blocking the other from accessing the CPU.

The rule set of the Fuzzy Controller specifies that the smallest, but still significant, virtual machine in terms of memory and CPU load is migrated away from the overloaded host. In our case this turns out to be VM 2. It has 1700 MB of memory assigned and about 30% of CPU load at that time, as compared to 2600 MB and 30% in VM 1. It is consequently migrated away to a second, idle host at 28:15 minutes. This migration takes almost 60 seconds until about 29:15. In the resource graphs 5.11a for VM 1 and 5.12a for VM 2, this is visible by the sudden vanishing of CPU steal on both virtual machines after the migration is completed. Both are now alone on their respective host, so there is no more competition for physical CPUs. Additionally, the total load in the test is now balanced across both hosts, as is visible in the graph in Figure 5.13.

The same graph also shows that the load of Host 1 already exceeds 60% several times before the moment of migration. The AutoGlobe Migration Controller deliberately does not react to these isolated spikes, as this would be counterproductive, given that a migration takes some time to perform, uses considerable network bandwith and increases CPU load on the two hosts involved, in our tests by about 10%. Instead the controller works with load values that have been averaged out over a period of five minutes, as has been stated before, to level out the effects of such spikes.

To find out about the impact of this migration on the SAP system, we have a look at the response times around the time of migration and afterwards.

Observing the performance of the SAP system remaining on the original host shows decent response times in the graph in Figure 5.11b, in spite of a brief period of heavy load on the host system, prior to the migration. The virtual machine that is being migrated, on the other hand, shows increasing response times from 27:15 to 34:45 in the graph in Figure 5.12b, even though the actual migration only takes place from 28:15 to 29:15. After the migration, the response times of the second virtual machine slowly go back to a level of five to ten seconds. Given that the SAP system had the computing power of a full host at its disposal, this is not an ideal performance. As we have seen before and in the previous scenarios, a host can easily handle 250 to 300 users.

Looking closely at the resource graph for VM 2 in Figure 5.12a reveals seemingly odd behaviour of the framework right after the migration. In the face of rising numbers of users and response times, the number of virtual CPUs is *reduced* from four to two. Responsible for this is the HPLB Local Controller, which reacts to the low CPU Load of under 20% in the virtual machine, which is completely contrary to the observations that the controller is based on. Why the CPU load of the second virtual machine falls from above 30% before to 10% after migration (factoring out the number of CPUs), inspite of the number of users rising from 170 to over 200 in the same interval, has to be investigated further. From what we know, the most likely source is that Xen is not correctly assigning CPU time to the virtual machine subsequent to the migration.

This theory is backed by messages we witnessed in the console of the virtual machine indicating kernel errors that have occurred during or shortly after the migration. They state that a kernel bug occurred at `drivers/xen/core/smpboot.c`, which could imply a problem with the multiprocessor management, and another one at `drivers/xen/core/evtchn.c`. Further issues included that in similar previous tests the number of virtual CPUs became immutable following a migration. The corresponding command did not change the number any more and thus left the Closed Loop Management Framework struggling to keep the performance within bounds. Furthermore we could observe that the source the framework consults to determine the number of virtual CPUs, `xm list --long`, delivered incorrect results in such a situation. The number given by the command would reflect the changes issued, even though, as other sources, such as the standard `xm list` command, revealed, they had not been carried out. In many cases, the virtual machine would even be unable to further migrate. Issuing the associated command resulted in both hosts getting ready for the migration (with the virtual machine being renamed to “migrating-” on the original host and a new domain being created on the target host), but the migration is never actually being executed.

Whether these problems provide a full explanation for the limited performance that VM 2 showed subsequent to the migration in the test run remains an open question. However these problems, if they prove to be persistent and regular, can without a doubt affect a system in a production environment.

5.4. Summary

The test runs presented in the previous chapter have shown that, in principle, closed loop control of virtual machine resource allocation is possible. The framework manages to retrieve the necessary monitoring data from its various sources. The two controllers, HPLB Local and AutoGlobe Migration, are able to process the data, identify exceptional situations and perform changes to the virtual environment. In particular, the tests show that the SAP software reacts extremely well to these changes performed to its computing environment. At the same time it shows extremely adverse reactions to memory shortage. This has happened occasionally during the tests where the Local Controller either did not react in time or removed too much memory and is even more pronounced in the static run of Scenario 1 in Section 5.3.1.

For the dynamic runs, one has to keep in mind, though, that the management profile was not yet particularly optimized. The peaks in the graphs plotting the response time of the SAP system show some clear potential for improvement. In future tests, we would significantly reduce the thresholds for CPU Wait, since problematic memory lack seems to occur before the current 40% threshold is reached.

Some issues remain with the effects of migration on the response times of the SAP systems. The phenomena observed after migration, in particular the drop in CPU Load in the relocated virtual machine, lead to inappropriate decisions by the HPLB Local Controller that may have had an influence on the performance of the SAP system. A possible workaround could be to block decisions of the Local Controller for a longer period of time after the migration. For our tests, any actions for a virtual machine are blocked for 120 seconds after the migration is initiated, to prevent interference of other changes with the migration.

However, we believe the problem lies deeper, either on the level of SAP internal processes or the Xen virtualization. The configuration of the virtual machine at the relevant time during the test does not justify the poor performance that the SAP system displayed.

The test run with static resource allocation in Section 5.3.1 showed how the SAP system reacts, when the resource allocation is insufficient for the number of users running on the system and it is not adjusted accordingly. The system practically came to a halt with response times reaching over two minutes for as long as the number of users was over the limit for the static resource allocation. This could effectively be prevented when the dynamic resource control of the Closed Loop Management Framework was present. In case there was a problem with response times in the dynamic test runs, we were mostly able to indicate how the closed loop control could be optimized to prevent even these short spikes from occurring.

Chapter 6.

Conclusion and Future Work

6.1. Summary

This thesis has explored the possibilities of applying the theory of closed loop control onto scenarios in which SAP as an enterprise application is run inside virtual machines. To that extent, it has presented the concept of a Closed Loop Management Framework in Chapter 3, which defines interfaces for sensors, controllers and actuators to gather monitoring data, analyse this data and perform changes to the environment. At the same time, an implementation of this framework has been put forward that incorporates the necessary implementations of sensors and actuators to work in the environment of the Adaptive IT project at HP Labs. This made it possible to run the framework with two controllers, the HPLB Local Controller and the AutoGlobe Migration Controller, and present and analyse the results of a series of test runs in Chapter 5.

The HPLB Local Controller is the product of the work of this thesis as well. It has been presented in Chapter 4, alongside the AutoGlobe Migration Controller, which was developed in the AutoGlobe project at the Technische Universität München. The HPLB Local Controller solely relies on data of the CPU usage of the virtual machines to determine appropriate changes to maintain application performance. Compared to performance data measured inside the application, this data can be measured more efficiently and potentially makes the controller applicable to scenarios involving applications other than SAP as well.

The test results presented in Chapter 5 give a promising outlook on the abilities of this solution to be applied to the automated provisioning of enterprise services through the Model Information Flow, as envisaged by the AIT project. The aim of this work was to provide an adaptable mechanism that would concentrate performance relevant data for analysis and distribute management decisions that originate from this analysis. The framework manages both these tasks through the use of exchangeable layers of independent components that provide these mechanisms.

Furthermore, the possibility of managing an infrastructure without highly specific and possibly detrimental application internal monitoring has been proven to work by means of the HPLB Local Controller.

6.2. Future Work

While the solution proposed so far already showed to work considerably well in the scenarios it has been confronted with, there always remains potential for further improvement.

It is necessary to further investigate whether the spikes in application response times we observed during some of the test runs can be attenuated by optimizing the configuration of the HPLB Local Controller, i.e. lowering the thresholds in the management profile or further improvements to the structure and the algorithms of the controller. These could include merging the individual controllers for separate virtual machines into a single instance and incorporating a more advanced state-machine model, as have been suggested among others in Section 4.1.4 of Chapter 4.

The focus for the immediate future should be the integration of the Closed Loop Management Framework into the Model Information Flow of the AIT project. Therefore, some of the mechanisms used in the MIF need to be implemented as extensions to the framework. This includes an actuator compatible to the Request For Change (RFC) mechanism used in the project to dynamically update the current model in the flow and have the changes carried out through a change orchestrator that is invoking the correct tools to execute the changes.

Other tasks considered for the near future are identifying other interesting controllers to be integrated into the framework. This includes the already mentioned CompFlu and ACTS controller, but also the capacity manager CapMan, that predicts future capacity needs by analysing historical data. Especially the latter could prove to be a valuable addition to the exclusively reactive controllers used so far. Integrating further controllers requires careful thought about the interaction between various controllers, however. The proposed proxy-controller should be implemented to intelligently coordinate the management decisions of multiple controllers.

Further research is also needed to better understand the reaction of SAP and other enterprise applications to the changes made possible by virtualization. While SAP seemed to tolerate changing memory and CPU allocation very well, migration remained an issue and did not fully provide the expected performance increase. Whether this is due to a possible problem in Xen, as indicated in Section 5.4 of Chapter 5, has to be investigated more closely. Understanding these stability issues and the general implications of virtual environments for large scale enterprise applications is crucial if this technology is to be applied to production systems.

Chapter 7.

Related Work

Since the drive toward enterprise services is gaining momentum, and offering such services at competitive costs requires sophisticated datacenter management, there are a number of other approaches to the subject.

Some of these have already been mentioned, in particular the AutoGlobe project at the Technische Universität München, presented in [10], of which we use the AutoGlobe Migration Controller. The work of the project spans further, though, striving to create a holistic automation solution for adaptive infrastructures, including fault-tolerance and preemptive measures for self-optimization and self-protection. Their current work focuses on the extraction of load patterns to perform load-prediction, which allows to optimally plan the distribution of workloads over the available infrastructure. Another important topic they pursue is the enforcement of Service Level Agreements (SLAs). Thereby, the assignment of computing resources to different customers with possibly different SLAs can be optimized to incur as little overall SLA infringements as possible.

Further projects include the ACTS and the Computational Fluidity project that have already been briefly mentioned before. ACTS, standing for Automated Control and Tuning of Systems, are also working to create a closed-loop control mechanism for enterprise applications. Their research, presented in [20], focuses on the controller necessary for this task, whereas we have been focusing on providing a complete framework to incorporate into the comprehensive Model Information Flow. The ACTS-controller purely enforces local control and focuses on sharing CPU resources between the different parts of multi-tier applications with Quality of Service in mind. Generally, it follows a much more analytical approach than the HPLB controller.

The Computational Fluidity project, see [16], on the other hand relies on the combination of VMM API and Systems Insight Manager to provide access to sensors and actuators of the virtual environment. It focuses on relocating workloads in order to achieve certain goals, which can be set to optimising performance through load balancing, as in our work, but also server consolidation to economise on energy costs. They perform research focused on heuristic approaches to solve what they call the “Rearrangement Problem”, the NP-hard problem of finding an optimal placement of virtual machines on physical hosts.

Both these approaches are complementary to our work and we are aiming at leveraging their achievements by making it possible to integrate them into the Closed Loop Management Framework.

Another candidate for future integration is the capacity manager CapMan, presented in [22]. Its approach is not reactive, as in our implementation, but relies on analysis of longterm historical data to extract recurring load patterns and estimate the resource allocation needed by a certain workload in the future, based on its past load patterns. This information can be used to trigger changes to the virtual environment *before* problems occur.

A future cooperation of ACTS, the Computational Fluidity project, the developers of CapMan and the AIT project aims at bundling all the efforts for automated datacenter management inside HP Labs.

Similar efforts exist outside of HP Labs as well. An example is *Sandpiper*, presented in [28]. Their focus lies on migration of virtual machines as a measure of adapting to the load they experience. In order to assess the load situation, they contrast a black box approach, similar to ours, to a grey box approach, that uses some data from application logs. However their current work deals with rather simple applications as opposed to complex enterprise applications. Furthermore, they have not got a sound monitoring and actuation framework.

The difficulties of estimating resource requirements of a virtual machine from the outside is discussed in [17]. They observe that it is possible to infer useful information on the memory structure in a virtual machine by observing swap activity from outside the virtual machine. We have a similar problem, since SAP tends to occupy all memory in the virtual machine without ever releasing it. Hence we do not know when the virtual machine needs to be assigned more memory by looking at the free memory value. The “Geiger” mechanism presented in the above mentioned paper gave us the idea to rely on CPU Wait to infer this information, as it is roughly an indicator of swap activity.

There are also commercial solutions to this problem, such as VMware’s Distributed Resource Scheduler (DRS) [26]. It distributes its own monitoring solution amongst the virtual machines to retrieve load data. User-defined policies and rule sets configure its behaviour when load increases. This solution works with VMware’s own virtualisation technology only and uses its live migration technology *VMotion* to enforce the policies.

Appendix A.

Tools Used

The following sections serve to provide a quick glance at some important tools used and mentioned throughout this document. For a more thorough introduction please see the cited references.

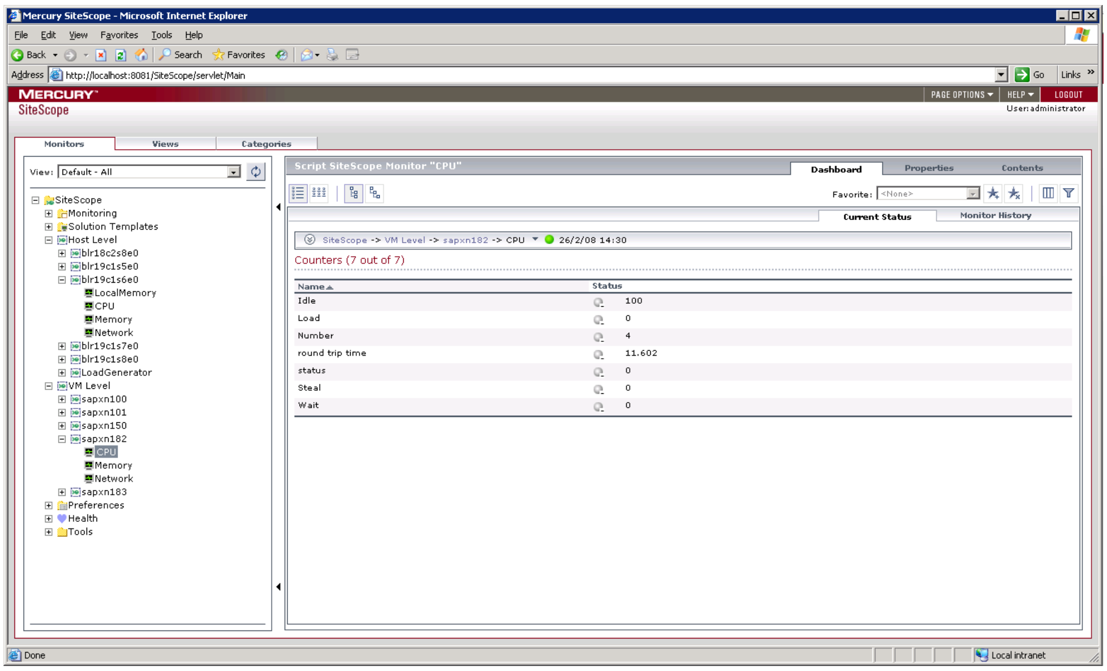
A.1. Xen

Xen is one of the major players in the emerging field of virtualization technologies. It is an open-source project that was founded at the University of Cambridge [2] and later promoted by the company XenSource [4]. More information on the Xen project can be found at [3].

It is a *virtual machine monitor*, an extension to a traditional operating system, that makes it possible to partition the resource of a physical host among several virtual machines. The virtual machines boot an operating system of their own which does not necessarily have to be the same one that is used as the host system. To any application that is run inside it, the virtual machine acts much like a physical computer.

Xen provides two modes of virtualization: Paravirtualization and full virtualization. Whereas full virtualization requires hardware support to run unmodified guest operating systems in the virtual machines, paravirtualization works on any hardware, but requires the guest operating system to be modified, to accommodate virtualization. This is not possible with proprietary operating systems, such as Microsoft Windows, which have to rely on full virtualization to work in Xen. Open source operating system like Linux can be modified, on the other hand, and special releases of certain distributions are available that run in a Xen virtual machine. An example is SuSe Linux Enterprise Server 10, which acts both as host and guest operating system.

A virtual environment of Xen consists of several layers that have different access privileges to the hardware. The lowest layer is the *hypervisor*, managing the


Mercury SiteScope - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8081/SiteScope/servlet/Main
Go Links
MERCURY SiteScope
PAGE OPTIONS HELP LOGOUT
User:administrator
Monitors Views Categories
Dashboard Properties Contents
Favorite: <None>
Current Status Monitor History
Done
Local intranet

View: Default - All

```
SiteScope
  Monitoring
  Solution Templates
  Host Level
    blr18c2s8e0
    blr19c1s5e0
    blr19c1s6e0
      LocalMemory
      CPU
      Memory
      Network
    blr19c1s7e0
    blr19c1s8e0
    LoadGenerator
  VM Level
    sapxn100
    sapxn101
    sapxn150
    sapxn182
      CPU
      Memory
      Network
    sapxn183
  Preferences
  Health
  Tools
```

Script SiteScope Monitor "CPU"

SiteScope -> VM Level -> sapxn182 -> CPU 26/2/08 14:30

Counters (7 out of 7)

Name	Status
Idle	100
Load	0
Number	4
round trip time	11.602
status	0
Steal	0
Wait	0

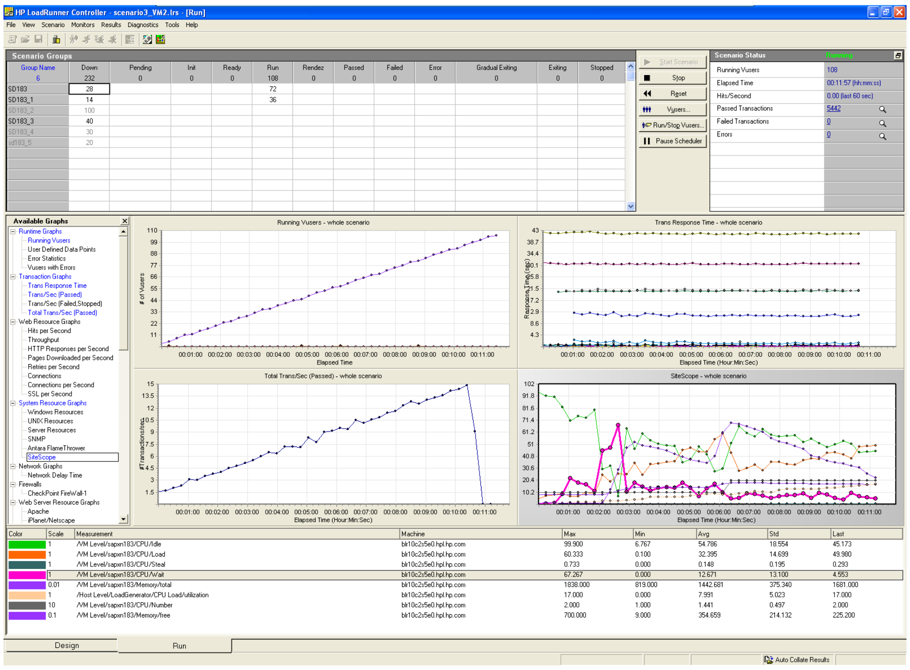
Figure A.1.: SiteScope Graphical User Interface

hardware access of all other layers. It boots the host operating system which is called *Domain-0*. The Domain-0 provides commands to start and manage virtual machines, so called *Domain-U*s. The drivers and kernel of the operating system in a Domain-U have to be modified, since all hardware access of the Domain-Us is performed through the drivers running in the Domain-0. It schedules the virtual machine's access to every component in the hardware, including the CPU.

A.2. SiteScope Monitoring Application

To monitor CPU usage and other monitoring data, we use HP SiteScope 8.5. It is basically a front end for various methods of accessing measurement data. Many monitors are predefined, but it is also possible to define custom ones, based on shell scripts or other commands that can be executed via a remote connection. It allows monitoring Linux-based machines as well as as Windows-based. The monitors can be organized in groups and are managed through a browser-based user interface. The monitoring data can be displayed in the user interface and optionally written to a database. A third way is to access the SiteScope server via http and extract the data in an XML-format. This was used by the framework to read the data.

Figure A.1 shows the browser-based GUI of SiteScope. On the left hand side, the tree structure, based on user-defined groups, for the different monitors is visible.



Scenario Groups

Group Name	Down	Pending	Init	Ready	Run	Rendez	Passed	Failed	Error	Gradual Exiting	Exiting	Stopped
6	232	0	0	0	108	0	0	0	0	0	0	0
SD183	28				72							
SD183_1	14				36							
SD183_2	100											
SD183_3	40											
SD183_4	30											
sd183_5	20											

Scenario Status	Running
Running Vusers	108
Elapsed Time	00:11:57 (hh:mm:ss)
Hits/Second	0.00 (last 60 sec)
Passed Transactions	5442
Failed Transactions	0
Errors	0

Color	Scale	Measurement	Machine	Max	Min	Avg	Std	Last
🟩	1	/VM Level/sapxn183/CPU/Idle	blr10c2s5e0.hpl.hp.com	99.900	6.767	54.786	18.554	45.173
🟧	1	/VM Level/sapxn183/CPU/Load	blr10c2s5e0.hpl.hp.com	60.333	0.100	32.395	14.699	49.980
■	1	/VM Level/sapxn183/CPU/Steal	blr10c2s5e0.hpl.hp.com	0.733	0.000	0.148	0.195	0.293
🟪	1	/VM Level/sapxn183/CPU/Wait	blr10c2s5e0.hpl.hp.com	67.267	0.000	12.671	13.100	4.553
🟪	0.01	/VM Level/sapxn183/Memory/total	blr10c2s5e0.hpl.hp.com	1838.000	819.000	1442.681	375.340	1681.000
🟧	1	/Host Level/LoadGenerator/CPU Load/utilization	blr10c2s5e0.hpl.hp.com	17.000	0.000	7.991	5.023	17.000
■	10	/VM Level/sapxn183/CPU/Number	blr10c2s5e0.hpl.hp.com	2.000	1.000	1.441	0.497	2.000
🟪	0.1	/VM Level/sapxn183/Memory/free	blr10c2s5e0.hpl.hp.com	700.000	9.000	354.659	214.132	225.200

Figure A.2.: LoadRunner Graphical User Interface

The right hand side displays information on the item, that is currently selected on the left hand side. In this case, this is the group representing a virtual machine. Further information on the software's additional features, such as alerts and templates, is available from [14].

A.3. LoadRunner

HP LoadRunner 9.0 is the enterprise application benchmarking solution used to generate the load driving the test runs in Chapter 5. It is compatible to a multitude of business and web applications, including SAP. To realisticly generate the load it provides a recording function that captures human user input to the graphical user interface of the target application, in our case SAPGUI 6.40. The captured actions are transformed into a script which can later be modified to reflect exactly, what kind of user input should be simulated. The script also contains think time, to account for the time a user spends filling out forms or reading text. Measuring application performance is possible by inserting measuring points throughout the script. The time that passes between these points is recorded and shown during the test runs.

To run a load test, a scenario can be defined containing various user groups. Each group can run a different script and contain any amount of users. During the test each user performs the steps that have been recorded in the script. At the same time data about the performance of the tested application is gathered. Additionally, LoadRunner can connect to various other source of monitoring data, including HP SiteScope. All the data is recorded and can be analysed in great detail by the provided analysis tool. It can also be exported to be analysed by an external application.

Figure A.2 shows the LoadRunner GUI during a load test. The different groups and the status of their users are shown in the table on the top. To the left is a summary of the progress of the test run. Several graphs provide an overview of the data that has accumulated so far during the run. More information on HP LoadRunner can be found in [13].

Appendix B.

Configuration Files

B.1. HPLB Local Controller Management Profile

Below is an example of a management profile XML file for the HPLB Local Controller. It contains the default values that would be used already, if no profile was given. The file consists of four parts, dealing with different aspects of the functioning of the controller. For any missing key, the default value will be used, any additional key will be ignored.

The first part, `<Deferrers> ... </Deferrers>`, controls the responsiveness of the controller. In this example, it will immediately trigger a resource increase and wait four iterations before triggering a decrease. This allows the controller to react quickly to sudden load surges, while preventing precipitous removal of resources, which could destabilize the application.

In the second part, `<Metrics> ... </Metrics>`, the respective maximum and minimum thresholds for the four metrics considered by the controller are defined. The third part, `<Resources> ... </Resources>`, defines maximum and minimum values for the three resources the controller can control, as well as the increment and decrement rate. In the default scenario, memory is the critical resource. Thus, the increment rate is much higher than the decrement rate, to rather overprovision the virtual machine at first, which can then later be adjusted by slowly decreasing the memory allocation. It is noteworthy, that the increment rate for the CPU weight is a multiplier. Hence, the weight is doubled when increased.

Finally, the fourth part, `<Attributes> ... </Attributes>`, defines, if the controller should be aggressive or not, where `0` means not aggressive and `1` aggressive. This controls, whether the controller should adjust the scheduler weight and thus the amount of CPU time the virtual machine is assigned compared to other virtual machines using the same CPU. So, in aggressive mode, the CPU allocation is increased at the expense of other virtual machines. In general, this does only make sense for particularly important virtual machines and can be ineffective if applied by multiple controllers for virtual machines on the same host.

```
<ManagementProfile name="default" controller="hplb">
  <Deferrers>
    <defer action="Increase" by="0"/>
    <defer action="Decrease" by="4"/>
  </Deferrers>
  <Metrics>
    <metric name="CPU/Load" max="65" min="25"/>
    <metric name="CPU/Idle" max="80" min="35"/>
    <metric name="CPU/Wait" max="40" min="10"/>
    <metric name="CPU/Steal" max="40" min="15"/>
  </Metrics>
  <Resources>
    <resource name="Infrastructure/Memory" max="3000" min="800"
                                   incRatio="200" decRatio="50"/>
    <resource name="Infrastructure/vcpus" max="4" min="1"
                                   incRatio="1" decRatio="1"/>
    <resource name="Infrastructure/cpu_weight" max="4069" min="64"
                                   incRatio="2" decRatio="2"/>
  </Resources>
  <Attributes>
    <attribute name="aggressive" value="0"/>
  </Attributes>
</ManagementProfile>
```

B.2. AutoGlobe Migration Controller Rulebase

The rulebase defines when and how the AutoGlobe Migration Controller reacts to situations it is confronted with. It contains linguistic variables and rules. Linguistic variables represent an input value to the controller, such as `serverCpuLoad`. The membership functions associate different input values with linguistic terms, standing for different states of the variable, for example `medium`. Rule sets combine different variables and states to evaluate and classify an associated solution. This allows the FuzzyController module to select the best solution.

The idea behind the rules in the example given below is that, once a host is overloaded, which is determined by the Advisor module, one of the virtual machines (`services`) should be migrated away from the host. Ideally, it should not be too small in terms of CPU or memory load or the migration will not have a big impact.

On the other hand, it should not be too large since migrating a virtual machine with a large memory footprint uses more network bandwidth and takes longer to complete. Furthermore, if the workload is too heavy, it might be capable of overloading a host on its own and would consequently be migrated from host to host. The rule set responsible for this decision is `serverOverloaded`.

After the virtual machine is identified, a target host has to be determined. This is achieved through the `move` rule set. In the given example, it classifies hosts as being `ok` or `perfect`. A host is perfect when it is lightly loaded, but not empty. If a host is empty, it is ok, since it also is a valid target for migration. The intention is to work with already active hosts for as long as possible, so that hosts that are completely empty can be put in a power saving stand-by mode. Hence, hosts that have a high degree of being “perfect” are preferred over hosts that are “ok”.

```
<?xml version="1.0" encoding="UTF-8"?>
<rulebase xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation=
          "../../schemata/fuzzycontroller/rulebase-schema.xsd">

  <default group="common"/>

  <group name="common">
   <linguisticVariable name="serverCpuLoad">
      <membershipFunction ll="0.0" lr="50.0" name="low"
                          ul="0.0" ur="0.0"/>
      <membershipFunction ll="0.0" lr="80.0" name="medium"
                          ul="30.0" ur="50.0"/>
      <membershipFunction ll="70.0" lr="100.0" name="high"
                          ul="70.0" ur="100.0"/>
      <membershipFunction ll="0.0" lr="0.01" name="free"
                          ul="0.0" ur="0.0"/>
      <membershipFunction ll="0.001" lr="100.0" name="asLow"
                          ul="0.001" ur="10.0"/>
    </linguisticVariable>
    <linguisticVariable name="serverMemLoad">
      <membershipFunction ll="0.0" lr="50.0" name="low"
                          ul="0.0" ur="0.0"/>
      <membershipFunction ll="0.0" lr="75.0" name="medium"
                          ul="50.0" ur="50.0"/>
      <membershipFunction ll="50.0" lr="100.0" name="high"
```

```
                         ul="90.0" ur="100.0"/>
     <membershipFunction ll="0.0" lr="0.01" name="free"
                         ul="0.0" ur="0.0"/>
     <membershipFunction ll="0.001" lr="100.0" name="asLow"
                         ul="0.001" ur="10.0"/>
   </linguisticVariable>
  <linguisticVariable name="serviceCpuLoad">
     <membershipFunction ll="0.0" lr="30.0" name="low"
                         ul="0.0" ur="0.0"/>
     <membershipFunction ll="0.0" lr="80.0" name="medium"
                         ul="40.0" ur="60.0"/>
     <membershipFunction ll="50.0" lr="100.0" name="high"
                         ul="80.0" ur="100.0"/>
     <membershipFunction ll="0.0" lr="60.0" name="idealForMigration"
                         ul="10.0" ur="15.0"/>
   </linguisticVariable>

  <linguisticVariable name="serviceMemLoad">
     <membershipFunction ll="0.0" lr="30.0" name="low"
                         ul="0.0" ur="0.0"/>
     <membershipFunction ll="0.0" lr="80.0" name="medium"
                         ul="40.0" ur="60.0"/>
     <membershipFunction ll="50.0" lr="100.0" name="high"
                         ul="80.0" ur="100.0"/>
     <membershipFunction ll="0.0" lr="60.0" name="idealForMigration"
                         ul="3.0" ur="15.0"/>
  </linguisticVariable>

  <linguisticVariable action="true" name="move">
     <membershipFunction ll="-100.0" lr="100.0" name="applicable"
                         ul="100.0" ur="100.0"/>
  </linguisticVariable>

  <linguisticVariable action="true" name="server">
     <membershipFunction ll="-70.0" lr="70.0" name="good"
                         ul="70.0" ur="70.0"/>
     <membershipFunction ll="-30.0" lr="30.0" name="ok"
                         ul="30.0" ur="30.0"/>
     <membershipFunction ll="-100.0" lr="0.0" name="bad"
```

```
                          ul="-100.0" ur="-100.0"/>
      <membershipFunction ll="-100.0" lr="100.0" name="perfect"
                          ul="100.0" ur="100.0"/>
   </linguisticVariable>

    <ruleset name="serverOverloaded">
      <rule>
        <condition>
            ((serviceCpuLoad <is/> idealForMigration) <and/>
            (serviceMemLoad <is/> idealForMigration))
        </condition>
        <action>
          move <is/> applicable
        </action>
      </rule>
    </ruleset>

    <ruleset name="move">
      <rule>
        <condition>
            ((serverCpuLoad <is/> asLow) <and/>
            (serverMemLoad <is/> asLow))
        </condition>
        <action>
          server <is/> perfect
        </action>
      </rule>
      <rule>
        <condition>
            ((serverCpuLoad <is/> free) <and/>
            (serverMemLoad <is/> free))
        </condition>
        <action>
          server <is/> ok
        </action>
      </rule>
    </ruleset>
  </group>
</rulebase>
```

Appendix C.

Java Interfaces

C.1. Sensor Interface

```
package com.hp.ait.clm.sensor;

import com.hp.ait.clm.common.SensorData;
import com.hp.common.monitoring.MonitoringEventListener;

/**
 * Sensors for the framework need to implement this interface.
 *
 * It defines methods for adding and removing probes for a system
 * and forces implementation of an internal event trigger mechanism.
 *
 */
  public interface Sensor {

/**
 * used by the SensorService when asked to install a certain probe
 * returns an Integer representing success or failure of the operation
 *
 * @return MonitorOpSUCCESS or MonitorOpFAILURE
 * @model
 */
public int addProbe(SensorData sd);

/**
 * used by the SensorService when asked to remove a certain probe
 * returns an Integer representing success or failure of the operation
 *
 * @return MonitorOpSUCCESS or MonitorOpFAILURE
 * @model
 */
public int removeProbe(SensorData sd);

/**
 * signal to any subscriber the presence of new monitoring data
 * calls MonitoringEventListener.newMonitoringData(SensorEvent)
 * on the subscriber
 *
 * @param sd  SensorData object containing the new monitoring data;
 * will be encapsulated into a SensorEvent
 */
public void fireNewMonitoringDataEvent(SensorData sd);

/**
 * store the MonitoringEventListener in an internal list,
 * so that it can be considered for future events
 *
 * @param mel MonitoringEventListener that provides methods
 * for notification of new events
 */
public void registerEventListener(MonitoringEventListener mel);
}
```

C.2. Actuator Interface

```
package com.hp.ait.clm.actuator;

import com.hp.ait.clm.common.ActionItem;

/**
 * Actuators for the ActuatorService need to implement this interface.
 *
 * It defines, that an Actuator needs to be able to perform an
 * action defined by an ActionItem object.
 *
 */
  public interface Actuator {

/**
 * Perform the action as specified by the given ActionItem object
 *
 * @param ai details the action, contains all necessary information
 */
  public void performAction(ActionItem ai);
}
```

C.3. Virtual Machine Management API Interface

Shown below is an excerpt of the Virtual Machine Management API interface. The full API is far more comprehensive.

```
public interface APIinterface {

/**
 * Cancel the specified job.
 *
 * @param token  String token identifying the session;
 *                given by the login method
 * @param jobId  Integer ID referring to a specific job;
 *                given by any method starting an action
 * @throws RemoteException Exception passed on from Java RMI
 * @throws APIException "Job could not be cancelled" if appropriate
 */
public abstract void cancelJob(String token, int jobId)
           throws RemoteException, APIException;

/**
 * Returns an array of PerfData structures.
 * Element 0 contains the host's performance data,
 * and the remaining elements are for the VMs
 * running on the host, in no particular order.
 *
 * @param token  String token identifying the session;
 *                given by the login method
 * @param hostAddr  String name of the host (hostname in the network,
 *                  for example)
 * @param avgPeriod Integer period in minutes for which to supply
 *                  average data
 * @return array of PerfData structures
 * @throws RemoteException Exception passed on from Java RMI
 * @throws APIException "Token expired" if appropriate
 */
public abstract PerfData[] getCurrentHostPerformance(String token,
           String hostAddr, int avgPeriod)
           throws RemoteException, APIException;

/**
 * Returns performance data for the specified VM.
 * @param token  String token identifying the session;
 *                given by the login method
 * @param vmId  String identifier for the virtual machine
 * @param avgPeriod Integer period in minutes for which to supply
 *                  average data
 * @return performance data for the specified VM
 * @throws RemoteException Exception passed on from Java RMI
 * @throws APIException "Token expired" if appropriate
 */
public abstract PerfData getCurrentVmPerformance(String token,
String vmId, int avgPeriod)
          throws RemoteException, APIException;

/**
 * Get a Host object (see below) for the specified hostAddr.
 *
 * A Host object contains
 * <ul>
 *  <li>IP address of host in dotted notation</li>
 *  <li>Hostname</li>
 *  <li>VirtualizationLayers running on this host</li>
 *  <li>Performance data for this host</li>
 *  <li>CPU clock in MHz of one of the cores of this host</li>
 * </ul>
 *
 * @param token  String token identifying the session;
 *                given by the login method
 * @param hostAddr String name of the host (hostname in the network,
 *                  for example)
 * @return Host object for the host, see above
 * @throws RemoteException Exception passed on from Java RMI
 * @throws APIException "Token expired" if appropriate
 */
public abstract Host getHostDetails(String token, String hostAddr)
           throws RemoteException, APIException;

/**
 * Get an array of IP addresses for all registered hosts.
 *
 * @param token  String token identifying the session;
 *               given by the login method
 * @return array of IP addresses for all registered hosts
 * @throws RemoteException Exception passed on from Java RMI
 * @throws APIException "Token expired" if appropriate
 */
public abstract String[] getHostIds(String token)
           throws RemoteException, APIException;

/**
 * Get a Job object (see below) for the specified jobId.
 *
 * A Job object contains
 * <ul>
 *  <li>jobID for calls like API.getJobDetails(), cancelJob(),
 *      and waitFor()</li>
 *  <li>The date/time when the job completed (ull if the job is
 *      still in progress).</li>
 *  <li>job state (for example "SUCCEEDED")</li>
 *  <li>a descriptive message of what went wrong, if getState()
 *      does not equal "SUCCEEDED"</li>
 *  <li>percent completed of the job</li>
 * </ul>
 *
 * @param token  String token identifying the session;
 *                given by the login method
 * @param jobID Integer ID of the Job
 * @return Job object (see above)
 * @throws RemoteException Exception passed on from Java RMI
 * @throws APIException "Token expired" if appropriate
 */
public abstract Job getJobDetails(String arg0, int arg1)
throws RemoteException, APIException;

/**
 * Returns an array of IDs of all Jobs available.
 *
 * @param token  String token identifying the session;
 *                given by the login method
 * @return Integer array of IDs of all Jobs available.
 * @throws RemoteException Exception passed on from Java RMI
 * @throws APIException "Token expired" if appropriate
 */
public abstract int[] getJobIds(String token)
throws RemoteException, APIException;

/**
 * Returns an array containing all Jobs available
 *
 * @param token  String token identifying the session;
 *                given by the login method
 * @return array containing all Jobs available
 * @throws RemoteException Exception passed on from Java RMI
 * @throws APIException "Token expired" if appropriate
 */
public abstract Job[] getJobs(String arg0)
           throws RemoteException, APIException;

/**
 * Get an array of vmId's for all registered VMs
 *
 * @param token  String token identifying the session;
 *                given by the login method
 * @return array of vmId's for all registered VMs
 * @throws RemoteException Exception passed on from Java RMI
 * @throws APIException "Token expired" if appropriate
 */
public abstract String[] getVmIds(String arg0)
           throws RemoteException, APIException;

/**
 * Create a new VMM API session
 * The returned token string is needed as a parameter
 * for various other API methods.
 *
 * @param username
 * @param password
 * @return a token string if username and password are valid.
 * @throws RemoteException Exception passed on from Java RMI
 * @throws APIException if username/password are invalid
 */
public abstract String login(String username, String password)
           throws RemoteException, APIException;

/**
 * Logout from VMM
 *
 * @param token  String token identifying the session;
 *               given by the login method
 * @throws RemoteException Exception passed on from Java RMI
 * @throws APIException "Token expired" if appropriate
 */
public abstract void logout(String token)
           throws RemoteException, APIException;

/**
 * Initiates a live migration of the specified VM
 * from its current host to the specified host.
 *
 * @param token  String token identifying the session;
 *               given by the login method
 * @param vmId the virtual machine to be migrated
 * @param hostAddr the target host
 * @return jobID for this migration
 * @throws RemoteException Exception passed on from Java RMI
 * @throws APIException "Token expired" if appropriate
 */
public abstract int migrateVm(String token, String vmId, String hostAddr)
           throws RemoteException, APIException;

/**
 * Blocks until the specified job completes
 *
 * @param token  String token identifying the session;
 *                given by the login method
 * @param jobId the Job to wait for
 * @param answers
 * @throws RemoteException Exception passed on from Java RMI
 * @throws APIException "Token expired" if appropriate
 */
public abstract void waitFor(String token, int jobId, String[] answes)
           throws RemoteException, APIException;

}
```

List of Figures

List of Tables

Bibliography

[1] Procps Linux System Tools [online]. Available from: `http://procps.sourceforge.net/`.

[2] Xen at Cambridge University [online]. Available from: `http://www.cl.cam.ac.uk/research/srg/netos/xen/`.

[3] Xen project [online]. Available from: `http://www.xen.org`.

[4] XenSource (acquired by Citrix) [online]. Available from: `http://www.citrixxenserver.com/`.

[5] G. Belrose, K. Brand, N. Edwards, S. Graupner, J. Rolia, and L. Wilcock. Adaptive Infrastructure meets Adaptive Applications. In *HP Tech Con (HP internal)*, 2007.

[6] G. Belrose, K. Brand, N. Edwards, S. Graupner, J. Rolia, and L. Wilcock. Adaptive Infrastructure meets Adaptive Applications at HP Tech Con. Presentation Slides, 2007.

[7] T. Chou. *The End of Software: Transforming Your Business for the On Demand Future.* Sams Publishing, 2004.

[8] W. Dwinell. Putting Fuzzy Logic to Work: An Intro to Fuzzy Rules. *PC AI Online Journal - http://www.pcai.com*, 2002. Available from: `http://will.dwinnell.com/will/Putting%20Fuzzy%20Logic%20to%20Work.pdf`.

[9] European Commission. The new SME Definition – User Guide and Model Declaration, 2005. Available from: `http://europa.eu.int/comm/enterprise/enterprise_policy/sme_definition/sme_user_guide.pdf`.

[10] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper. Adaptive Quality of Service Management for Enterprise Services. *ACM Transactions on the Web (TWEB)*, 2(2), 2008.

[11] P. Hajek. Fuzzy logic. *Stanford Encyclopedia of Philosophy*, 2006. Available from: `http://plato.stanford.edu/entries/logic-fuzzy/`.

[12] Hewlett Packard. HP ProLiant Essentials Virtual Machine Management Pack [online]. Available from: `http://h18004.www1.hp.com/products/servers/proliantessentials/valuepack/vms/`.

[13] Hewlett Packard. HP LoadRunner product page [online]. Available from: `https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17%5E8_4000_100__&jumpid=reg_R1002_USEN`.

[14] Hewlett Packard. HP SiteScope product page [online]. Available from: `https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-15-25%5E849_4000_100__`.

[15] Hewlett Packard. HP Systems Insight Manager product page [online]. Available from: `http://h18004.www1.hp.com/products/servers/management/hpsim/`.

[16] C. Hyser, B. McKee, R. Gardner, and B. Watson. Computational Fluidity using SIM/VMM. In *HP Tech Con poster (HP internal)*, 2007.

[17] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. *SIGPLAN Not.*, 41(11):14–24, 2006.

[18] C. T. Kilian. *Modern Control Technology*. Delmar/Thomson Learning, 2005.

[19] G. J. Klir and B. Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice Hall, 1995.

[20] P. Padala, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, K. Salem, and K. Shin. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *Proceedings of EuroSys*, 2007.

[21] R. Redgate. The Future of IT Delivery, Oct. 2007. Available from: `http://www.idc.com/uk/prodserv/feature_article_2007-10-05.jsp`.

[22] J. Rolia, L. Cherkasova, G. Belrose, P. Vitale, L. Srinivasan, W. Satterfield, and D. Gmach. Capacity Management and Planning for Next Generation Data Centers. In *HP Tech Con (HP Internal)*, 2007.

[23] SAP AG. SAP Benchmark: Sales and Distribution (SD) [online]. Available from: `http://www.sap.com/solutions/benchmark/sd.epx`.

[24] S. Seltzsam, D. Gmach, S. Krompass, and A. Kemper. AutoGlobe: An Automatic Administration Concept for Service-Oriented Database Applications. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE 2006)*, Industrial Track, Atlanta, Georgia, USA, 2006.

[25] I. L. Sriram. Monitoring and Management of Virtualized Infrastructures. Master's thesis, Hewlett Packard Laboratories, 2007.

[26] VMware Inc. Dynamical Resource Scheduler Datasheet [online]. Available from: `http://www.vmware.com/files/pdf/drs_datasheet.pdf`.

[27] P. Wainewright. Engines Emerge That Will Drive Software as a Service. *ASPnews.com*, 2001. Available from: `http://www.aspnews.com/analysis/analyst_cols/article.php/709821`.

[28] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, page 229–242, 2007.

[29] L. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.