DIPLOMARBEIT

An Autonomic Manager for a Java Application Server using Web Service Technology

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs unter der Leitung von

Univ.Prof. Dipl.-Ing. Dr.techn. Hermann Kaindl
und
Wiss.Mitarb.i.A. Dipl.-Ing. Edin Arnautovic
als verantwortlich mitwirkendem Assistenten am
Institutsnummer: 384
Institut für Computertechnik

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Christoph Bernhard Schwarz, Bakk.techn.
Matr.Nr. 0125088
Ahornstrasse 31, 4820 Bad Ischl

Wien, im Mai 2007

Kurzfassung

Die informationstechnologische Infrastruktur bietet heutzutage eine Fülle von komplexen Funktionen. Allerdings wird die Verwaltung dieser Infrastruktur auch zunehmend komplexer. Die *Autonomic Computing* Forschungsinitiative zielt darauf ab, den menschlichen Systemadministrator zu unterstützen, indem das Softwaresystem mehr Selbstverwaltungskompetenzen verliehen bekommt. Diese Funktionalität wird durch einen *Autonomic Manager* realisiert, der dem Softwaresystem als zusätzliche Komponente hinzugefügt wird. Um sie von der Kernfunktionalität des Systems zu trennen sollte die Schnittstelle zwischen *Autonomic Manager* und dem ursprünglichen System (welches dann zum verwalteten Element wird) klar festgelegt sein.

Im Rahmen dieser Diplomarbeit wird die Erweiterung eines bestehenden Softwaresystems — es handelt sich um einen Java Applikationsserver — mit Selbstverwaltungskompetenzen untersucht und umgesetzt. Dem System werden grundlegende Fähigkeiten für Selbst-Konfiguration, Selbst-Heilung und Selbst-Optimierung verliehen. Um dies zu erreichen werden darüberhinaus die hauptsächlich ausschlaggebenden Parameter untersucht. Zusätzlich werden verschiedene Szenarios entwickelt und durchgeführt um diese Fähigkeiten zu testen.

Die Funktionalität zur Überwachung und Einstellung (*Sensoren* und *Aktuatoren*) wird dem Applikationsserver hinzugefügt. Die Schnittstelle zwischen dem Softwaresystem selbst und seinem *Autonomic Manager* basiert auf Technologien aus dem Bereich von *Web Services* und orientiert sich an einem bestehenden Standard für verteilte Verwaltung (WSDM). Diese Technologien werden eingesetzt, weil sie die logische Trennung von *Autonomic Manager* und dem verwalteten Element erlauben. Es ist damit auch möglich, die Verwaltung als verteilte Anwendung auszuführen. Und weil auf einen offenen Standard aufgesetzt wird, ist es auch anderen Verwaltungsapplikationen möglich, mit dem System zu kommunizieren.

Die entwickelte Selbstverwaltungsfunktionalität ist teilweise sehr einfach, aber trotzdem wirkungsvoll. Die Konzepte, welche in dieser Fallstudie verwendet werden, können noch verfeinert und auf andere Systemtypen übertragen werden und in weiteren Untersuchungen der Umsetzung von *Autonomic Computing* zum Einsatz kommen.

Abstract

Today's IT infrastructure provides complex functionality, but also needs complex management. The *Autonomic Computing* research initiative aims at assisting human administrators by making software systems more self-managed. This autonomic functionality is performed by an *autonomic manager* which is added to the software system as an extra component. In order to keep the autonomic functionality separate from the core system functionality, the interface between the autonomic manager and the original system (which then becomes the *managed element*) should be well defined.

This thesis explores and implements the enhancement of an existing software system — a Java application server — with autonomic capabilities. Basic abilities to self-configure, self-heal and self-optimize are added to this system. To accomplish this, the parameters with main impact on the system are also explored. Additionally, different scenarios are developed and executed to test these capabilities.

The functionality for monitoring and tuning (*sensors* and *actuators*) is created. The interface between the system itself and its autonomic manager builds upon Web service technology and respects an existing standard for Web Service Distributed Management (WSDM). Web service technology is employed because it allows the logical separation of the autonomic manager and the managed element. It also enables autonomic management to work as a distributed application. This makes the autonomic manager and the managed system easily exchangeable as well. And since WSDM is an open standard, other management applications can interface with the managed element too.

Some of the added autonomic functionality is simple, but it is effective nonetheless. The concepts used in this case study can be refined and extended to other kinds of systems and also be used in further research on implementing autonomic computing functionality.

Contents

Abbreviations and Acronyms

AC	Autonomic Computing
AE	Autonomic Element
AM	Autonomic Manager
ANS	Autonomic Nervous System
API	Application Programming Interface
AS	Autonomic System
CPU	Central Processing Unit
EJB	Enterprise Java Beans
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IIS	Internet Information Services
IT	Information Technology
J2EE	Java 2 Enterprise Edition
JVM	Java Virtual Machine
MUWS	Management Using Web Services
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SUT	System under Test
TCO	Total Cost Of Ownership
URI	Uniform Resource Identifier
WS	Web Service
WSDL	Web Service Description Language
WSDM	Web Service Distributed Management
WWW	World Wide Web
XML	Extensible Markup Language

1 Introduction and Motivation

Contemporary Information Technology (IT) systems are feature-rich, highly configurable and efficient. But they are also very complex. Deployment and configuration of these systems — both at initial start-up and during the system's lifetime — has therefore become a time-consuming and error-prone task. And it is not only the complexity of a single system that troubles today's IT administrators; it is also the number and diversity of the systems that are in use.

Additionally, modern IT systems are offering a number of networking capabilities. This creates the need to also configure and account for interactions between the different systems in use. And the heterogeneity that results from integration of systems from different manufacturers also contributes to the complexity of managing contemporary IT systems.

One possible way out of this unfortunate development is called "Autonomic Computing" (AC). International Business Machines (IBM), one of the biggest IT companies worldwide, has initiated this research area in 2001 [Hor01]. Autonomic Computing aims at developing computer systems that are capable of managing themselves. Its ultimate goal is to have human administrators specifying high-level goals that the autonomic computing system will try to achieve in the details. The basic architecture of such an autonomic computing system can be seen in Figure 1. It consists of two parts: an autonomic manager and the managed element. Such an approach separates the functionality provided by the managed system from all the logic that implements autonomic computing features.

In this work, a typical IT service — an application server — is extended with autonomic features. The AC features are introduced by a manager that can be placed in a remote location. This makes such an autonomic manager easily exchangeable and its actions observable. And it still respects the basic architecture of Figure 1. It also allows other autonomic managers or management applications to interface with the managed element.

There are various ways to implement communication between two remote systems such as the autonomic manager and the managed system in this case. Preferably, the communication should respect existing standards for the exchange of management information. Furthermore, it should be

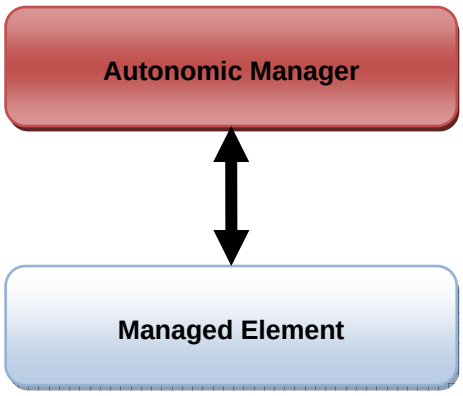

Figure 1: The basic Autonomic System

built in a way that makes it an open system: usable from any platform with an arbitrary programming language and easily accessible over a network.

The implementation work of this thesis, therefore, concentrates on the development of an interface to the managed system and an autonomic manager, which uses that interface. Nevertheless, the managed system has to provide *sensors* to monitor its status and *actuators* to modify its behavior. Obviously, this also requires identification of the fundamental parameters that can be used to tune the system and identification of status indicators to use as monitors for achieving management goals. These two steps can be regarded as common tasks for all systems that are to be managed.

After implementation, the newly added autonomic functionality has to be tested. Freely available benchmarking and testing applications are employed and scenarios for the newly created autonomic element's activity are shown. Finally, the results obtained by implementing such autonomic computing features are summarized.

This diploma thesis is organized as follows: Background information on autonomic computing can be found in section 2.1, followed in section 2.2 by material on Web services, which are used to implement communication. Chapter 3 gives an overview of the current state of typical IT system management, focusing on systems to be managed in section 3.1 and on standards for communicating management information in section 3.2.

Chapter 4 is concerned with the design and implementation of the case study. It thereby first introduces JBoss[1], the enterprise application server upon which the case study is based. Subsequently, section 4.2 describes the autonomic manager that has been developed. The methodology and results of the tests carried out in the case study are presented in section 4.4. Chapter 5 then concludes this diploma thesis with a summary of the lessons learned and an outlook on related and future work.

[1] an open-source implementation of the J2EE specification; see also [FSR06]

2 Background

As the case study conducted for this thesis builds upon terminology and ideas that have been presented in the autonomic computing initiative, this chapter presents autonomic computing more thoroughly. Additionally, web services are discussed, because the management standard used in this work (WSDM) highly depends on the properties of and functionality facilitated by web service technology.

2.1 Autonomic Computing

In October 2001, IBM presented their view on the state of Information Technology, in which they identify the growing complexity of IT systems as a hindrance for future progress [Hor01]. The paradigms of development, configuration and management were seen as insufficient for supporting future computing needs. IBM, therefore, suggested "*Autonomic Computing*" as a solution to that imminent problem.

Autonomic computing (AC) is inspired by the human body's autonomic nervous system (ANS). The ANS, as a part of the nervous system, controls the vegetative functions of the body [Ste05]. These are, for example, blood circulation, digestion, control of body temperature, reflexes, breathing and the production of hormones and other chemical messengers. No conscious activity or effort is required for these functions, and most of them cannot even be influenced willingly. By controlling such vital functions of the body, the ANS frees the human mind from conscious low-level management tasks and makes it available for higher level activity.

Such self-governing systems are quite common in nature and society and range from small molecular machines within cells to human socioeconomy [KC03]. Autonomic computing seeks inspiration in those systems and presents a challenge to create self-governing or *self-managing* systems that can also reach beyond organizational borders. In this way, such AC systems relieve users of low-level management tasks by hiding the complexity and presenting the user with an interface that exactly suits the user's needs. However, not all of the ideas combined in the AC

research effort are new: artificial intelligence and fault-tolerant computing, for instance, have already pursued goals of reliable, self-adapting systems [Ste05].

2.1.1 Self-Management Capabilities

The essence of autonomic computing is enabling self-management. More specifically, four areas of management are of fundamental importance for computing systems [KC03]:

- Configuration
- Problem Solving
- Optimization
- Security

Each of these is addressed by concepts of the AC initiative. The corresponding capabilities that make them possible are described in the following sections.

2.1.1.1 Self-Configuration

Traditionally, complex software systems have to be installed, configured and integrated by highly skilled experts. But these tasks are still very time-consuming and error-prone. On one hand this is caused by the vast number of parameters and configuration options that contemporary systems have. On the other hand, large IT systems as they are today are an aggregation of different technologies and platforms from various vendors, which gives another cause for management complexity.

Self-configuration is the AC system's capability to adjust and readjust itself automatically. It will detect changing circumstances, for example the introduction of a new device on the network or the installation of a new software service, and adapt to its presence. The newly introduced component will use its AC capability to automatically learn the configuration and composition of the overall system into which it is introduced. It will then modify its behavior accordingly in order to properly provide its services.

Another feature of the self-configuration capability is assisting administrators in configuration tasks that cannot completely be carried out automatically. For all other configuration-related tasks, the AC system follows high-level policies that are specified by human administrators or system experts. Finally, self-configuring may also be carried out in assistance to the other self-management capabilities.

2.1.1.2 Self-Optimization

The many parameters that today's complex computing systems have are hard to tune manually. It is not only their number which poses a problem to a human administrator, the effects that the different parameters have may be nonlinear, hard to measure or tightly coupled with other parameter settings. Finding the ideal parameterization hence requires detailed expertise.

An AC system will continually try to find ways to improve its operation. It will seize opportunities to make itself more efficient, both in terms of performance and in terms of cost. This means that the

system can measure current performance and decide whether it is close to its ideal performance. Based on predefined policies, the system is then able to attempt improvements.

Self-optimization allows IT systems to rapidly adapt to changing operating conditions and also to changes in business policies. Hence, administrators are freed from tedious performance optimization tasks and can focus on higher-level improvements.

2.1.1.3 Self-Healing

Because complex IT systems exhibit a large range of different failure behavior, a lot of people are currently involved in identifying, tracing and determining failure causes. Root cause analysis may take a substantial amount of time already, and diagnosis and fixes again need effort.

Autonomic computing's *self-healing* capability is concerned with ensuring effective recovery when failures occur. Upon failure detection, an AC system will take measures to automatically return the failed components or services to a working state. Such systems will also try to repair faulty components and thereby prevent future faults from arising. In this sense, AC systems will use information from system-logs and diagnosis utilities and even install additional monitors to uncover the causes of faulty system behavior.

As system downtime is unacceptable and its cost is damaging to businesses of any size, the rules for self-healing have to be defined very carefully [Mur04]. Excluding the human manager from the decisions to restart a system or parts of it requires confidence and trust to be established into autonomic computing systems.

2.1.1.4 Self-Protection

The corporate world becomes increasingly interconnected. This is also reflected in the IT systems that back the operations of such businesses. Threats, both from external networks and from the inside, have to be taken seriously. Attacks, malware and viruses are increasingly frequent. Even though firewalls and intrusion-detection tools already exist, it is at present still up to humans — who have a certain latency with respect to reaction — to decide how the systems are to be protected and to insure their integrity.

AC systems with self-protection capabilities will defend the system as a whole against malicious attacks and inadvertent cascading failures. This makes it necessary for the system to be aware of threats at the platform, operating system, network, application and Internet level. Applying preventive measures, such as the installation of patches and updates, will also be a responsibility of self-protecting AC systems. Self-protection is perhaps the most ambitious capability targeted by the autonomic computing initiative, because the possibilities for attacks are numerous and not exhaustively definable.

2.1.2 Levels of Autonomic Computing

Because the transition from traditional system management to self-managing solutions based on the AC principles will take place in the course of several years, five stages on the path to fully autonomic computing functionality corresponding to the level of implementation are defined [Mur04]:

1. **Basic level**: The IT infrastructure elements are independently monitored and configured by IT professionals. Any IT environment on the transition will start at this level which involves extensive and highly skilled human resources.
2. **Managed level**: Information from different systems is collected by management tools, which results in great system awareness for the IT staff. This reduces time for analyzing and taking decisions and improves IT administrators' productivity.
3. **Predictive level**: Technologies that provide correlation among several IT infrastructure elements are introduced, which can recognize patterns and provide advice on what actions should be taken. These are approved by IT staff, which results in better and faster overall decision making.
4. **Adaptive level**: The IT system monitors itself and is able to automatically take actions based on the knowledge acquired during the previous level. At this stage only minimal human interaction is needed to manage performance against service-level agreements.
5. **Autonomic level**: Business policies and rules govern the management of IT infrastructure. The processes in the system can be monitored by the users and the objectives altered; management is done by the system itself. This provides agility and resiliency, because IT staff can focus on enabling business needs.

Deciding which maturity level a specific system achieves is a complex task. Consequently methods for assessing a system's status in the AC evolution are in the process of being developed [Ste05].

2.1.3 The basic Autonomic System

The discussion of a possible architecture for an AC system starts with the basic autonomic system which is depicted in Figure 1 on page 2. It consists of:

- ***The Managed Element****:* Any resource of an IT infrastructure can be a managed element. Examples for such an element include:
  - *Single devices,* such as a router or gateway
  - *Services,* such as databases or directory servers
  - *Software components*
  - *Aggregations of related elements,* such as a cluster of servers or a software application made up of multiple components
- ***The Autonomic Manager****:* Collecting, filtering and analyzing of the data obtained by sensors from the managed element is overseen by the autonomic manager. It thereby gains knowledge about the element, which it can use to predict future situations and plan actions to achieve the goals set forth by the overall objective of

the autonomic system. Every autonomic manager executes a basic management cycle consisting of the four phases: monitoring, analyzing, planning and executing as can be seen in Figure 10 on page 41.

In larger scale systems, the basic autonomic system as just discussed is called an *autonomic element*. The overall autonomic system is then made up by several such elements, each of which has its managed resource and an autonomic manager.

Within such a system, the managers will not only need to communicate with the resources they manage, but also among themselves. This is necessary, because the goals which an AC system has to achieve are specified at the higher levels [KC03]. The distinct managers in this case collaborate in the realization of the management target.

2.1.4 Autonomic Computing System Development

The vision of Autonomic Computing is a solution to problems in system operation and integration, but at the same time introduces a lot of new questions [Mur04]. Defining goals for the system on a high level of abstraction, for instance, will on one hand reduce low-level errors made by human administrators. At the other hand, a misinterpretation of such a high-level directive may cause even bigger failures. So it is absolutely necessary to find models and abstractions that are easy to use for humans and can accurately describe the desired behavior of self-managing systems. Additionally, algorithms have to be found that allow the mapping of such abstractions into concrete behavioral rules.

The problems in development of AC systems become even more aggravated for large-scale AC environments. While multiple autonomic managers (AMs), each of them acting as an individual, try to optimize some given aspects of their controlled resource, they need to constantly modify their behavior. As a consequence, a change of behavior of one AM is likely to result in a reaction in behavior from another AM. This might lead to instabilities preventing the overall system from finding its optimum configuration or even to work properly. Problems like this are treated by learning and optimization theory, but while learning in single agent environments is well understood, multiagent systems still are challenging.

Moreover, when different AMs come into play it cannot be ruled out that they will not have conflicting objectives. Hence, these AMs need to negotiate, which creates the need for negotiation protocols and algorithms for such systems. And it has to be taken into account that AMs need to develop some kind of trust-relationship while conducting their multilateral negotiations. In this way, Autonomic Managers can be identified as software agents and results from research on agent-based software systems also apply [Kep05].

But not only is the trust that has to be established between different AMs a topic in AC system development. The user of an AC system — who is imagined to be an IT administrator for now — needs to trust the system to make the right choices, because he eventually hands over control to the AC system. As a result, it has to be possible for the user to understand the actions taken by the AC system [BMK04].

As AC evolves, it will eventually cross traditional corporate boundaries. Communication between different autonomic elements will then only be possible, if the concerned AMs "speak the same language". This means that autonomic computing needs open standards, not only for communication, but also to enable a broad spectrum of developers to contribute to the challenges that AC will continue to face.

All the functionality and properties of AC systems that have been presented in this section are part of the great idea that is autonomic computing. The first systems to emerge with AC functionality are not able to incorporate all of the features that are envisioned. But even a subset of the overall AC capabilities is a definitive advance towards a more dependable, self-managing computing environment.

2.2 Web Services

The need for integration of applications hosted on different computers on a network has brought forth different technologies for inter-application communications. Currently, web services are widely adopted, because they provide the following advantages:

- Not confined to a specific programming language
- Not confined to a specific programming data model
- Scalability
- Easy to set up
- Do not require huge frameworks

The first two of these advantages exist, because web services use the extensible markup language [XML] to encode all the data transmitted. This makes it possible to interface with the services provided using any language on any platform.

Because web services are based on other web technologies, they are scalable and easy to set up. The scalability results from the statelessness of the underlying transport protocol (mainly HTTP). And WS messages easily travel through firewalls, because they use the same protocols as other wide-spread networking applications.

The many different technologies involved in the web service architecture are visualized in Figure 2. At the base of every Web Services architecture is a communication protocol as depicted at the bottom of the figure. Security and Management, which are shown at the left and right border, are outside of the scope of the WS architecture. All WS technology for Messages, Description and Processes builds atop of standards for data representations:

- **XML**: defines the syntax of documents and document elements.
- **DTD**: is a document type that is used to define the types used in XML documents.
- **Schema**: using an XML-document itself, it is used to define types and structure used in XML documents. It thereby replaces the older DTD.

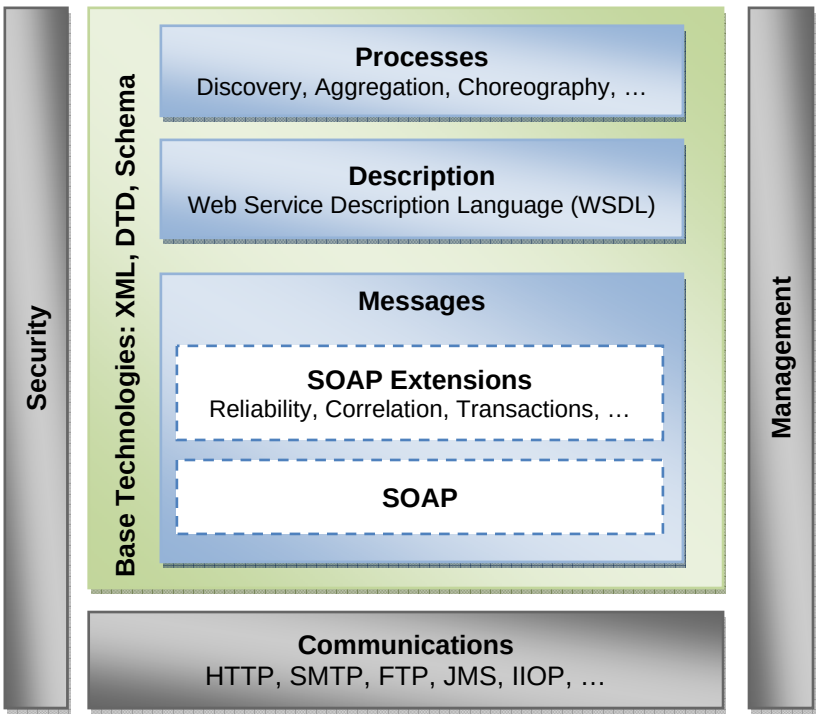

Figure 2: Web Services Architecture Stack (from [WSA])

The communication protocols which web services use to communicate their messages are already existing standard. Virtually every technology that allows the transmission of plain text can be used as communication layer in a WS architecture.

Management and security are not directly addressed by the definitions of web services. These aspects can be handled on any of the other layers of the application in which WS are used. Security, for example, can be realized by using a secure communication protocol, or by implementing security mechanisms in the messages that are transmitted.

The processes associated with the WS architecture stack are outside of the scope in this work. The messages and description technologies are essential and, therefore, discussed in the following sections.

2.2.1 Simple Object Access Protocol

SOAP provides a simple and lightweight mechanism for the exchange of structured data in distributed environments using XML [SOAP]. Because it does not define any programming model or implementation semantics, it can be used in many different kinds of systems, ranging from messaging to remote procedure call.

Messages in SOAP consist of a SOAP envelope, which may contain a SOAP header and must contain a SOAP message body. The envelope is the top-level element of the XML-document which is the SOAP message. If a header is present, it must be the first child element after the envelope. The header can contain a set of header entries as its immediate child elements.

Headers are a flexible extensions mechanism provided by SOAP. The application that sends the message can add any headers it wants to the message. Depending on the receiving application, these headers may or may not be processed. By using the "`mustUnderstand`" attribute the sender can specify headers that it requires the receiver to process. If the receiver is unable to do so, it will respond with an error message. Headers do not have to be intended for the final receiver. They can also be meant for proxies or other application that are involved in message transmission.

The body of a SOAP message contains the actual data that is transmitted from one application to the other. It must be present in any SOAP message and can contain any number of body elements. Three types of SOAP bodies exist [Liv02]:

- Request
- Response
- Fault

A SOAP request message contains the description of the task which the sender would like the receiver to carry out, and the result of which it would like to receive. When SOAP is used as a protocol to carry out remote procedure calls, the body will just consist of the procedure name as first XML element in the body, with the parameters to the procedure appended to the procedure name element as immediate child elements.

The results of the request are sent in a response message. Child elements of the SOAP body are in this case the response elements. These elements have the name of the request element with "`Response`" appended to them. This way, it is possible to distinguish a response message from a request message (without involving SOAP headers).

Because the remote processing of the request may also fail, the third kind of SOAP bodies is fault messages. Such messages have a `fault` element as immediate child of the body element, which is required to have at least a `faultcode` and a `faultstring` as contents. The `faultstring` is a human-readable explanation of the fault, whereas the `faultcode` is used for machine-processing.

SOAP is a stateless protocol and thus needs the three different message types just discussed. Requests and responses can be transmitted separately. If, however, SOAP is used in conjunction with HTTP — which is mostly the case — then the request-response type fashion of HTTP is directly used for the SOAP messages [Wal02].

2.2.2 Web Service Description Language

Web services are concerned with delivering a service of some sort to a service consumer. Because the consumer has to know, how the messages for that service have to be formatted, and how the service can be accessed, some sort of description of the service is needed.

This description should, however, not only be readable by humans, but also allow being processed automatically by a machine. In this way, it can be guaranteed that the client to a service will comply

with all the prerequisites which the service imposes. The description technology used in the web service architecture is the Web Service Description Language (WSDL) as defined in [WSDL].

A WSDL document is an XML document that defines the following elements:

1. Types
2. Messages
3. Operations
4. Port types
5. Bindings
6. Services

These elements are child elements of the root node of the document, whose tag name is "`definitions`". As the web service created for this thesis and discussed below is based on a WSDL description, these elements are further explained in the following to help understand how that web service, based on MUWS (cf. section 3.2.3), is defined.

The `types` section defines all the kinds of data that will be used in any part of the web service. In defining custom types, the syntax of XML Schema has to be used. Instead of defining the types in place, they can also be defined in a separate schema document, and only the reference to that document included in the WSDL types section.

Everything that is to be transmitted between a web service client and the provider of the service has to be defined in the `messages` section. A message in the WSDL consists of any number of message parts, which can either be specific elements as defined in the types section or types. In that last case there is more flexibility in what a message actually contains.

The `operations` section then associates the messages with their function as input or output parameters. For every operation that is to be offered by the web service, a name is defined and the messages that are exchanged in the execution of the operation are mapped. WSDL operations are not required to have either input or output parameters by the WSDL standard. However, if WSDL is used to define a service that will operate using SOAP, every message has to have input and output, even if they may be empty messages. Additionally, every operation can define special fault messages, in case the defaults are not sufficient for the problem at hand.

Any number of operations can then be grouped within a `portType` section. The collection of operations in such a port type defines the complete functionality of a specific web service in terms of possible message exchanges. All the definitions up to and including the port type are abstract information. This means, that no concrete implementation, data format or protocol is associated with a port type and the other elements the port type references. In this way, the web service functionality definition is reusable for different implementations.

The `binding` subsequently assigns a transport protocol and a data format to a combination of messages, operations and port types. The transport is assigned on a per port type basis. Within a port type, every operation gets assigned format information one by one. The encoding of the messages is even defined for every message within the operations. WSDL itself does not provide

the necessary XML tags to define all the possible kinds of associations — it only includes association definitions for web services that use SOAP over the hypertext transfer protocol (HTTP). Any other format or protocol mapping needs custom-defined extensions to WSDL. In this way, WSDL will also be applicable for future WS technologies.

Finally, after the web service has been completely defined, the `service` section defines a port, which is a combination of a port type and its binding with the location, where it can be accessed. Again, WSDL only includes the necessary XML tags and attributes to specify a web location reachable by HTTP. A service in terms of WSDL is then composed of one or more ports. In this way, a WSDL service can contain more than just one web service, but rather specify everything that a certain provider has to offer.

The information contained in a WSDL document is highly redundant. One possible explanation for why the WSDL standard has arranged for such a document structure is that a WSDL service definition can be split into multiple parts, which are then processed by different automated tools that will need to understand the exact semantics of a web service. And since ambiguities must not exist in order to have the service function correctly, the creation of the WSDL document has to be carried out carefully [Liv02]. Fortunately, the WSDL document doesn't have to be entirely created by hand, because automated tools will also be able to translate existing web service implementation code into its WSDL conformant description.

3 Current Management Technology

The management of IT resources is a core task in every organization. Enterprises nowadays depend on a functional IT infrastructure, and computer failures and resulting downtimes can cause tremendously high costs [Pat02]. Of particular interest for this thesis are the services that are typically provided by IT infrastructure and the standard protocols that can be used to communicate management information to and from the systems that provide the services.

3.1 Systems to be Managed

As mentioned in the introduction, the number and diversity of computer systems that are in use today is vast. The selection of systems that are presented here and discussed with respect to their manageability is deliberately limited. But it is assumed that the three types of software servers that have been chosen, namely Web servers, database servers and application servers, are wide-spread in contemporary IT departments and should be representative.

3.1.1 Web Servers

Today's business — as well as private — world has become highly dependent on services provided via the World Wide Web (WWW). The center piece of software upon which this technology is based is the Web server. This server receives client requests and processes them to deliver the demanded content to the clients.

According to the April 2007 Web Server Survey [NC407] the two predominant web server software products are Microsoft's Internet Information Services (IIS) [Mic07] and the HTTP server of the Apache Software Foundation [Apache]. These two together account for almost ninety percent of the web server software in use. While Microsoft's product is commercial and ships with various versions of the Windows operating system, Apache's HTTP server is freely available and its code is under an open-source license.

3.1.1.1 Microsoft Internet Information Services

There are a number of different approaches to manage IIS Version 6.0. The central configuration file, called "*configuration metabase*" is a plain-text file that holds configuration information in XML format. One way of configuring and managing IIS is to directly modify that file. The server will detect changes and apply them while running. Other management interfaces are a GUI application, a snap-in for the Microsoft Management Console (MMC) and command-line administration tools.

IIS can also be managed remotely, either through a web-interface or by logging into the server that hosts the IIS using terminal services. In the last approach, remote configuration can be done through either modification of the XML configuration file or by executing commands of the command-line tool.

Even though they are not called autonomic computing features, some characteristics of IIS could be regarded as showing AC attributes (refer to section 2.1.1 for details on AC features). "Rapid-fail protection" can be regarded as a kind of *self-protection* mechanism. According to the IIS documentation, this feature should help protect IIS against denial-of-service attacks[2]. Additionally, there is a "Health monitoring" feature, that will restart web applications if failures are detected. This in turn can be seen as a *self-healing* feature.

3.1.1.2 Apache HTTP Server

Configuration of Apache is done through a text-file, which contains configuration directives in a format specified in the Apache documentation. Changes to that file will not be reflected by the behavior of a running server, unless it is explicitly told to restart — either fully or gracefully. A graceful restart means that the different threads, which serve client requests, will be terminated once they successfully responded to their clients and will be replaced by a new thread that carries the new configuration.

There are currently no features in the Apache web server that could be seen as enabling AC functionality. But it has to be noted that most functionality that is provided through Apache as a web server is not actually provided by the Apache web server itself, but by extension modules. However, the standard mechanism of configuration for the extension modules is again text-file based and, therefore, not suited for run-time modification to their parameters.

3.1.2 Database Servers

Our modern society is often called "information society". With all the data that is stored in electronic form and almost immediately available, this might well be *the* era of information. But all that data has to be kept somewhere. The software that manages and stores most of the information and allows us to retrieve specific sets of it is the database server or database management system (DBMS).

[2] Denial-of-service attacks are malevolent attempts to make a specific service on a network unavailable to other users. This is mostly done by flooding the service provider (the server) with requests from one or more origins, possibly with fake requests [HHP03].

The market for database servers is very large, with many specialized products that account for specific needs. Since in the previous section a commercial product and a freely-available one were chosen as representatives, this approach is repeated here.

3.1.2.1 MySQL

The MySQL database server is available under the open-source license GPL and under a commercial license. Its documentation [MyS07] covers all kind of management related tasks. None of them, however, are in any way described as automated. There is a basic configuration file with a syntax specific to MySQL that holds most of the server's configuration information. A few variables can afterwards be changed during runtime by means of specialized queries to the database. Nearly all management tasks have to be executed without much support from the system itself.

3.1.2.2 IBM DB2

Since IBM initiated the Autonomic Computing research initiative, it should not be surprising that their database product, called DB2, is more advanced with respect to AC features. The SMART project [LL02] aims at making this product self-managing. DB2 as a consequence already has a few AC capabilities. It shows *self-configuration* features, because it can (since version 7.2) tune about 36 of its most crucial performance parameters to values near their optimum without the need for human intervention.

Self-optimization is constantly done by automatically adjusting the search indexes. There is also a wizard which advises database designers in an effort to create more optimized database layouts. As far as *self-healing* is concerned, there is currently no feature that could directly be seen as implementing such a capability. But DB2 has the so-called "Health Center", which assists database administrators in detection, isolation and correction of problems. Of course, IBM's ultimate target with this feature is to carry out these tasks without the need of administrator interference.

3.1.3 Application Servers

An application server is a middleware computing platform on which component-based software can be developed and deployed. It usually handles most of the business logic and access to data, leaving only presentation to the clients. The use of application servers, therefore, has a little similarity to the old concept of mainframes and terminal clients.

Even though the term "application server" is applicable to any kind of platform, it is mostly associated with the Java platform [J2EE] developed by Sun Microsystems. In this thesis, "Application server" is used as synonym for "J2EE application server".

The use of application servers provides the following benefits over traditional application deployments:

- Application servers provide an API to programmers, so that these are not concerned with operating system details. Applications developed for Java application servers are deployable on any platform for which a JVM implementation is available.

- They contain the communication facilities used to exchange information with the clients over a network, thereby enabling easy remote access of data.
- Since the business logic is located in one central location, applications can be updated without the hassle of having to install updates on every client machine.
- Access to the underlying data store is encapsulated through the application server. This makes improved data consistency and security possible.
- Most application servers also bundle web servers, which means that the business logic and data can be reused for being accessed via the internet.

There are again two different representatives chosen here. The first one is Sun's own implementation of the Java 2 Enterprise Edition specification — the Sun Java Application Server Platform — and the other one a much-used open-source implementation that is now backed by RedHat — the JBoss Extensible Application Server.

Because JBoss is the implementation target for this diploma thesis, research efforts that contribute to adding autonomic computing functionality to the JBoss application server are also discussed.

3.1.3.1 Sun Java System Application Server Platform

Edition 9 of Sun's application server provides the following means of system management [SASE9]:

- A browser-based Admin Console
- A scriptable command utility — `asadmin`
- AMX – the Application Server Management Extension, which is an API that exposes configuration- and management-related JMX managed beans to clients.

However, it is noted in the administration guide, that changes of the following configuration options are not possible without a restart of the application server:

- JVM options
- port numbers
- HTTP services
- thread pools

The browser-based configuration tool offers descriptive text that assists in application server configuration. Monitoring and optimization has to be done by hand or by other (third-party or custom developed) tools. There are currently no self-management features incorporated in the Sun Application Server platform.

3.1.3.2 The JBoss Extensible Application Server

Development of JBoss has originally been done partly as a research effort by a geographically dispersed team headed by Marc Fleury [FR03]. Its first release dates back to 1999, and while it is still open source software, commercial support is available.

The architecture of JBoss is a minimal core server based on the Java Management Extensions (JMX) into which all services and applications are plugged. Since JMX defines a dynamic resource management architecture, components — which can be services and applications on JBoss — become implicitly manageable on JBoss.

JBoss itself does not incorporate any self-management functionality. There are, however, research efforts that can be regarded as implementations of self-healing and self-optimizing capabilities. These are presented below.

3.1.3.3 "Self-Healing" for JBoss

The autonomic computing self-healing capability describes the ability of a system to automatically recover from failures. JAGR — which stands for JBoss with Application-Generic Recovery — is an extension of JBoss to enable such *self-healing* [CKZ03]. It combines three recovery-oriented computing techniques to achieve this:

- application-generic failure-path inference (AFPI),
- path-based failure detection and
- micro-reboots.

Relations between failure causes and manifestations of failures are detected by AFPI, which deliberately injects failures into applications and follows the path along which these failures spread. Using the information obtained by AFPI, the system (JAGR) can determine which components are malfunctioning and need to be restarted.

Path-based failure detection uses statistics to compare client requests recorded over a longer stretch of time. The client requests are classified by tagging and anomalous behavior of the system is detected by statistical analysis.

Micro-reboots are restarts or reboots below the application level. This means that instead of restarting an entire application, which may result in a considerable penalty in terms of computing time, only a small subset of components within that application are rebooted. In component-based systems such as J2EE applications this requires the components to be independently bundled.

JAGR augments JBoss with multiple modules:

- monitoring infrastructure, consisting of
  - end-to-end monitoring,
  - exception monitoring and
  - pinpoint monitoring
- a stall proxy (used to delay client requests during self-recovery activities),
- a fault injector,
- a recovery agent and
- a recovery manager.

With the exception of the recovery manager, all of the components are directly integrated using JBoss' modular extension facilities. The recovery manager executes on a different computer,

because the calculations done by the recovery manager would impose unacceptable performance penalties on the application server.

For JAGR to work properly, the recovery manager's knowledge about recovering possibilities — stored in what is called the failure-propagation map — has to be generated. This is done by running the fault injector with all the applications deployed on the server, prior to using the system productively. After this off-line testing phase, JAGR has collected all the information it needs to make the proper recovery choices when fault events are detected.

The detection of such fault events is always done by correlating the information gathered by at least two different monitors. This is necessary, because using only the exception monitor might cause JAGR to detect a failure, when it really is normal application behavior. Due to the generality that this project wants to achieve, JAGR is only allowed to use information gathered automatically. A human administrator cannot support JAGR by providing additional information about the system's composition and configuration.

When the recovery manager determines that a fault has occurred, it will decide which components are to be rebooted and sends this information to the recovery agent. This agent is then responsible to issue the proper commands to the underlying JBoss instance. At the same time, the recovery manager will activate the stall proxy, so that any client requests that arrive during the (micro-) reboots will be delayed up to 8 seconds, which in most cases is enough time for the reboot process to have been completed.

This *self-healing* capability introduced by JAGR is, however, limited to three-tier web applications. More specifically, it is only applicable to applications that store their persistent data in a database outside the scope of the application server and that only have clients, which access the application via a web browser interface. The reason for this constraint lies in the loose coupling between the three tiers (web browser sessions are regarded stateless), which permits the (micro-)reboots without breaking already existing client connections.

But it is still the only implementation of a *self-healing* capability that is close to use in production environments. Trying to implement this same functionality for core services of a J2EE server is only achievable with expert knowledge of the system itself [LHF05] and not yet apt for widespread use.

3.1.3.4 "Self-Optimization" for JBoss

Application servers are highly complex software products. They have many parameters that can modify system behavior at various levels. This makes tuning of these systems without detailed expert knowledge very difficult.

There are best-practice guides that can be of assistance to administrators, but there are still cases in which the best-practice suggestion is in conflict with the best possible configuration achievable [RRJ03]. What is more, static configurations are not very well suited for the dynamic demands of today's e-businesses.

An empirical approach to maintain quality of service even under constantly changing conditions has been presented in [FSE04]. In this work, an "Automated Tuning System" has been developed. This system allows the selection of the current application server configuration from a set of optimum configurations. The optimum configurations, however, have to be determined in an off-line testing phase with simulated loads on the server.

Such offline testing imposes the drawback that an optimal configuration can only be found for request patterns that resemble the load conditions during the preceding test period. The prototype presented in [FSE04] enriches JBoss with a monitor and a controller service implemented as managed beans, which operate on the previously collected data. A server-side interceptor constantly updates an "Actual QoS" object, from which a response time is obtained. The monitor component then uses this information to guide the controller in the tuning of server configuration. The controller accesses other MBeans — which are not further detailed — to make the configuration changes. Results are presented as response time data which is recorded during runs of the ECperf benchmark (cf. section 4.4.1) under different load configurations.

Another possibility to implement self-optimization behavior in JBoss has been presented as the "Adaptive Self-Configuration Architecture" or "Adaptive Performance Configuration Architecture" [ZQL06], which is shown in Figure 3. The Data Collector obtains real-time performance values, which are further processed by the Data Processor to obtain average values. A Predictor assists the Decision maker to proactively deal with dynamic workloads. The Comparator detects deviations from the Service Level Agreement and feeds this information to the Decision maker. This decision maker applies algorithms from its Knowledge Base, based on input from Comparator, Predictor and the performance Data Base to create instructions for the Tuner, which will finally modify the system's behavior. In this approach, modifications to the JBoss source code were necessary,

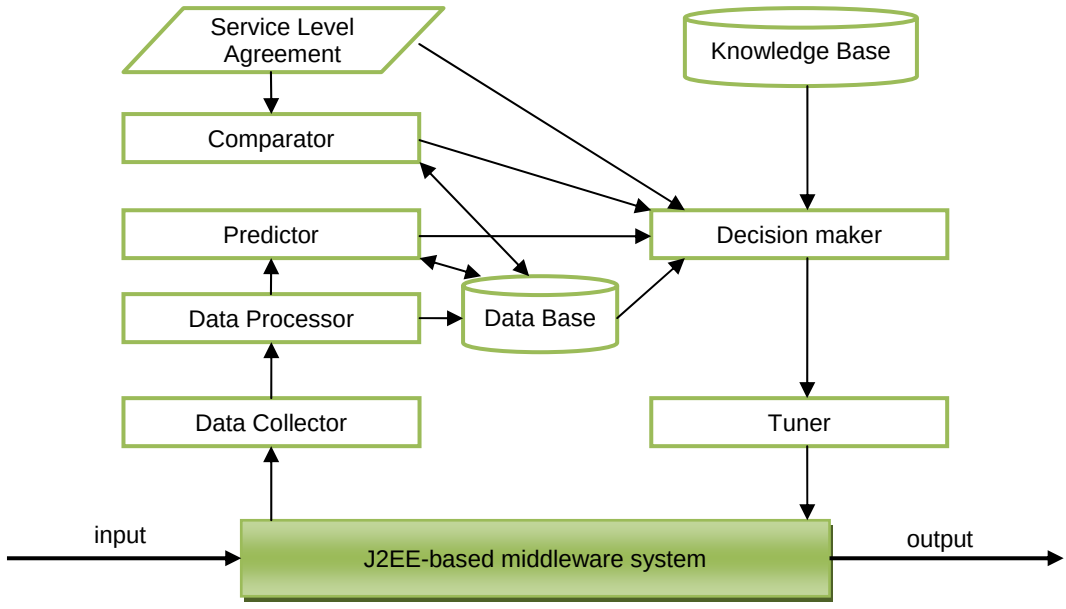

Figure 3: Adaptive performance configuration architecture (from [ZQL06])

because otherwise the performance configuration changes that had been planned would not have been possible without a shutdown and restart of the JBoss application server.

The adaptive performance configuration algorithm in this case uses fuzzy control. This means that the rules which govern the optimization efforts are expressed in fuzzy linguistic terms (e.g. “change in MaxPoolSize is large”). Input parameters have to be normalized and transformed into fuzzy sets to be processed by the fuzzy ruling engine. The same applies to output variables which have to be de-fuzzifized and mapped to real-world values.

This model-free approach showed acceptable results in performance tuning and research is still ongoing to reduce the time the algorithm needs to converge to the optimum values. The results presented have in this case been obtained using a benchmark which simulates on-line stock-broking. Communication with the clients is completely done through a browser-based interface. So in contrast to the ECperf benchmark, there are no Java RMI connections which would influence the results obtained.

3.2 Management Standards

Autonomic computing is, in essence, concerned with introducing *self-management* features into software — and in the future also hardware — resources. Any kind of management has to have access to information about the system it is supposed to manage.

Today, there are various standards for the different levels and aspects concerned with management information. At the foundation of management is the way in which management information is stored and how it is gathered. The simple network management protocol (SNMP), for example, defines its data model to be the “Management Information Base” and is discussed in section 3.2.1.

Another aspect which plays a central role in IT system management is protocols and data formats used in accessing management information over a network. Current management implementations base upon internet protocols, such as the User Datagram Protocol (UDP) or the Transfer Control Protocol (TCP) to send either proprietary messages or exchange information in a standardized format, such as the Extensible Markup Language (XML) [TM06].

Other standards, such as the Java Management Extensions (JMX) additionally define an API for the instrumentation of software applications [Kum05]. The need for IT Management Standards has clearly be expressed [ITS06] and very often different standards from different vendors have to be combined to be able to get hold of the complexity of current IT system deployments.

The latest development in IT management is to develop management applications using the Service Oriented Architecture (SOA) [Kum05]. SOA is described as “… an architectural style whose goal is to achieve loose coupling among interacting software agents. A service is a unit of work done by a service provider to achieve desired end results for a service consumer.” [HeH03]. Loose coupling in this case means, that the different software applications should be interchangeable and inter-workable. This allows application programs written in any programming language and running on any platform to access services offered by applications, which adhere to SOA.

Currently, SOA implementations are often based on Web Services (WS), because their properties are largely congruent with the requirements of SOA. Use of WS for the exchange of management information is made by the Web Services Distributed Management (WSDM) group of standards. These only define the representation of management information as it is transmitted and to be seen on the "wire".

The following sections give insight on currently used management standards, with SNMP as the ancestor of distributed management. JMX and WSDM are subsequently covered, not only because they are currently being adopted, but also because of their relevance to the case study developed in the course of this thesis.

3.2.1 Simple Network Management Protocol

The simple network management protocol [RFC1157] was defined by the Internet Engineering Task Force (IETF) in 1988. It was originally thought to be used mainly for networking hardware, such as routers, switches and gateways. Since then, there have been two major new versions:

- SNMPv2 [RFC1441] included a highly-controversial security system, which caused the creation of different flavors of this standard. SNMPv2c [RFC1901] is the one that was mostly adopted.
- SNMPv3 [RFC3411] finally added important security features to the SNMP specification in 2002, which are
    - Message integrity,
    - Authentication and
    - Encryption.

Management information is stored in the so-called "Management Information Base" (MIB). This database can be viewed as a tree. The information is stored in the tree's leaves and is accessible using an object identifier. As the identifiers are also hierarchical, SNMP partitions the namespace of the information in its MIBs. This allows certain namespaces to be assigned to different organizations, which can then specify the contents of the sub-tree in their namespace. The main advantage of this approach is to allow SNMP to be extended by different parties, thereby being applicable to a wide range of tasks. An example of an SNMP-standardized object identifier is "1.3.6.1.2.1.1.1" with a textual representation of "iso.org.dod.internet.mgmt.mib.system.sysDescr". This however, is not the leave, but the object type identifier, the actual value is addressed by appending a ".0" to the object identifier, which points to the first (and in this case only) instance of the object.

The information contained in the MIBs and addressable by the object identifier can be accessed via a network using SNMP's own protocol, which is based on UDP. There are — since version 1 — five core messages that SNMP supports:

- GET REQUEST: retrieves the addressed management information from the MIB
- GETNEXT REQUEST: can be used to iteratively request information sequences
- GET RESPONSE: is sent in response to either GET or SET requests

- SET REQUEST: initializes or changes the value of a management item stored in the MIB
- TRAP: is used for the asynchronous notification mechanism provided by SNMP

An interface such as SNMP is quite useful in health monitoring and simple control tasks, but it is not suitable for more complex tasks, such as configuration and controlling of advanced software systems. This is mainly caused by the limited number of data types supported by SNMP and the low-level programming required to implement SNMP-based applications.

Because SNMP has its roots in managing network devices it cannot handle management of business applications very well [ITS06]. Additionally, because it took so long for SNMP to finally incorporate adequate security features, it has been regarded inappropriate for the management of sensitive software systems.

3.2.2 Java Management Extensions

The Java Management extensions (JMX) published by Sun defines an architecture, design patterns, application program interfaces and services for application and network management and monitoring in the Java programming language [JMX14]. The architecture comprises three levels, as shown in Figure 4:

- Instrumentation level
- Agent level
- Distributed Services level

Manageable resources are found at the instrumentation level. These can be devices, service implementations or applications. The instrumentation is then provided by one or more *Managed Beans* (MBeans), which can be standard MBeans or dynamic MBeans. An MBean is a Java object that implements a specific interface and respects the JMX design patterns. Every manageable object also has an associated interface, which contains all the attributes and operations that are used in controlling the resource. Attributes are encapsulated through corresponding public getter and setter methods. By not making the setter method public or not implementing it, the attribute will be read-only.

While standard MBeans provide information about all their attributes and operations using a static Java interface, dynamic MBeans provide a method to dynamically request metadata about their interface. This means that for dynamic MBeans the interface can by created and even modified at runtime of the manageable resource. A specialization of dynamic MBeans, called open MBeans, restricts the use of types by the MBean. This way, class loading is not necessary to use the MBean, which can be quite useful in constrained environments, such as hardware devices that do not incorporate a full-blown JVM.

At the agent level there are services for handling MBeans and the managed bean server. This is a registry for objects which model manageable resources that are exposed to management applications. The MBean server, however, only exposes the interfaces of all the MBeans that have registered, but never allows direct access to the MBean objects. Management of the MBeans must be done using the API provided by the MBean server. All registered MBeans have a unique object

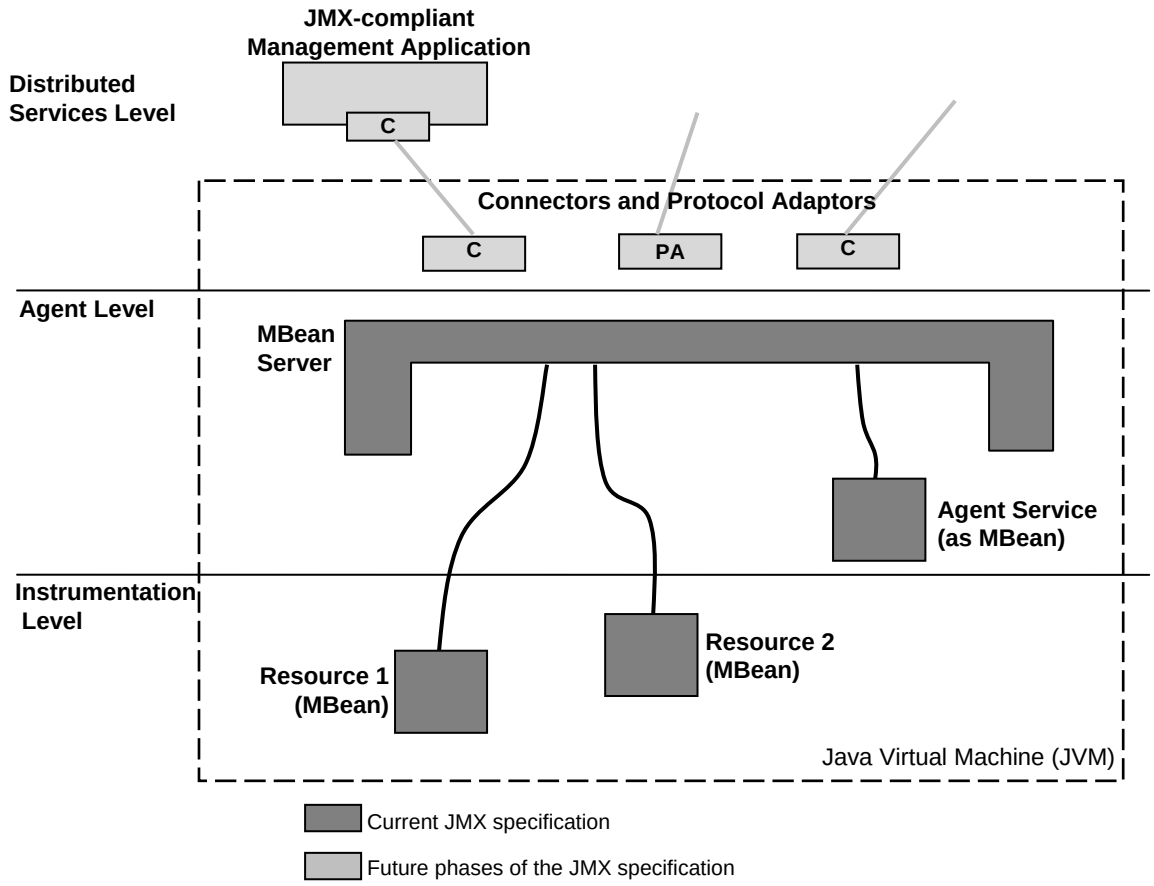

Figure 4: Components of the JMX Architecture (from [JMX14])

name by which they are identified. Management applications can then query the MBean server for manageable resource in which they interested by using parts of the object name. The MBean server also provides a notification mechanism. Management applications can thereby subscribe to notification messages emitted by the MBeans managed by the MBean server.

The agent services that are required by the JMX specification include:

- a dynamic loading server, which allows the creation of MBeans from classes and libraries that can be dynamically downloaded from a network,
- a monitoring service, which will emit notifications in the event of attribute changes of MBeans that can be specified and
- a timer service, which will emit notifications at specified time intervals and can be used as a scheduler by the MBeans.

If management applications outside of the MBean server's JVM want to communicate with the resources, they have to use the connectors and protocol adaptors that are specified on the distributed services level. This is covered by the "JMX Remote API Specification" which is part of the Management Extensions Specification [JMX14].

The only connector defined in the current version of the JMX specification is a communication protocol based on the Java Remote Method Invocation (RMI) API. This, of course, limits the usability of JMX to management applications written in the Java programming language. Wrappers for existing management standards, such as SNMP are suggested by the specification, but there is no implementation for these at the time of this writing.

Currently, JMX is mostly used as management architecture to be used only within the same virtual machine as the manageable resources reside in. Many application servers now use JMX to build upon. Examples include IBM's WebSphere, BEA's WebLogic, Sun's Java System Application Server and JBoss.

Since version 1.5 of the Java 2 Platform (Standard Edition), JMX is part of the core platform API. The Java Virtual Machine developed by Sun also uses JMX to make management information about the JVM itself available.

3.2.3 Web Services Distributed Management

As a management standard that avoids the problem of lacking interoperability, Web Services Distributed Management (WSDM) is an open standard devised by OASIS — the Organization for the Advancement of Structured Information Standards [MUWS11a, MUWS11b]. It builds upon other WS-related standards from OASIS to define the interactions between a manageable resource and its manager as well as the representation of and access to the resources properties as encoded in WS messages.

The definition which is at the core of the WSDM standards is the definition of "*resource*" and "WS-resource", both of which are contained in the WS-Resource standard that belongs to the specifications of the Web Service Resource Framework (WSRF) [WSR12, WSRP12]. A resource is defined as an identifiable, logical entity that has a set of zero or more properties which can be expressed in XML. Additionally, a resource may have a lifecycle. The term "*WS-resource*" describes the combination of a resource with a web service through which the resource can be accessed. As such, the WS-resource is represented by an endpoint reference (EPR), which can be used to access the resource.

The messages exchanged between the WS-resource and its clients are defined in the WS-Resource Properties (WSRP) specification. This specification also introduces:

- the resource property, which is a piece of information defined as part of the state model of a WS-resource,
- the resource property element, which is the XML representation of a resource property and needs to have a unique qualifying name in the XML namespace and
- the resource property document, which is the combination of all the resource's property elements in an XML document together with their resource property values. This document is a snapshot view of the resource's current state.

The types of all the elements that are represented in XML have to be defined in a corresponding XML schema document, which will in most cases be integrated into the WSDL document that describes the web service of the resource.

The operations that a client can use to access the information stored in the resource property document are listed as follows.

- `GetResourcePropertyDocument` returns the complete XML document in the body of the response message.
- `GetResourceProperty` is used to request the value of exactly one resource property
- `GelMultipleResourceProperties` retrieves the values of all the resource properties specified to this operation as parameters.[3]
- `QueryResourceProperties` can be used to search the resource property document for specific values using some query expression dialect (such as the XPath language [Cla99]).
- `PutResourcePropertyDocument` permits a client to replace the resource's current resource property document with an entirely new version.
- `SetResourceProperties` allows the client to make multiple updates to the resource property document in a single request message, which can be of type insert, update or delete.
- `InsertResourceProperties` supports augmenting the resource property document by one or more instances (element values) of a single resource property.
- `UpdateResourceProperties` is used to change the value(s) of a resource property.
- `DeleteResourceProperties` can allow the removal of all values attached to a certain resource property.

Using the above list of operations allows a client to the WS to monitor and change the state variables of a resource. For management-related tasks, these are the groundwork, upon which two WSDM standards are built:

- Management Using Web Services (MUWS)
- Management Of Web Services (MOWS)

For the purpose of this thesis, only the WSDM MUWS standard is of further interest; it defines how an IT resource connected to a network provides manageability interfaces such that it can be managed locally and from remote locations using WS technology. The manageable resource exchanges messages with a manageability consumer, which allows the consumer to make use of the resource's manageability capabilities. These capabilities are the central point of the WSDM MUWS standard.

Additional items covered by MUWS include the definition of an event format for management-related events and the adoption of the web services platform. MUWS itself does not define concepts for metadata access, addressing and security functionality, but rather mentions, that due to the

[3] This operation is the response to criticism that has been exerted on SNMP's value-retrieving operations, such that it can be used to request an arbitrary combination of values from the resource.

adoption of the WS platform, other implementations of these features are applicable to WSDM MUWS as well.

The standard is composed of two parts, the first one of which provides the fundamental concepts and the second one provides specific information used to enable interoperability of MUWS implementations. The manageability capabilities defined in these two parts are presented in Table 1. Other, management-related capabilities are concerned with relationships among resources.

Because WSDM is an open standard and builds upon web services, it can also be embraced for the implementation of manageability interfaces in autonomic computing systems. Furthermore, the only required capability that a resource has to implement is the Identity capability. Other capabilities, resource properties and operations can be added at will without interfering with the specification. As a result, conforming implementations are possible, ranging from a minimalistic manageability endpoint embedded in some hardware device to a highly-complex and multi-featured WS resource, as for example an enterprise information system.

Table 1: WSDM MUWS Manageability Capabilities

Capability	Description
Identity	Used to establish whether two entities are the same. This capability defines the `ResourceId` property.
Manageability Characteristics	Provides information about the characteristics of the manageability endpoint implementation.
Correlatable Properties	Lists property values that can be compared in order to establish a relationship between different resources of the same type, including identity.
Description	The description capability gives the manageable resource additional metadata, which may include descriptive names and explanations intended for human administrators.
State	If the resource exhibits behavior according to one or more state models, this capability allows the resource to give additional information about its internal state or states.
Operational Status	Represents the availability of a resource, where allowed values are: • Available • Partially Available • Unavailable • Unknown
Metrics	The metric capability applies to a specific type of resource properties, which represent values collected during a period of time. Such properties are then attributed with metadata describing the circumstances under which the values are obtained.
Configuration	Properties tagged with the configuration capability have direct influence on the operational behavior of the resource.

Scalability and availability of management systems based on WSDM have been argued to be insufficient for large-scale distributed systems [ABI05]. It has to been noted, however, that any system beyond a certain scale will always need performance improvements that are only possible by modifications of the standards that are designed to suit a broad spectrum of demands.

Backed by major players in the industry, WSDM is currently moving toward merging the MUWS and MOWS standards into a unified approach. This approach will define a common management profile that provides Web-service-based access to autonomic elements [TM06]. It is expected that this will finally install WSDM as a management message representation standard for autonomic computing systems in general.

4 Case Study

In the course of this diploma thesis a prototype of a remote autonomic manager (AM) has been designed and implemented to show how an existing system can be enriched by autonomic computing (AC) capabilities. This required not only development efforts for the manager, but also on the side of the system to be managed. This system needs to meet certain criteria:

- It has to be freely available, preferably under an open-source license as this would also allow modifying the source code if necessary.
- The system needs to be extensible by design. Adding a management interface for use with the autonomic manager to be developed should be achievable within the time given.
- Parameters that influence system behavior must be changeable at run-time to be able to show the effects of the autonomic manager's activity without having to restart the system.

All these needs were found to be met by the JBoss application server [FR03]. The next step towards a prototype that exhibits AC capabilities was to define scenarios in which AC features would be observable.

As an application server, JBoss is a complex software system which has many parameters that have to be configured. Because such a kind of configuration is tedious and prone to errors, it could be facilitated by a *self-configuration* capability. Anticipating that the development of an autonomic manager that would respect all such configuration parameters is far beyond the time-horizon possible for this work, the configuration of some basic parameters would be accepted as sufficient. The AM would additionally check the system's resource usage and prevent system failures by adapting the configuration settings.

The *self-optimizing* capability is also of interest for a system like JBoss. An optimization in this case would consist in reducing the response times for client requests sent to the application server. This requires the response times to be measurable and system parameters to be accessible and changeable.

Being an application server, JBoss can host multiple different applications. The autonomic manager should be able to notice when an application has failed and revert it to a functional state. This could be achieved by re-initialization or re-deployment of the application. Such a behavior can then be seen as the result of the overall system having a *self-healing* capability.

The last of the four basic AC capabilities (see section 2.1.1), *self-protecting*, is not addressed in this case study. A summary of the self-management features that are implemented in this case study is given in Table 2.

Table 2: self-management features dealt with in the case study

self-configuration	• check the correctness and usefulness of selected parameters • monitor resource usage and reconfigure if necessary
self-optimization	• reduce response times for client requests
self-healing	• revert failed applications to functional state

The communication between the autonomic manager and the managed system will be based on the WSDM MUWS standard (see section 3.2.3). Because WSDM uses HTTP for transport, firewalls and gateways that may have to be traversed by management messages can easily be configured to allow such messages. And since the management information is encoded in XML, which basically means that it is represented by structured text, any programming language on any platform can be used to interface with the developed management Web service.

4.1 JBoss — the Prototype Environment

The JBoss Application Server is a production-ready J2EE application server. Its core design makes it highly extensible and dynamically reconfigurable. Because it is distributed under an open-source license, its source code is freely available. This makes the product attractive for businesses. But JBoss is also very attractive for research, because it is at state of the art with research areas such as component-based software development, reflective middleware and aspect-oriented programming [FR03].

JBoss achieves high modularity by building upon the Java Management Extensions (JMX) specification, which is explained in section 3.2.2. All the components that make up the server's functionality — as for example transaction support, persistence, messaging service or naming service — are developed as Managed Beans (MBeans). The core of JBoss — or its "spine" — is an MBean server that all the components register with. It is that MBean server which provides the different components access to each other by accessing their attributes, invoking their methods or distributing their notification messages to all registered listeners. The MBean server can, therefore, also be regarded as "JMX communication bus". JBoss' extension mechanism, which highly depends on JMX's characteristics, is explained in section 4.1.1.

Several custom extensions were developed for this diploma thesis. They all plug into the central JMX-bus as can be seen in Figure 5. In this figure, the custom extensions are highlighted by a broken line in the lower left. From left to right these are:

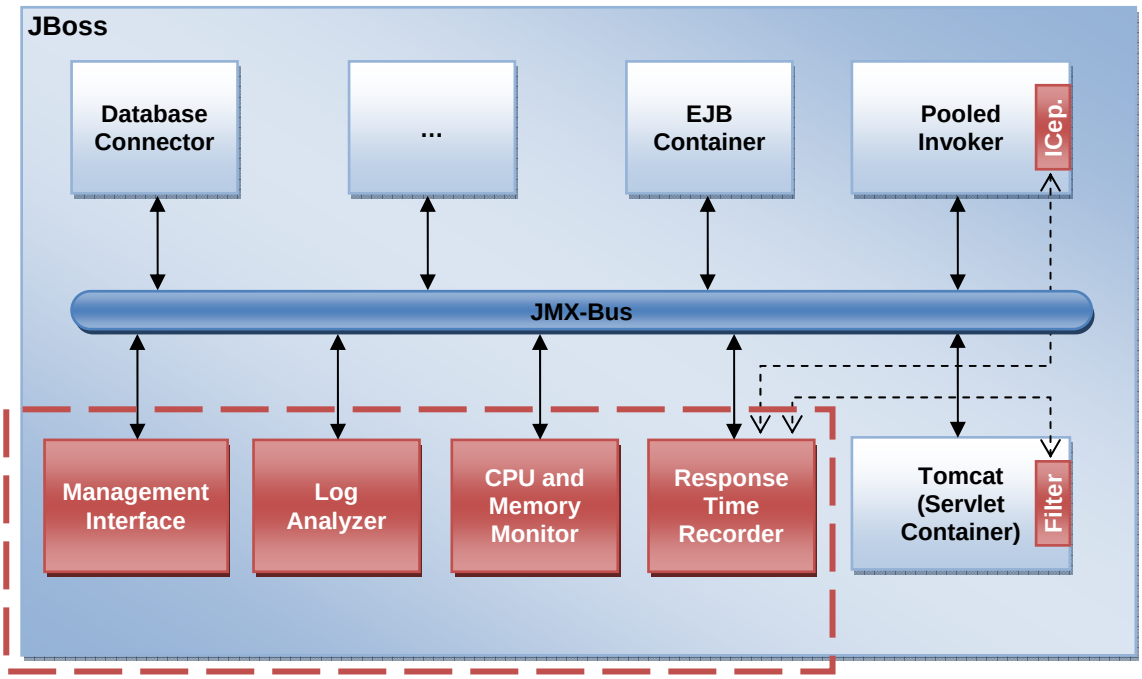

Figure 5: Extensions to JBoss

- the Management Interface which provides access to management information and operations through a web service,
- the Log Analyzer, which continuously monitors the log for any occurrence of application failure messages (i.e, exceptions),
- the CPU and Memory Monitor which collects data from the underlying Java Virtual Machine (JVM) and
- the Response Time Recorder. This extension is concerned with gathering information about the response times for client requests. It hence also has connection with an interceptor and a filter module as indicated by the small double-arrowed lines. These modules reside within the Pooled Invoker and the embedded Tomcat, respectively, and are discussed below along with the Response Time Recorder.

All the developed components are presented subsequently after the JBoss extension mechanism.

4.1.1 JBoss' Extension Mechanism

It is not too difficult to extend the capabilities of JBoss with custom services and functionality. Extension modules just have to be developed in accordance with JBoss' MBean service architecture. A custom service is created by developing a Java class that implements an interface with the class' name appended by "`MBean`". A custom service class named "`MyService`" should, e.g., implement an interface by the name of "`MyServiceMBean`". All the operations that are to be made accessible via the JMX bus have to be declared in the interface and all attributes to be exposed must have their corresponding getter- and setter-operations depending on whether they are to be readable, writable or both. The Java classes and interfaces that make up the custom service then only need to be packaged into a Java Archive file with the file extension ".sar". Such an

archive file must also contain a so called deployment descriptor, which is an XML file with the name “jboss-service.xml”, that needs to be put in the “META-INF” directory of the archive file.

The only thing left to deploy the created service is to copy the “.sar”-file to one of JBoss’ hot-deployment directories. These directories are continuously monitored for changes and JBoss will immediately react to changes to these directories. In the case of a newly added service archive (SAR) file, the application server will load the contained classes, parse the deployment descriptor, inspect the interface to be exposed using the Java reflection mechanism and then create and instantiate the service, and finally register it with the MBean server.

Additionally, JBoss allows custom services to expose special life-cycle operations. Their purpose is to make it possible for custom services to properly allocate and free external resources. These operations are defined to have the following method signatures exposed in the service’s MBean interface definition:

- `public void create()`
- `public void start()`
- `public void stop()`
- `public void destroy()`

At deployment, every service class that exposes any of these operations will have the `create()` method called after it has been constructed. Additionally, it is guaranteed to have its `start()` operation executed before the service is registered with the MBean server and thereby made publicly accessible. At shutdown of the service, the `stop()` operation will first be called and eventually the `destroy()` method will be invoked before the object that implements the service will be marked for garbage collection.

Should a specific service depend on other services in order to function correctly, these dependencies can be expressed in the deployment descriptor. This configuration file also is the location where the name, under which a service is to be registered, has to be specified. An example for such a deployment descriptor is given in Figure 6.

```
<server>
   <mbean code="at.schwarz.jboss.ManagementWS"
          name="at.schwarz:service=ManagementInterface,type=WebService">
      <depends>at.schwarz:service=ResponseTimeRecorder</depends>
      <depends>at.schwarz:service=Monitoring,type=MemoryAndCpu</depends>
      <depends>at.schwarz:service=Monitoring,type=Exceptions</depends>
      <depends>at.schwarz:service=Utility,type=ClearCaches</depends>
      <depends>jboss.system:service=MainDeployer</depends>
      <depends>jboss:service=invoker,type=pooled</depends>
      <depends>jboss.jca:service=ManagedConnectionPool,name=TestmeDB</depends>
      <!-- Startup Attributes -->
      <attribute name="PortNumber">8888</attribute>
      <attribute name="ContextPath">/management/services/</attribute>
   </mbean>
</server>
```

Figure 6: The deployment descriptor for the WS management interface service

4.1.2 CPU and Memory Utilization Monitoring

JBoss is written in the Java programming language and as such runs on a Java Virtual Machine (JVM). For any application that runs on a JVM, the JVM manages access to the two system resources: memory usage and processor time. Since version 1.5, the JVM itself exposes its management API via an MBean server: the platform MBean server. This gives the JVM the advantage of having different kinds of management information available for the different operating systems that a JVM can run on. For the Microsoft Windows operating systems used in this work (XP and Vista) the JVM provides the value of "ProcessCpuTime". This value indicates, how many milliseconds the JVM process has been using the CPU so far. By repeatedly polling this value, an indicator of current CPU load caused by the JVM is established. This indicator is made accessible as an attribute to the "CPU and Memory Monitor"-MBean which implements this service.

Additionally, the JVM exposes an MBean that provides detailed information about the threads currently controlled by the JVM. This information includes not only thread ID and name, but also the state of each thread, its current stack trace and — if supported by the JVM — the total CPU time for each thread in nanoseconds[4]. The "CPU time per thread" is used to detail the total CPU time usage of the JVM process down to the individual threads. Again, constant polling of these values permits obtaining indicators for the current CPU load caused by the individual threads. Detailed information about all the currently running threads can be obtained by a management client through the developed "`extractInfoForAllThreads()`" method, which interfaces with the JVM to collect all the relevant data.

The memory used by the JVM can be divided into two main categories: heap memory and non-heap memory. Heap memory holds all the data created by the applications that run within the JVM. For the purpose of memory usage monitoring, the values of used and free heap memory have to be taken into account. Because Java objects that are no longer used in memory are garbage collected by the JVM at a random instant, the amount of used memory, as reported by the JVM[5], is fluctuating constantly. Any decision that is targeted for longer timescales should consequently not rely on a single sample of the "free memory" value.

Other values of interest regarding the JVM's heap memory usage are the amount of memory that is currently allocated and the maximum amount of memory that the JVM is permitted to allocate for the heap. It has to be noted, however, that the maximum allowed value is not available in all JVM implementations or on all platforms. When it is available, it may be constrained by configuration parameters passed to the JVM at startup or by the specific JVM implementation.

All the data mentioned above are gathered and assembled by the "Memory and CPU Monitor"-MBean service that has been developed for this thesis. Its complete interface is shown in Figure 7.

[4] The value obtained is of nanosecond precision but not necessarily of nanosecond accuracy. This depends on the specific implementations of the underlying operating system and JVM.

[5] The JVM does not directly report the amount of free memory. This value can be obtained by subtracting the amount of memory used from the amount of memory available.

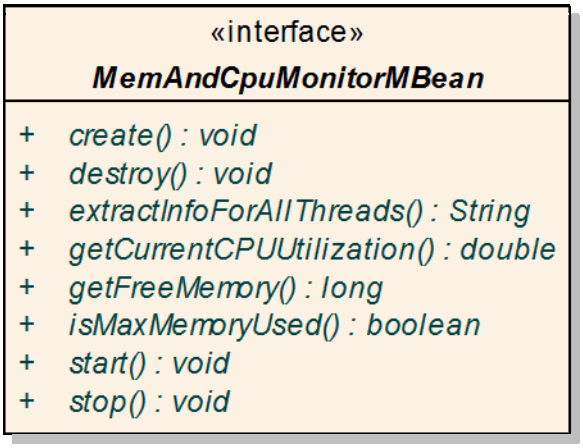

Figure 7: Interface of "MemAndCpuMontiorMBean"

Since the detailed information about all the currently running threads in the JVM will not always be needed, it has been decided that it would be too much of an overhead if these values would be constantly polled and calculated. They are available through an MBean operation that will sample all thread-data twice with a pause of approximately 1,000 milliseconds in between the samples. The difference of the two values obtained is then divided by the difference in the timestamps for the two values to assemble a processor time usage percentage for each thread. The percentage values obtained are only to be used to establish a ranking among the JVM's threads. This is due to the fact that the accuracy of the values, on which the calculation is based, is unknown, because they are highly dependent on the implementation of the JVM and the underlying operating system.

On startup the "MemAndCpuMonitor" MBean creates a thread that samples the "ProcessCPUTime" once every 1,000 ms and updates the MBean attribute "CurrentCPUUtilization" accordingly. The "FreeMemory" attribute is updated every time it is requested, as is the other memory-related attribute: "MaxMemoryUsed". This is a Boolean value that indicates whether the JVM has allocated the whole amount of heap memory it is allowed to. The additional load introduced on the system by the periodically executed update thread is negligible.

4.1.3 Recording Response Times

Measurement of response times in this case study basically involves three separately developed software units, which are all shown in Figure 5 on page 30:

- The *Response Time Recorder* through which average response times can be obtained by a management client,
- a custom "*Filter*" that measures response times for HTTP requests and
- a custom "*Interceptor*" to measure response times for requests to Enterprise Java Beans deployed in JBoss' EJB container.

The latter two are inserted into the request flow as visualized in Figure 8: Every time a new client request arrives at an instance of one of these modules, the current system time is sampled. The client request is then handed on to the next interceptor on the stack. When the processed response

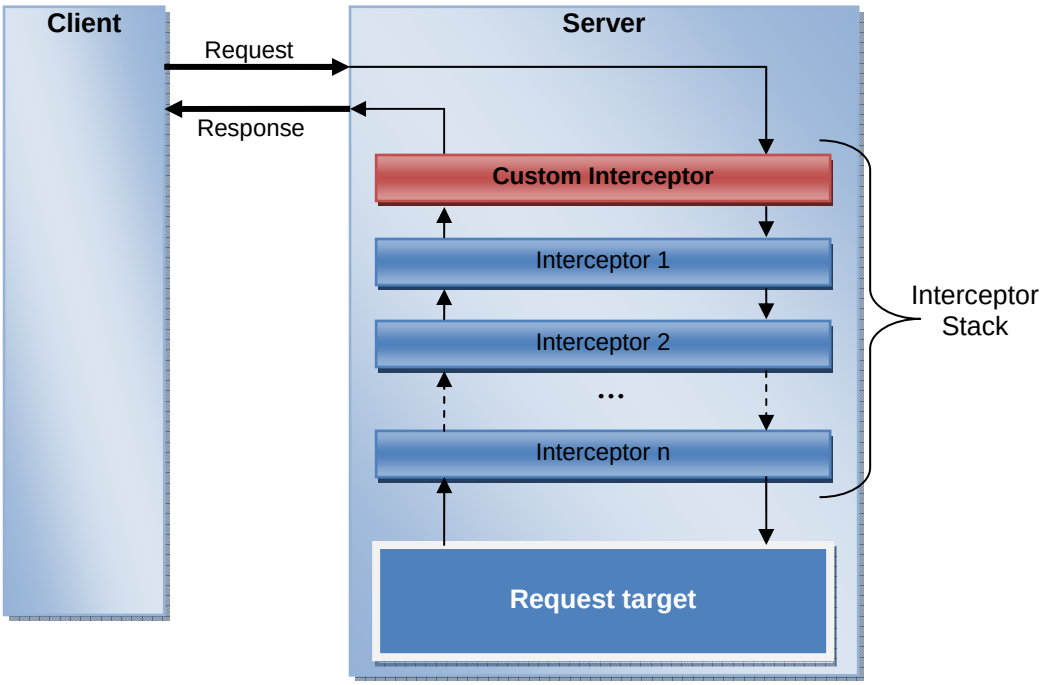

Figure 8: The Interception Mechanism

gets back to the interceptor, the system's current time is sampled again, and the difference between the two samples will give the server-side response time for that specific request.

4.1.3.1 The Response Time Filter for Tomcat

The "`Filter`"-interface is defined by the J2EE standard.[6] Its purpose is to allow the modification of HTTP requests before they are forwarded to the target servlet and the modification of HTTP responses before they are sent back to the client. Various filters can be combined to form a so-called "filter chain". The concept of filters and filter chains is equivalent to that of interceptors and the interceptor stack as shown in Figure 8. The Response Time filter implemented for this case study does not modify the request or the response, it merely measures the request-processing time and extracts the target URI from the request's metadata. For every unique target URI, the Response Time filter holds the total average of all response times measured.

In order for this information to be accessible for a management application, the Response Time Filter has to register with the Response Time Recorder. This is done in the initialization phase that is defined by the "`Filter`"-interface. Being registered, the Response Time filter can be polled by the Response Time Recorder to return all collected response time information compiled into a String.

The filter is configured to be used for all the HTTP requests that arrive at JBoss' embedded Apache Tomcat. This is done by specifying the filter's Java class name in the global web application configuration file "conf/web.xml", which can be found in the "jbossweb-tomcat55.sar"-directory.

[6] The full qualified Java classname is `javax.servlet.Filter`.

4.1.3.2 The custom Interceptor for EJB calls

JBoss processes all requests to the EJBs deployed in its EJB container by routing them through an interceptor stack. All elements on that stack have to implement the "`org.jboss.ejb.Interceptor`" interface. The last interceptor in the stack ("Interceptor n" in Figure 8) then looks up an instance of the targeted EJB object, and invokes the corresponding method on it. Configuration of that interceptor stack is performed in JBoss' global EJB configuration file "standardjboss.xml".

The `ResponseTimeInterceptor` that has been developed is configured to intercept all calls to Stateless Session Beans. JBoss will create one instance of the interceptor for every EJB deployed on the server. All of these instances register themselves with the Response Time Recorder in their initialization step. The interceptors will then measure the response time for each call and extract the name of the method invoked on the EJB. Furthermore, at the first invocation that is made through the response time interceptor, the target EJB's name is determined and stored locally for every interceptor instance.

For every method that has been called, the interceptor holds a total average of response times measured. This information can then be polled by the Response Time Recorder and is returned compiled into a String.

4.1.3.3 The Response Time Recorder

All the response time information collected by the filters and interceptors is accessible through the response time recorder (RTR). This unit — deployed on JBoss as a custom service — maintains a list of all the registered response time information providers. The gathered information can be requested through the RTR by invoking the "`pollInfoEJB`" or "`pollInfoHTTP`" operation exposed on the MBean server. Upon calling one of those methods the RTR invokes the "`pollInfo`" method on all the filters or interceptors iteratively and combines the data received.

As already mentioned in the previous two sections, the return value of this method is of type String. This requires the data to be parsed into an appropriate data structure before it can be further processed. Normally, one would define a structured data type to hold the information and use that data type as return type. But in this specific case a basic Java data type had to be used, because of the following reason: In the Java programming language, every type — which means every class or interface — is identified by its fully qualified class name and its defining class loader [LB98]. In most Java applications the developer doesn't even have to be aware of the existence of class loaders. JBoss however uses a sophisticated class loading mechanism to prevent namespace clashes if two different applications should use the same qualified class name [FSR06]. The three separate units that make up the response time functionality in this use case are located within different naming and class loading contexts (cf. Figure 5):

- The Response Time Recorder has a class loader of its own, since it is deployed as an extensions service of its own right.
- The Response Time Interceptors inherit their class loading context from the Invoker service within which they operate.

- The Response Time Filter is loaded by the class loader JBoss designated for the embedded Apache Tomcat.

If these three modules were indeed to exchange information by a complex return type, they would each need to load that class' definition by their assigned class loader. In essence, this means that there would be three different types, which — in terms of Java — are incompatible. The returning of a value from one of the three to another would therefore fail with a `ClassCastException`.

All the basic Java types (and all types introduced by the basic JBoss libraries) are treated specially by JBoss' class loaders to be equal and compatible across all components that run within the JBoss application server. It is hence possible to use any of those types for the return value.

By the way, the same problem as with the return values also occurs for method calls on classes that reside in a different class loader context. The invocations of the "`pollInfo`" method as well as the registering and unregistering operations are consequently done using the reflective method invocation paradigm provided by the Java language. The overhead caused by the reflective invocation of the "register" and "unregister" operations is negligible, since these operations are only called once each in the lifecycle of the response time filter or interceptor. Noticeable penalty due to the `ClassCastException` avoiding approach is only to be expected in the invocation of the "`pollInfo`" method of the response time recording components. But this overhead could be reduced with respect to the current implementation by caching the Method object associated with "pollInfo" for each of the polled components.

4.1.4 Monitoring the Log

JBoss uses Apache log4j as its internal logging system. Following JBoss' service architecture it is represented by a management bean registered at JBoss' MBean server for the purpose of basic configuration. Log4j offers various options for how logging messages will be stored, displayed or sent to other message consumers. One of the options is to register another MBean with a configurable ObjectName that emits JMX notifications for every log message processed by log4j. This option has been enabled with its configuration set to the default values for JBoss.

An MBean service — named LogAnalyzer — has been created that subscribes to JMX notifications issued by Log4j. The subscription and message forwarding for notifications is handled by the JBoss MBean server. Every time a logging message is received, its textual representation is parsed to test whether the message was caused by an exception.

Once it has been established that an exception has occurred the LogAnalyzer will itself issue a JMX notification containing the exception's stack trace, as seen in the log. The web service management interface component described below will subscribe to these notifications and pass them on to all the web service endpoints that have claimed their interest in these.

4.1.5 The Web Service Management Interface

All the components to provide supplemental functionality on the JBoss platform described so far use the JMX bus to exchange their management related messages. Since it is the aim of this thesis to

create a scenario in which JBoss can be managed remotely via a web service interface, a component which provides this interface has also been created as a service on JBoss' MBean-based architecture and can be seen in Figure 9.

The web service endpoint that constitutes a management interface for JBoss has to be decoupled from other web service endpoints, which pertain to applications deployed on JBoss. Otherwise, management operation requests would have to share the available communication paths with application requests. In this case, it might be impossible to still carry out management related tasks when there is heavy load on the server.

This is the reason why the web service infrastructure that is offered by JBoss could not be used to implement the management interface. But this means that another way had to be found to handle incoming management client connections and process the requests issued therein. As the transport mechanism for the management messages is HTTP, an HTTP-server is required to accept client connections. A fully-featured HTTP-server would, however, impose too much overhead for the purpose at hand. Since it is known that all expected incoming requests will be of a very specific type, a minimalistic HTTP-server has been developed, which is able to process exactly the one request type that is used in the management-related communication in this use case.

For parsing and constructing the management messages from and into the WSDM MUWS XML-format, the Apache Muse framework is employed. This framework has the additional benefit that it can process standard WSDM operations. Only custom capabilities that are specific to the resource to be managed have to be implemented and plugged into the framework.

The structure of the complete management interface and its communication with the client on the one side and JBoss on the other can be seen in Figure 9. As indicated in the bottom left of that figure, all client requests to the management interface arrive as HTTP requests at the minimalistic

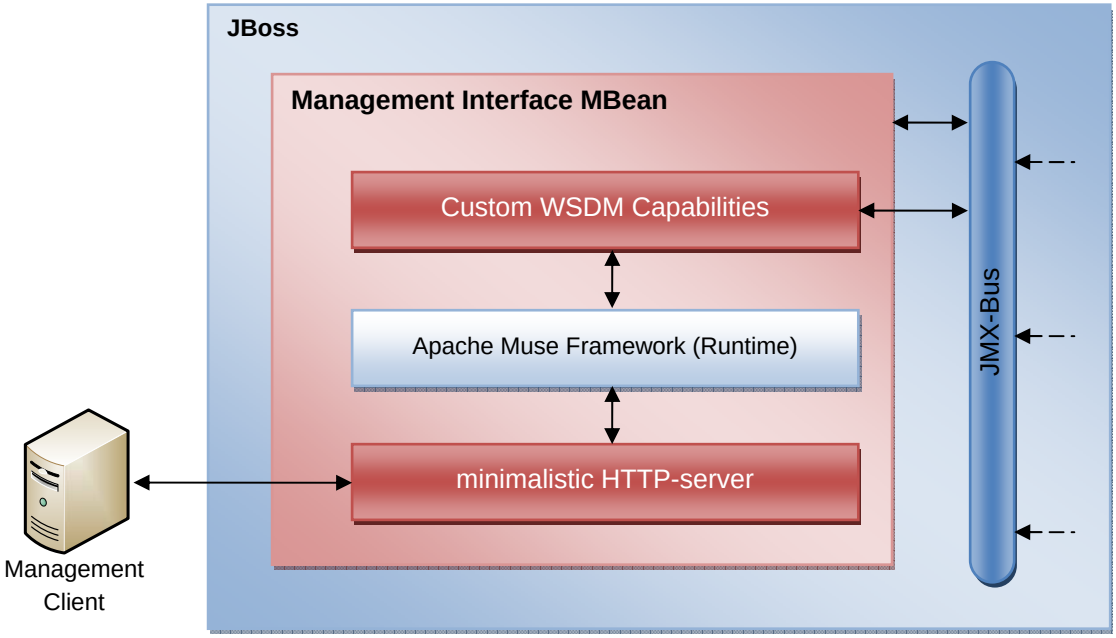

Figure 9: Structure of the Management Interface MBean

HTTP server. Upon initialization of the management interface MBean, a listener thread is installed at the TCP port configured to be used for management message exchange.[7] The minimalistic HTTP-server extracts the data of all incoming requests that are HTTP POST-requests and forward it to the Muse Framework. The return value from the call to Muse is an XML document, which is sent back to the client as contents of the HTTP response.

All other request types are answered with an HTTP error code message. The minimalistic HTTP-server is, therefore, in no way conforming to either the HTTP 1.0 or HTTP 1.1 standards [RFC2616].

The framework employed for parsing and constructing WSDM MUWS conformant messages, Apache Muse, has been created by the Apache WS project and is freely available [Muse]. It provides an implementation of the WS-ResourceFramework (WSRF), WS-BaseNotification (WSN) and WS-Distributed Management (WSDM) specifications in the Java programming language. It is distributed together with template WSDL files that show typical management interface definitions. The distribution also contains a utility that can generate skeleton code in Java based on the definitions found in a WSDL file. The WS interface for JBoss has been created with the help of the Muse framework libraries and utilities.

At first, a WSDL file that contains the definitions from the WSDM specifications that are needed for this interface was created based on the aforementioned templates. Then, the JBoss-specific properties and operations were added. The *WSDL2Java* utility was used afterward to generate the implementation skeleton for the capability class "`MyCapability`" that is configured to be used by the Muse runtime.

The capability class skeleton contains method signatures corresponding to all the operations and properties that were added to the WSDL file. All the standard operations and properties do not have to be implemented, because their implementation is already supplied by Muse. The custom operations introduced for management of JBoss are:

- `public Element getHttpResponseTimes()`
- `public Element getBeanResponseTimes()`
- `public Element listApplications()`
- `public void restartApplication(String application)`
- `public Element getThreadDetails()`
- `public void clearCaches()`

The return values of `org.w3c.dom.Element` represent portions of an XML file, according to the Java implementation of the Document Object Model (DOM). This is the required return type for any method that returns structured data over a WS call. Because all WS messages are sent in XML format, any data that they contain has also got to be XML compliant. For structured data, this means it has to be represented in XML.

[7] This configuration is done in the deployment descriptor of MBean service and can be seen in Figure 6 on page 11.

Furthermore, for each of the following properties, which have been added to the manageable resource that models JBoss' management interface, *WSDL2Java* generated getter and setter methods:

- `int FreeMemory`
- `boolean MaxMemoryUsed`
- `int HttpThreadPoolSize`
- `int ThreadPoolSize`
- `double CpuUtilization`
- `int DatabaseConnectionPoolSize`

All the getter and setter methods were implemented to perform updates on the corresponding MBean attributes that are visible to the `MyCapability` class through the JMX-Bus as seen in Figure 9. The other methods invoke their respective operations via JMX and parse the obtained result values into XML fragments. Mappings of WSDM messages to JMX are shown in Table 3.

Whenever a client request is processed by the Muse runtime, the framework automatically translates the WSDM request messages into invocations to the respective method. In addition, it checks the resource's metadata descriptor to see if the invocation is allowed. The resource metadata descriptor (RMD) made it possible to prohibit update operations (calls to the setter methods) for the properties freeMemory, maxMemoryUsed and CpuUtilization.

Table 3: WSDM to JMX mappings

WSDM property/operation	JMX mapping (ObjectName and Attribute/Operation name)
FreeMemory	`at.schwarz:service=Monitoring,type=MemoryAndCpu` `FreeMemory`
MaxMemoryUsed	`at.schwarz:service=Monitoring,type=MemoryAndCpu` `MaxMemoryUsed`
HttpThreadPoolSize	`jboss.web:address=%2F0.0.0.0,port=8080,type=Connector` `maxThreads`
ThreadPoolSize	`jboss:service=invoker,type=pooled` `MaxPoolSize`
CpuUtilization	`at.schwarz:service=Monitoring,type=MemoryAndCpu` `CurrentCPUUtilization`
DatabaseConnectionPoolSize	`jboss.jca:service=ManagedConnectionPool,name=EcperfDB` `MaxSize`
getHttpResponseTimes	`at.schwarz:service=ResponseTimeRecorder` `pollInfoHTTP`
getBeanResponseTimes	`at.schwarz:service=ResponseTimeRecorder` `pollInfoEJB`
listApplications	`jboss.system:service=MainDeployer` `listDeployedAsString`
restartApplication	`jboss.system:service=MainDeployer` `redeploy`
getThreadDetails	`at.schwarz:service=Monitoring,type=MemoryAndCpu` `extractInfoForAllThreads`
clearCaches	`at.schwarz:service=Utility,type=ClearCaches` `clearAll`

4.1.6 Modifications to the JBoss Invocation Thread Pool

The JBoss invocation thread pool is part of the Pooled Invoker (Figure 5, upper right). This thread pool saves threads that handle client requests, so that they can be reused and the overhead of creating a new thread for every incoming request avoided. This is an application of the thread pool/worker thread-pattern [Lea00] and is a performance optimization measure, because it saves the CPU time otherwise needed for creation and destruction of threads. The number of client requests that can be processed simultaneously is restricted by the number of worker threads, because they do the actual processing.

When the management interface was tested, a problem with the thread invocation pool was detected: albeit the value for the maximum thread pool size had been changed, the number of threads that were actually processing the requests did not. So the change in the MBean's attribute was not reflected by the system's behavior. But since the goal of autonomic management is to tune system behavior in order to optimize the system for a specific goal, the source of the problem had to be identified.

After analyzing the source code of the JBoss distribution used, the culprit for the undesired behavior was found: it was the combination of the "PooledInvoker" and the "ServerThread" classes. The PooledInvoker creates a master thread waiting for client requests and assigns them to one of the worker threads (of type ServerThread) from the pool. After the worker thread has finished processing the request it is returned to the pool.

More specifically, the pooled invoker first checks if there are any currently unused (but already existing) threads that it could reuse. If there are none, it subsequently considers the number of threads currently serving a request. If it is below the specified maximum size, the pooled invoker creates a new thread and assigns it to the new request. If the number of threads currently serving requests is at or above the limit, the pooled invoker thread blocks until one of the worker threads becomes available.

If all worker threads are in use, and the maximum size for the threadpool is increased, this will not result in new threads being created, as the requests that have not been able to get a worker thread immediately are already blocked waiting on a currently existing thread. Therefore the change only becomes effective, once all the requests that have been queued have been served.

The other case, which would be the reduction of the number of concurrently allowed worker threads, was presumably never thought to occur. This is deduced by the fact that once a worker thread has been created it is never destroyed until program termination. Even if the maximum allowed number of pooled threads would be exceeded by adding a thread that has completed its work to the pool, it is still added, because that check is missing in the ServerThread class. (Interestingly enough, there is a source code comment in PooledInvoker.java which says exactly the opposite: that every ServerThread checks if it is allowed to add itself to the pool.)

These limitations of the ServerThread and PooledInvoker classes were eliminated by modifying JBoss' source code. Within the ServerThread class, the "setMaximumPoolSize" now tries to immediately discard all threads in the pool that are in excess. Additionally, the worker threads (of

type ServerThread) now check if they are permitted to go back to the pool when they have completed processing a client request. For these modifications the following source files were modified (locations respect to the source distribution root directory):

`.\server\src\main\org\jboss\invocation\pooled\server\PooledInvoker.java` (lines 451-470)

`.\server\src\main\org\jboss\invocation\pooled\server\ServerThread.java` (lines 183-189)

The modified source was then rebuilt with the ant tool [Ant]. After rebuild, the only file affected by the modifications is `server\default\lib\jboss.jar` of the JBoss installation.

4.2 The Autonomic Manager

The central piece for providing autonomic computing behavior in the basic architecture of Figure 1 is the autonomic manager. All the AC features that are listed in Table 2 have to be enabled by the AM. Given that the AM has access to the managed system, such that it can monitor and influence its behavior, the AM executes a basic management cycle as depicted in Figure 10 [KC03]. This cycle — which is also referred to as "MAPE" cycle [Ste05] — consists of four stages:

- **Monitor:** the AM extracts status information from the managed system.
- **Analyze:** looking at the data collected in step 1, the AM determines its view of the managed system (e.g. by condensing the available information into an indicator for system status by calculation of a metric).
- **Plan:** comparing the results of its analysis to the rules the AM has been given from a human administrator, the AM decides what to do.

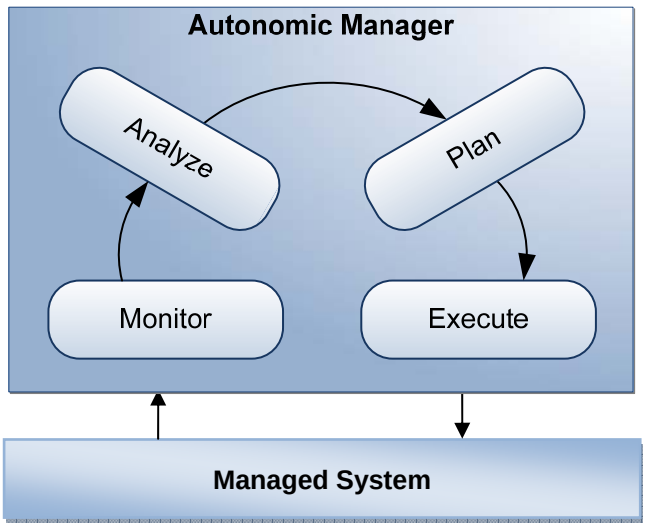

Figure 10: Basic Management Cycle

- **Execute:** the AM finally puts its plan into action, using its interface to the managed system and effecting changes in the system's behavior.

Four AC functionality requirements are laid out in Table 2 on page 29. These are:

- Checking for correctness of current parameter sets
- Monitoring resource usage and reconfiguring the system if needed
- Reducing the response times for clients
- Restart failed applications

For each of these requirements the AM developed for this work executes the MAPE cycle. In the implementation of the AM, the four AC capabilities as listed above were given different priorities to be followed. This had to be done, because the different management goals which the four AC capabilities define may be in conflict with each other.

4.2.1 Self-Configuration Features

The AM has two capabilities that are self-configuration AC features. These are configuration checking and resource usage monitoring with reconfiguration if necessary. Both of them are described in the following sections.

4.2.1.1 Checking current configuration parameters

Of the many configuration parameters that a complex system like JBoss offers, three were selected to be periodically checked by the AM:

- the size of the database connection pool,
- the pool sizes for HTTP request processing threads and
- the pool size for EJB invocation processing threads.

Figure 11 shows, where these different pools come into play.

Every request that arrives at JBoss via the embedded Tomcat has to wait for a thread from the HTTP thread pool to be available in order to be processed. If the client request involves calls to an EJB on the server, the request also has to acquire a thread from the EJB thread pool. And if the EJB call during its execution needs to make a request to the database (DB), a connection from the DB connection pool has to be retrieved. Requests that are made from the client directly to the EJBs require a thread from the EJB pool to be processed, and again, if the EJB needs to read or write to the database, a connection from the DB connection pool has to be acquired.

The amounts of threads/connection in the different pools limit the number of client requests that can be concurrently processed. From looking at Figure 11 it becomes obvious that having more connections in the DB connection pool than threads in the EJB pool is a configuration that is not useful, because the number of DB connections that can be used at the same time is limited by the number of threads in the EJB pool. Having unused DB connections, however, imposes a penalty in terms of resource usage for both JBoss and the database server. It is thus desirable not to not have more connections in the DB connection pool than threads in the EJB thread pool. On the other hand,

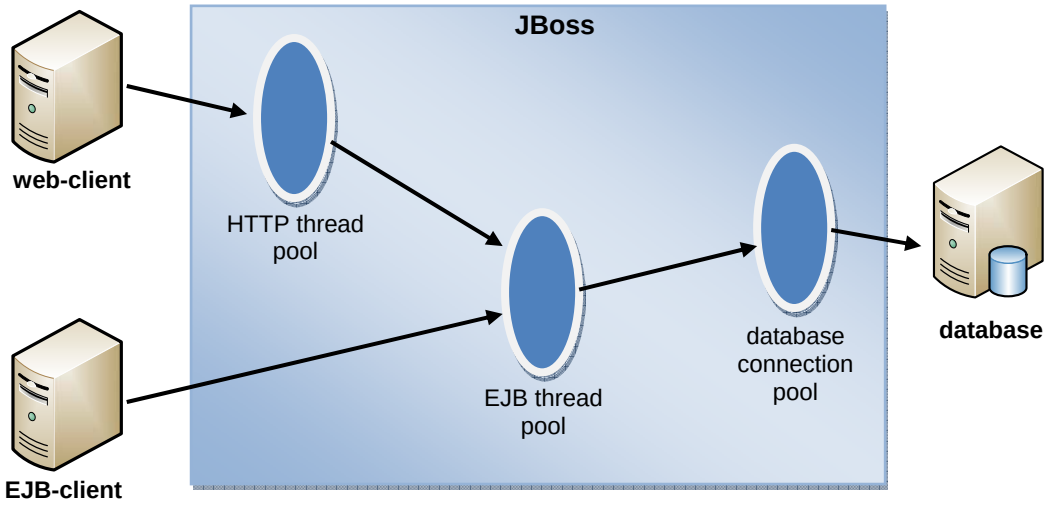

Figure 11: Pooling in JBoss

if the ratio of EJB threads to DB connections becomes too large, many client requests will fail because they cannot get a DB connection within the configured timeout.

Finding the optimum relation of DB connection pools to EJB threads is a non-trivial task. For the AM developed here, a change of the number of DB connection is not allowed. A change can only be done by a human administrator. This restriction has been introduced, because the database server is outside of the scope of the AM. This means that the AM has no knowledge about the number of connections that the database will at most accept. Additionally, the AM is not aware of any other users of the database for whom an increase of database connections for JBoss might imply degradation of service.

The AM will first of all maintain the number of pooled EJB threads above the number of available DB connections. All other changes to the amount of threads in this pool will be done during optimization or when memory becomes low.

4.2.1.2 Observing resource usage and adapting the configuration

The two main hardware resources that JBoss relies on are CPU time and memory. It is consequently essential to efficiently use these resources. As JBoss runs within a JVM, it does not have direct access to system memory, because constraints are enforced by the JVM. How much CPU time JBoss uses is however not constrained by the JVM. In the case that a component with JBoss malfunctions such that it enters an endless loop, that component will heavily degrade the system's performance. A software program executing an endless loop will use as much CPU time as it can possibly use. The AM developed, therefore, constantly monitors CPU utilization.

Should the value for CPU utilization exceed the configured threshold, the AM will request details for all the threads currently running within the same JVM as JBoss. If it can determine a thread responsible for the excessive CPU usage, the AM will see if it can stop the culprit. such behavior is not normal for the kind of application at hand, but rather caused by error. The thread causing excessive CPU load belongs to a user application that has been malfunctioning. The AM will therefore try to cure the problem by restarting that user application.

Even though “endless loop”-failures will drastically decrease performance, they will normally not cause a breakdown of JBoss. This is quite different, when the amount of available memory becomes low. Since every client request will cause memory to be allocated while it is being processed, request processing will fail if there is not enough memory. The exact location in the request processing routine, at which an error due to memory shortness (“OutOfMemoryException”) is encountered, is unknown in advance. But at runtime this location really matters, because if such a memory error is encountered in one of JBoss’ own modules, this error may — and most likely will — make that component unusable.

Furthermore, whenever exceptions are encountered, they will be processed by JBoss’ extensive logging facilities. But as handling of these error messages also needs memory, this will in most cases cause a complete crash of JBoss.

The AM developed here tries to proactively prevent that from happening. It periodically monitors the amount of available memory and when that amount drops below the configured minimum threshold, the AM will try to free up memory by:

- ordering JBoss to clear up all EJB caches and by
- reducing the amount of concurrency in request processing by reducing the number of worker threads.

Furthermore, the AM will inform the human administrator of every memory-related problem, because in the end, such memory failures can only really be avoided by guaranteeing that the JVM has enough memory available. This means, that the JVM startup parameters have to be modified accordingly. Since such an operation also requires a restart of the JVM and thereby also of all the services running within that JVM, an action like this will cause service interruption and must be sanctioned by the human administrator.

4.2.2 Self-Optimization Features

The autonomic element that has been created, comprising JBoss as managed system and the developed autonomic manager, also has the *self-optimizing* AC capability. Any optimization requires the optimization goal to be defined, which is almost always the minimization or maximization of an output variable of the system, which in this case is response time.

That output variable will be the result of a certain function with a number of input parameters. If the function and the values of all inputs were known, then the optimum could be derived by mathematical analysis. However, application servers are highly complex systems, in which all the interactions that take place would have to be taken into account. The complexity that such a system exhibits is difficult or even impossible to quantify even for very simple models. Java objects are pooled, cached and replicated both by JBoss and the JVM for performance optimization reasons. These techniques do lead to an improved runtime environment, but at the same time make the search for optimum parameter sets a frustrating and sometimes even pointless task.

The approach chosen here is a trial-and-error approach to address the problem. It is known that thread pool size for EJB invocations has the most significant effect on system performance, given

that the other performance parameters are constant [ZQL06]. The AM will consequently only use the *EJB thread pool size* as parameter for its optimization efforts. The method applied for optimization is as follows:

The AM makes small changes to that parameter and subsequently observes the system. If it can conclude that system performance improved due to the change effected, it takes the new values of both the configuration parameter and the performance metric to be optimized as new starting points for the next optimization step. Should the change done to the system have diminished performance, the AM undoes its last change and try to adjust the parameter in the opposite direction for the next optimization iteration.

Should the AM in its optimization effort be unable to achieve a better performance value than the one it has as reference from previous optimizations during more than ten iterations, it restarts its optimization effort. This means that the AM takes the current value of the performance indicator as new reference value and again tries to optimize as described.

The method applied for the search of an optimum is similar to *"hill-climbing"* as described in [Kai89 p. 53]. The major difference is that the approach presented here operates in a dynamic environment and consequently never settles for any value as *the* optimum. (It is much like climbing a hill, whose topography constantly changes.) A drawback of this kind of method is that it might settle for a local optimum.

Whenever memory shortness is diagnosed by the AM or the *thread pool size to DB connections ratio* becomes too large such that it causes failures, the AM will cancel the last optimization step it has taken. The AM has, however, no memory to save values which caused failures, because it operates in a dynamic environment. Parameter sets for the managed system that cause errors at one point in time may be totally fine at another.

The scheme which the AM follows in trying to optimize the managed system's performance is quite simple. Other optimization techniques that have been found in the literature (cf. chapter 2) — such as fuzzy control or techniques based on prediction — were not implemented, because they are beyond the scope of the case study presented.

4.2.3 Self-Healing Features

Applications and services are mostly deployed on top of JBoss' JMX-based architecture using JBoss' hot-deployment mechanism. As JBoss constantly monitors all directories that are configured as hot-deployment locations for changes, an application can be installed simply by copying the Java archive (JAR) file, which contains the application, into one of these directories. Replacing a certain file with a newer version will cause JBoss to undeploy the old version and load the new one. Deleting a file from the hot-deployment directory results in the de-installation of the corresponding application or service.

It is because of the hot-deployment mechanism that applications which are packaged into the same JAR file can only be re-started (which means re-deployed) together. Invoking the "`redeploy()`"-

operation of JBoss' main deployer component on a specific application has the same effect as removing the related file and then undoing the removal.

Application failures in the Java programming language are addressed by exceptions. An exception that has been generated at a specific point within an application will continue to "travel" upwards through the stack trace until it is "caught" by an exception handler. Any exception thrown within an application deployed on JBoss ultimately ends up being caught by JBoss' logging exception handler if it is not processed by the application itself. This handler will forward the exception's message to all logging targets, as for example the console, a log file or exception notification listeners.

The AM subscribes at startup for notifications at the management interface for JBoss that has been developed. Reception of WS notification is possible for the AM, because it integrates an instance of the Muse runtime itself (together with an instance of the minimalistic HTTP server already discussed), which handles decoding of the WS notifications and extracts the message received. The received message is then treated by the custom *Notification-reception capability* that puts the contents of the message into a Vector, which is shared with the main AM thread.

At every management cycle the AM will check for messages in the Vector. Searching the exception message text for occurrences of known strings, the AM determines if and how it will react to the exceptions. Two kinds of exceptions cause a reaction from the AM in this case study:

- Exceptions thrown by a custom built test-application[8], which indicate that aforementioned application has crashed and needs healing.
- Exceptions caused by database connection timeouts. These indicate that the ratio of EJB thread pool size to DB connections is too large.

In the first case the AM will request the full URLs of all applications deployed. Within that list it looks for the application that caused the exception and in the following step requests a restart of the malfunctioned application. This is an example of the *self-healing* AC capability.

In the other case the AM cancels its optimization efforts, which causes the EJB thread pool size to be reset. As such, this case does not affect the AM's *self-healing* capability, but rather serves as an indicator to the AM's *self-optimization* efforts.

4.2.4 Program Design

The structure of the developed autonomic manager is depicted in Figure 12. The parts that are within the dashed line have been created in the course of this thesis. The *Muse runtime* on the right is the same as in Figure 9 on page 37 and is described in section 4.1.5. This also applies for the *mini HTTP server*, which is covered in section 4.1.5 as well.

The custom Notification-reception capability handles messages received by the Muse runtime. It extracts the contents of exception messages and stores them for the Autonomic Manager Core. All such messages will then be acknowledged to the sender by the Muse runtime.

[8] The custom built test-application is a web application consisting of a single servlet that randomly produces failures, which cause the complete application to malfunction until restarted.

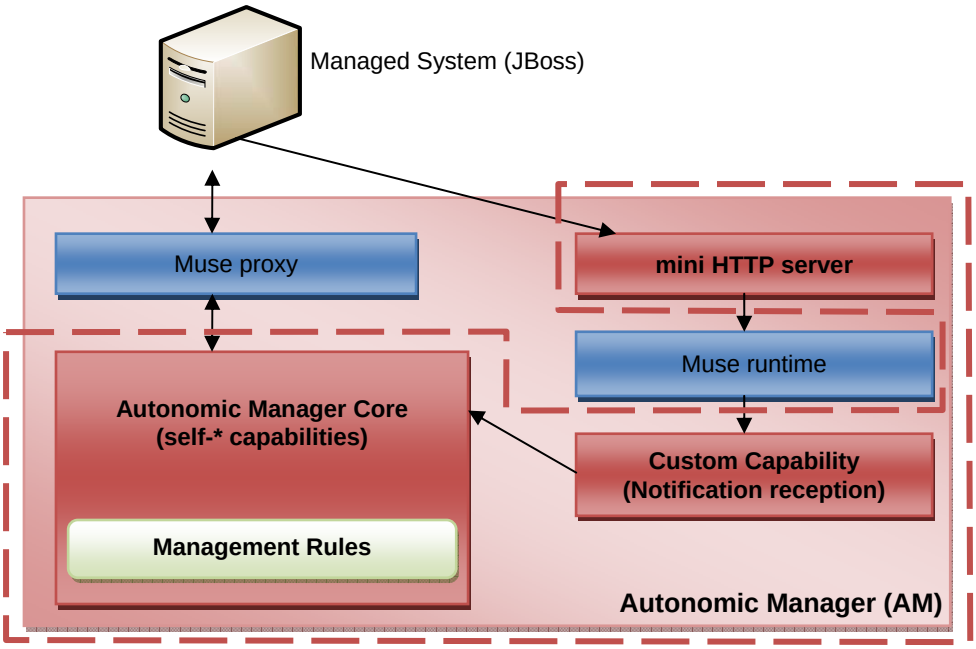

Figure 12: The Autonomic Manager's Structure

The Muse proxy is generated from the same WSDL file that was used for generating the server-side WS interface (cf. section 4.1.5). Again, the WSDL2Java tool that comes with the Muse framework has been employed to do the code generation. The generated code translates Java method invocations into the WS messages, sends them to the specified WS endpoint, receives the reply and translates the WS-encoded reply into Java objects. For primitive data types the generated proxy will directly translate the XML-representation of the data type into the corresponding Java data type.

Complex data types, however, are returned as Java objects of type `org.w3c.dom.Element`. These represent the portions of XML that contain the returned data. For methods that return such complex data, the return values can either be parsed in the code that makes the method calls or by custom serializers. The implementation of the AM directly extracts the needed information from the returned XML representation without parsing it into Java objects.

The top level priority of the AM is the self-configuration capability, as seen in Figure 13.[9] On the one hand, the AM requests the current configuration parameters and compares their relative values with reference ratios set by a human administrator. If the values are not within bounds, the thread pool size parameter is updated with a default setting.

On the other hand, the AM constantly requests system health information. More specifically it monitors CPU utilization and memory status. Should the amount of available memory drop below a specified threshold, the AM will cancel any optimization efforts and inform the administrator. If CPU utilization is not within the range that the AM has been told to consider as normal, it will try to find the responsible thread and restart the corresponding application.

[9] The notation used in the figure is UML 2.0 as described in [RHQ05]

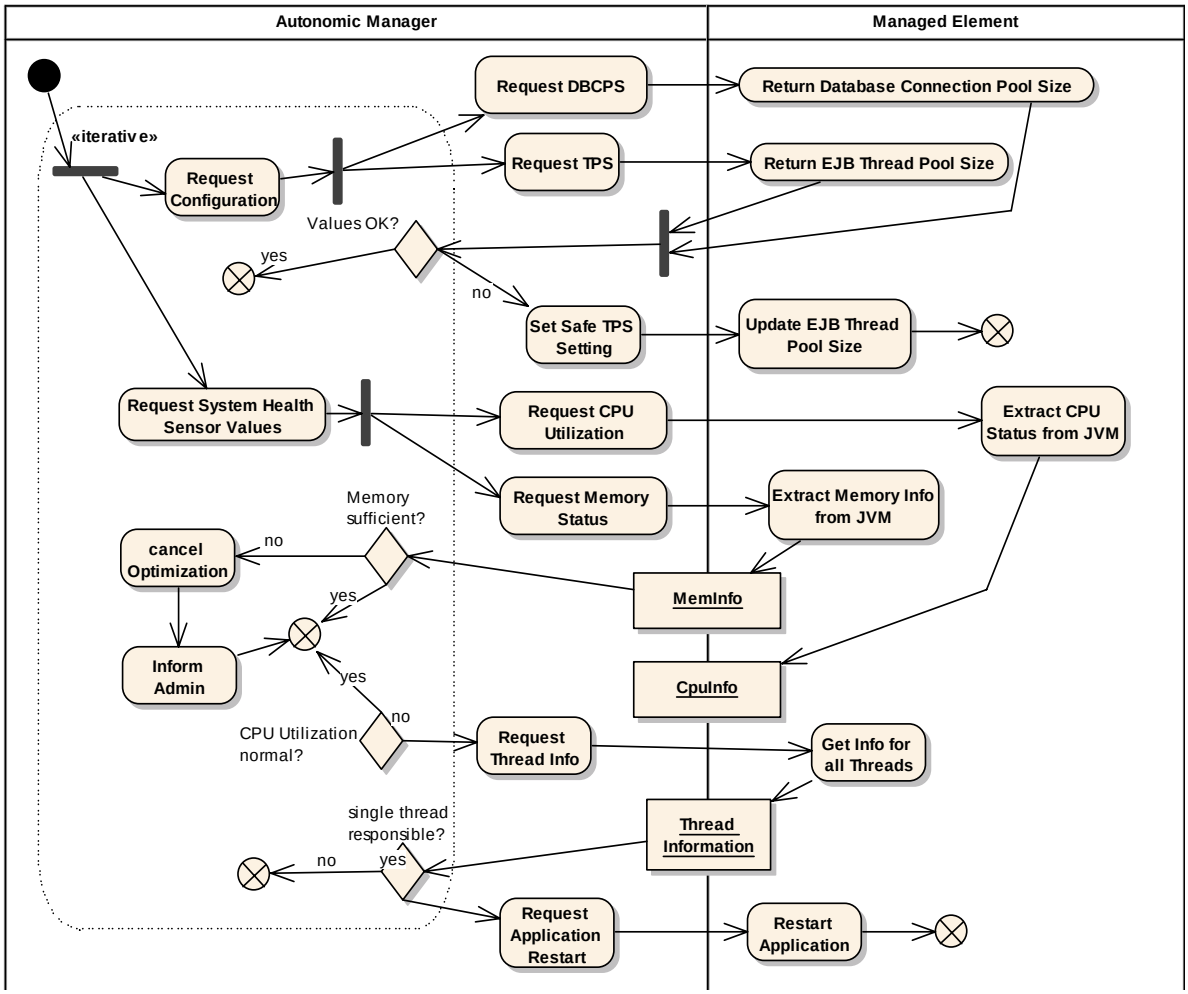

Figure 13: Autonomic Manager Activity — self-configuration

Only if the system's memory and CPU data is within the limits that have been specified to the AM by a human administrator, will the AM carry on with lower priority tasks. If either of the two main resources to the computing system is not available to the extent required, the AM will first try to solve this situation as described in section 4.2.1.

If the AM is assured that basic system operation is not hindered by resource problems, it goes on to see if any problems at the application level have occurred. This is shown in Figure 14. At first, the AM subscribes for notification messages. Any exception notification message sent by JBoss will then be put into a data store for such messages.

The AM continuously checks the data store for messages. Every message found in the data store is examined and treated corresponding to its type. In the event that database connections have timed out, the self-healing behavior interacts with the self-optimization by cancelling optimization to ensure correct database connectivity.

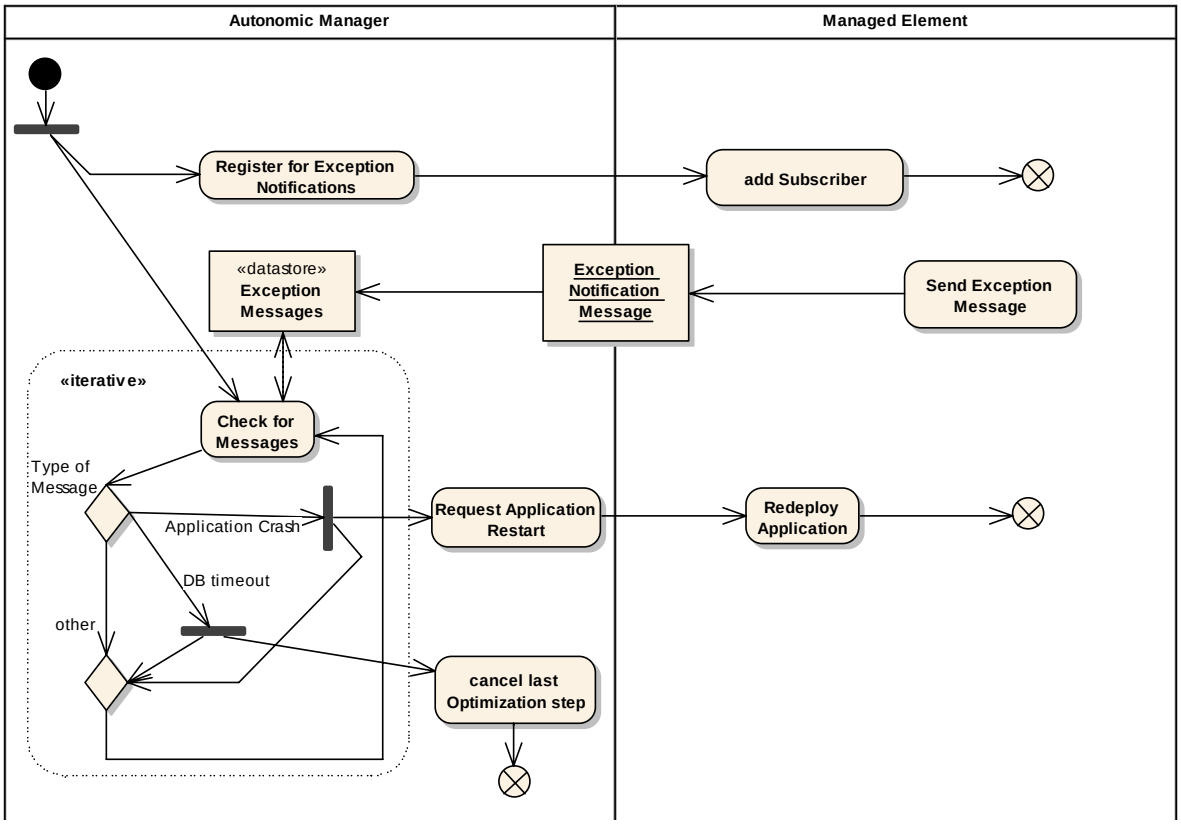

Figure 14: Autonomic Manager Activity — self-healing

Only if all other aspects monitored by the AM are satisfied the AM will try to optimize the system's performance as depicted in Figure 15. This is set as lowest priority task, because optimization is not vital to the overall system's working condition. (And it may even be counter-productive, because optimization usually requires more resources.)

All the management activities shown in figures indicated above will execute iteratively. They will only be ended if the AM is terminated by a human administrator through its text-based user interface.

4.3 Parameters and Performance Metrics

The JBoss application server offers a great variety of parameters which allow influencing the system's behavior and performance. The previous sections have already presented implementation details, which make it clear that only very few of these parameters have been used for the case study at hand. In addition, there is a single performance metric that is used throughout the implementation: response time. This section provides insight on the parameters and possible performance indicators that were investigated prior to implementation.

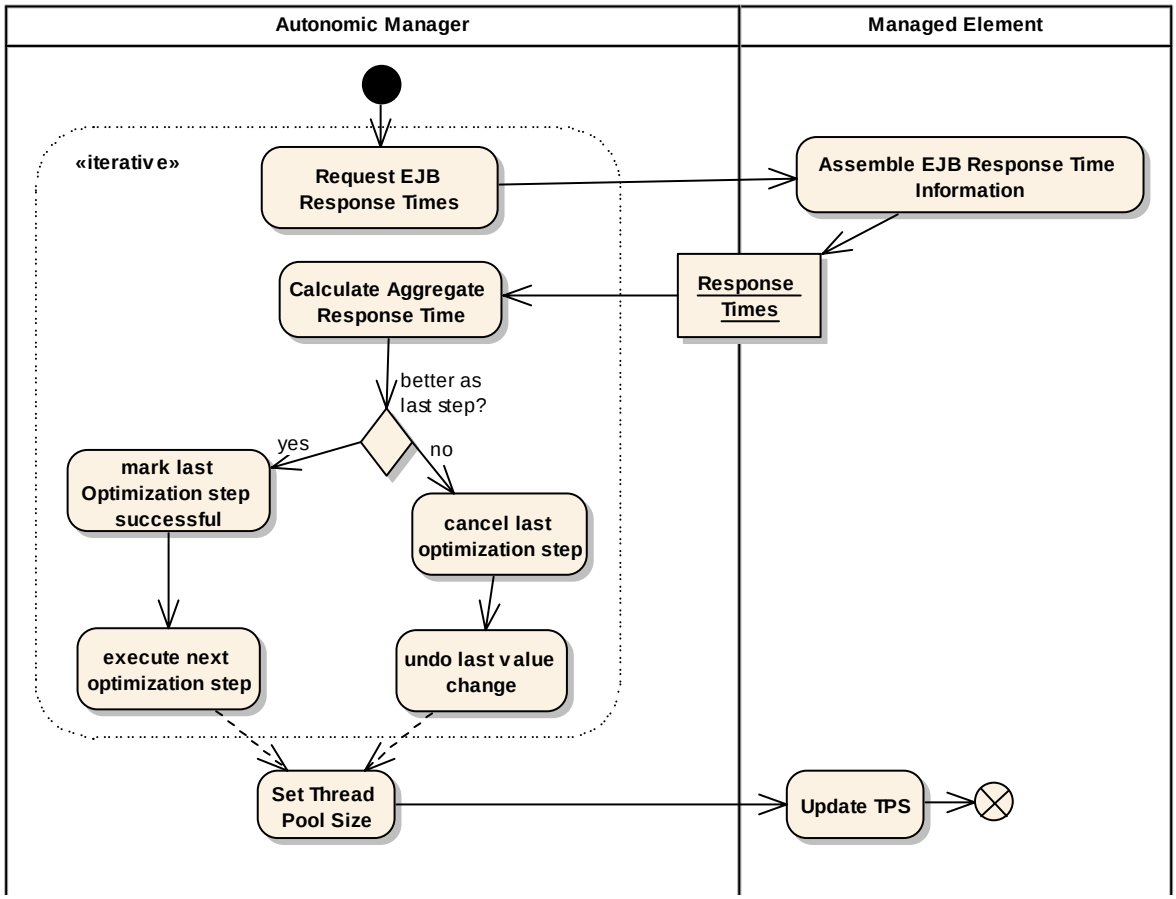

Figure 15: Autonomic Manager Activity — self-optimization

4.3.1 Configuration Parameters

For a parameter to be usable in this AC case study, it has to be

- accessible through some application programming interface,
- modifiable without restart of the server and
- traceable.

Traceability in this context indicates that it has to be possible to confirm changes to the parameter by changed system behavior. This last prerequisite for any parameter that would have been chosen was clearly the strongest one, because any kind of management is only possible if parameter changes can be clearly related to effects on the system's performance indicators.

The initial set of performance parameter that has been investigated — partly inspired by [RRJ03] — is:

- Web Container Thread Pool Minimum
- Web Container Thread Pool Maximum
- EJB Container Thread Pool Size
- Data Source Connection Pool Size

- SQL Statement Cache
- JVM minimum heap size
- JVM maximum heap size
- EJB cache maximum size

All of the parameters on the above list were considered. Afterwards, the final selection of performance parameters for the case study was done, as described below.

The values for the *web container thread pool* are indeed accessible and modifiable while the system is running. Moreover, changes made to the minimum and maximum settings can readily be observed. However, these parameters were not used in the AM, because the benchmarking and testing facilities primarily used do not build upon HTTP as their transport mechanism, but rather use direct EJB invocations over a specialized RMI protocol. The *EJB thread pool size* was consequently chosen to be used, and is discussed in section 4.2.1. This choice is also supported by the statement "(…) thread pool performance in EJB container has the most significant effect on the whole system performance (…)" found in [ZQL06].

The *data source connection pool size* is also a viable candidate for *self-configuration* and *self-optimization* efforts of the AM. But this parameter does not respect the system borders set for this use case. If it were to be changed, there must be assurance that the database server to which the connections are held supports the new setting. Even constraining the AM with a maximum value for this parameter could cause problems. Whenever the database connection pool size is increased, the whole connection pool has to be re-initialized. Using the monitoring tool for the MySQL database that was used, it could clearly be seen that whenever the pool size is changed, the number of connections made to the database does not respect the defined maximum value. This behavior and the DB connection pool-related problems mentioned in section 4.2.1.1 led to the exclusion of this parameter for use in the case study.

The size of the *SQL statement cache* is a parameter that cannot easily be configured during runtime of the system. Its value is set in the configuration file that sets up the relational data source. It can thus only be modified by editing the configuration file and restarting the data source service in JBoss. But this restart breaks all applications that depend on the corresponding data source. In order to achieve continued functionality of these applications, they would also have to be restarted. But because of the complicated dependencies and the laborious configuration method, the parameter was discarded for use in this case study.

Restrictions to the *JVM heap memory size* have a clear impact on system performance. Especially when multiple different services are located on the same physical hardware and hence have to share the same memory, it is advisable to restrict memory sizes for the different services. Unfortunately, for all JVM implementations currently on the market, the maximum heap size cannot be changed during runtime. In the cases where it can be influenced, minimum and/or maximum values are specified to the JVM as startup parameters on the command line. A modification of these values would thus imply a restart of the JVM and all the programs and applications currently running within that JVM instance. During that time, the system would be completely out of service, so it is

not desirable to effect frequent changes. The parameter has — because of these restrictions — also not been further investigated for the purpose of the AM that was developed.

The last item on the list of configuration parameters that were investigated is the *maximum size of the EJB cache*. All Enterprise Java Beans that are created in JBoss' EJB container are cached and kept in memory even when not currently used. The cache policy employed by default is "aging out", which means that EJB objects that have not been used for a certain period of time will be removed from the cache and the memory that they occupied will be freed. Even in situations when the JVM runs out of memory, the cache still holds a great number of unused EJBs, as they are only removed after the mentioned timeout. Due to its modular architecture, JBoss allows custom cache policies to be implemented. However, all implementation efforts carried out to build such a custom cache policy were unsuccessful. This is why this parameter setting was not further pursued. The problem of memory dedicated to unused EJBs was, however, addressed by the developed AM (cf. section 4.2.1.2).

4.3.2 Performance Metrics

Autonomic computing is an initiative pursued to make computing systems better. But this notion of "better" has to be quantified somehow. One possibility of quantification is to compare performance metrics of a system running with AC functionality to a system that does not have AC features. In the case presented — the JBoss application server — there are different possible performance metrics that could be considered.

4.3.2.1 Throughput

The first performance indicator that has been considered is the number of requests that are processed per amount of time. An indicator like this is also called throughput. This is definitely a value by which the system can be objectively rated, but not without prerequisites. To have comparable results, the system must for every test run be placed under the same conditions. Taking JBoss as the example, the tests must execute the same number of operations in the exact same order and operate on equal input data. These requirements can be met in a benchmark. If the system should be used in a production environment though, the conditions under which it is placed will never be exactly the same. Throughput is for that reason a performance metric that can be used in an offline benchmarking test, but it is not a value by which the system should really be measured under normal working conditions.

4.3.2.2 Time to Completion

Another benchmark test is to measure the time it takes the system to complete a certain set of tasks. This time-to-completion is again a good indicator for comparing systems in a predefined, tightly constrained environment. Normal working conditions will not allow obtaining objective results with such a kind of metric in a short amount of time, because statistical data needs to be collected over a longer period of time.

4.3.2.3 Response Time

The performance metric that is considered best for this case study is response time. Also used by various scientific investigations into service performance [FSE04, ZQL06, RRJ03], response times are system status indicators that have certain validity both during a benchmark test run and in a production environment.

Response time is defined as the time it takes the system to come up with an answer to a single request issued by a client. The term response time is mostly used in the context of communication networks. In such environments, the requests for which response times are measured are typically very simple. Such a request may be as minimalistic as to just ask for an echo. (The `ping` command used for testing network connectivity does exactly that.) In computing environments which are confined to a single, physical computer, the expression "response time" is seldom used. The terms *method invocation time* or *command execution time* more properly describe the circumstances for such a kind of performance metric on a single computer.

In networking applications, response times are always measured at the client side, as it is mostly the time lag introduced by the network that is of interest. Carrying out these kinds of measurements requires access to the clients. This is something that can be done in a benchmarking situation [ECperf], but not in a real-world deployment. The acquisition of a performance metric needs to be done completely within the boundaries of the system to be measured.

As discussed in section 4.1.3 this case study uses "response times" measured at the server side. Even if different terminology would usually apply to such a kind of measurement, the values obtained are named "response times" in accordance with other work in the field [FSE04].

The times taken in this case are mean values per invocation target. This way, response times could be calculated that use weighted algorithms, for example giving higher weight to operations that are known to be quite cost-intensive and could disregard operations that usually do not take up any considerable time at all. The implementation presented here leaves the details of how to make use of the response times obtained at the server side to the management client, which in this case is the AM.

Of course, giving the AM the possibility to weigh the importance and impact of the different targets' response times opens up a lot of possibilities for flexible adaptation to different environments. But it also requires the AM to be aware of the applications that are deployed and of their different operations. In some cases this might not be desired or even possible. For such cases, the AM employs a more generalized way of making use of the times obtained: it simply calculates the time that a single request has needed on average.

4.3.3 Influences on Measured Response Times

Especially for the AM's optimization efforts, it is critical to be able to relate configuration modifications to changes in the measured response times. The different other influences on response times should, therefore, be minimized — at least for benchmarking.

Of particular importance during such stress-tests is a constant load under which the system is to be placed. If the benchmark application that is used changes its request patterns in every benchmark run, then the results of these benchmarks cannot be compared for certain time intervals, but only for the whole test run. Additionally, the benchmarking application should operate on the same data for every test run.

While it is possible to use the same configuration for all test runs, the request pattern will not be identical for the testing and benchmarking applications used in this case study. This is mainly caused by the multi-threaded nature of the applications. Because the scheduler, who decides the order in which the different threads are executed and also the times that each of the threads is allowed to use the CPU, is not deterministic, the request pattern seen at the system under test will vary for every test run. Moreover, since testing is done with applications that access the system under test via a network, the network's indeterminate characteristics will also contribute to the changing request pattern.

Especially for the ECperf benchmarking application discussed in section 4.4.1, having equal data on which to operate is impossible. As this application randomly generates the data it will operate on in a distinct initialization routine, the values used will differ from run to run.

Another strong influence on the response times measured is the execution behavior of the garbage collector (GC). The GC will run at regular intervals and also when a threshold for low memory has been reached. Because the GC also puts load on the CPU, response times measured while the GC is running will be higher. Since the location in time when the GC runs is unknown to the AM, it will interpret the changed performance as a result of the current setting of the configuration parameter(s).

The configuration of the machine on which the system-under-test runs also influences the measured response times. It is mainly the other applications that are running side-by-side with the JVM that cause measurement influences, because they share the same CPU. File indexers, virus scanners, malware detection utilities and automatic update programs should consequently be terminated or deactivated for the time span of the benchmark run.

As not all of the influences other than changed system configuration could be eliminated, the measured response times will always exhibit certain variations that are not caused by the test itself. By running the same tests multiple times and combining the acquired results, it is possible, though, to obtain sufficient statistical data.

4.4 Testing and Benchmarking

The case study involves many different modules that were independently developed. First tests of all these units were carried out by hand without automated testing tools. But in order to test the full developed autonomic computing functionality, two specialized benchmark and test applications have been used.

Both of them are described in detail in the following two subsections. Concluding this case study, the results of the test that have been carried out are presented.

4.4.1 ECperf

ECperf is an Enterprise Java Beans (EJB) benchmark developed by Sun Microsystems and the Java community [ECperf]. It is meant to evaluate performance and scalability of J2EE application servers. More specifically it stresses the memory management, connection pooling, passivation and activation and caching capabilities of the tested system. As its aim is to provide objective data for comparison of different server and hardware products, its specification is very detailed and also covers topics like costs and pricing, benchmark result submission and benchmark run rules. Compared to other industry benchmarks it provides an objective view on the tested system [ZLQ03].

The ECperf benchmark models a large company's activities such as order processing, supply management, manufacturing and corporate data keeping. It is thus designed to comprise four domains: Customer, Manufacturing, Supplier and Corporate. Each of these is implemented by a set of session and entity EJBs in the ECperf application.

The benchmark tool — as available from `http://java.sun.com/` — includes:

- An EJB application to be deployed on the target EJB container (the ECperf application),
- a servlet to emulate an IT connection to a supplier company,
- a utility to load the databases needed to model all the EJBs used in the ECperf application and
- driver applications, which will create load on the system under test (SUT).

To run the ECperf benchmark, the following will also be needed:

- An EJB container/server to deploy the ECperf application,
- a servlet container to deploy the supplier emulator application and
- a database management system to hold the data that ECperf operates on.

Figure 16 gives an overview of ECperf's architecture. The system under test can be seen in the lower left and contains the EJBs for the four domains of the benchmarking application. At the top of the figure are the Drivers, which stress the system by sending requests via Java's RMI protocol, as indicated by the arrows marked "RMI calls". There can be any number of drivers, one of which has to have a controller component that orchestrates the load. On the right hand side are the supplier emulator and the database, which can simply be seen as infrastructure required for ECperf.

To be able to deploy the ECperf benchmark on the JBoss application server, the deployment descriptors of the ECperf application and of the supplier emulator were modified to conform to the format expected by JBoss. The JBoss-specific deployment descriptor "`jboss.xml`" was also added to all the applications. This descriptor file defines the mapping of EJBs to database entries. Finally, the database structure was created in MySQL and a corresponding data source configured for JBoss.

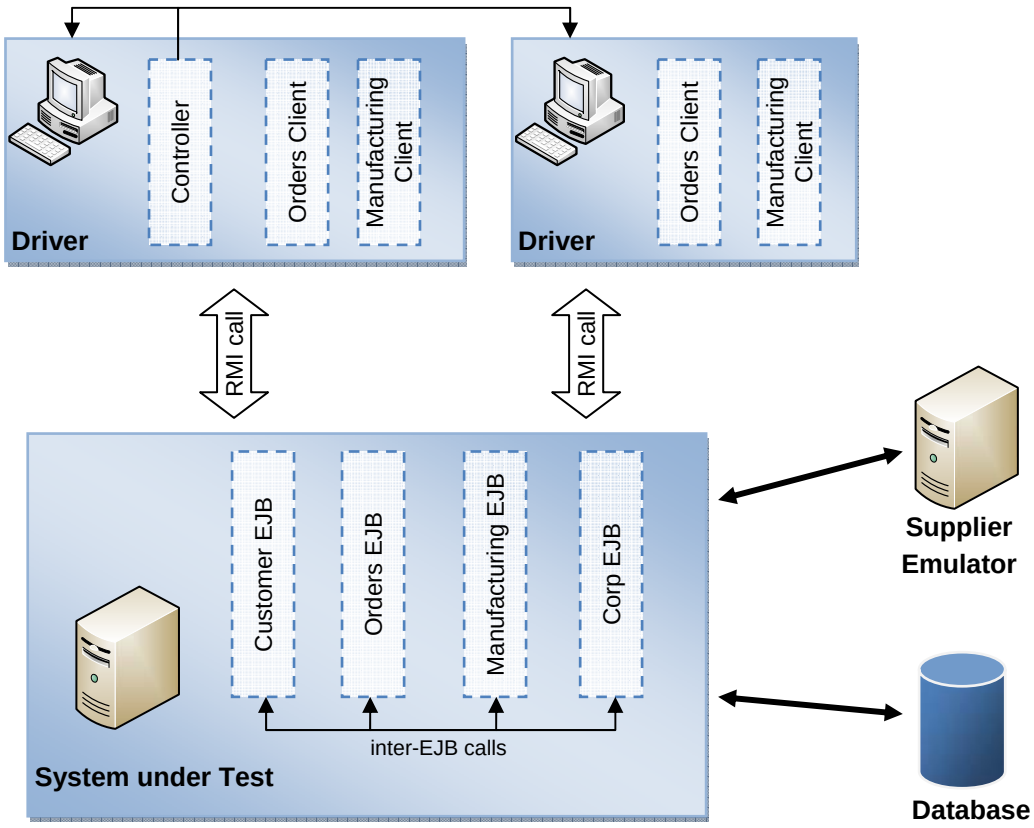

Figure 16: ECperf Architecture

Then the database was loaded with data generated by the "`loaddb`" utility that is part of the ECperf package. All that is left then is to invoke the benchmarking script, which will first start a naming service (`rmiregistry`) for all the drivers to register with, and then start the controller and all the benchmarking agents (clients).

The benchmark runs executed by the ECperf driver applications are set up through a configuration file. At startup of the controller, this configuration file is parsed and the clients for the different domains are parameterized. It is not possible to change the load exerted on the system under test — which is determined by a parameter called "injection rate" by the ECperf benchmark — during the benchmark run.

Once the time specified as benchmark duration has passed, the controller will collect performance statistics from all the clients and create a summary. The information collected by the benchmarking toolkit for performance analysis consists of:

- a histogram of the response times over time as perceived by the clients and
- the number of "workorders" (client requests) processed by the SUT over time.

All the response times are measured at the client side. As response times are only used to create a histogram, information about the mean response time at specific points in time during the benchmark is not available. It is not possible to use these statistical data as a reference for the developed AM functionality, because the kind of data collected is not directly comparable.

The ECperf benchmark is in this case study only used to create the load on the server. Since the load generated by the application is not modifiable during benchmarking runs, increasing the load is achieved by starting a second instance of the ECperf benchmarking tools on a different computer. The overall test setup can be seen in Figure 17.

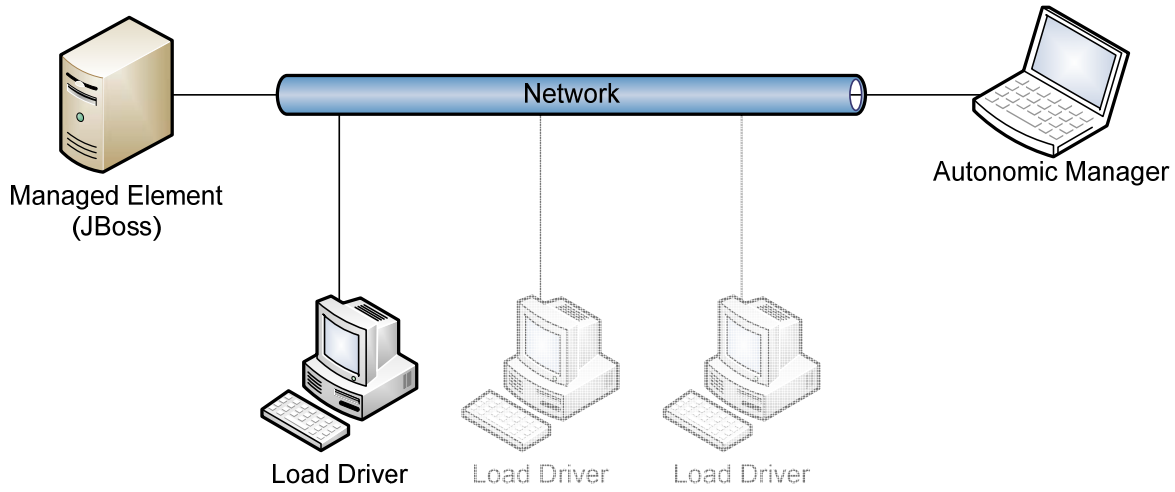

Figure 17: Test Setup for ECperf

In the test setup used, the *database management system* is located on the same computer as JBoss. For this reason it is not separately shown in the figure. The *supplier emulator* is not shown as well, because it resides on the same machine as the primary load driver. The freely available MySQL (version 5.0.24a-community) was used as database system for all the tests that have been carried out. The supplier emulator application was deployed on a JBoss application server instance (version 4.0.5.GA) with a minimal configuration necessary to run the emulator.

As ECperf's access to the SUT is always via RMI calls, a different testing tool had to be found to show the AM's *self-healing* functionality. Hence, the following section describes the tool that is used in this case study for that purpose.

4.4.2 JMeter

The original purpose of JMeter [JMeter] was to test what has now become the Apache Tomcat Servlet Container. It is an application which can be used to create and carry out load and stress testing of server applications — thereby measuring performance — as well as testing for correct functional behavior of server software and web applications. JMeter is entirely written in the Java programming language, a fact that makes it usable on almost any platform.

An active open source community that is behind the development of JMeter is continuously extending this already very extensive tool. Over time, the functionality of the JMeter tool has increased to allow testing — called "sampling" within the context of JMeter — using the following kinds of requests and samplers:

- FTP request
- HTTP request

- JDBC request
- Java request (sampling targets have to implement a specific interface defined by JMeter)
- SOAP/XML-RPC request
- WebService (SOAP) request
- LDAP request
- Access Log sampler
- BeanShell sampler
- TCP sampler
- JMS publisher
- JMS subscriber
- JMS point-to-point
- JUnit sampler
- Mail Reader sampler

The only request/sampler type of interest here is the HTTP request. It is used in this case study for the following purposes:

- making requests to the test-application to show the developed self-healing capability and
- exercise additional load on the ECperf benchmarking application through ECperf's web interface.

Configuration of JMeter is done via its graphical user interface (GUI). All the elements that make up the test profile are hierarchically structured in a tree-like view. This can be seen in Figure 18, which shows the actual configuration that was used for the tests that have been carried out.

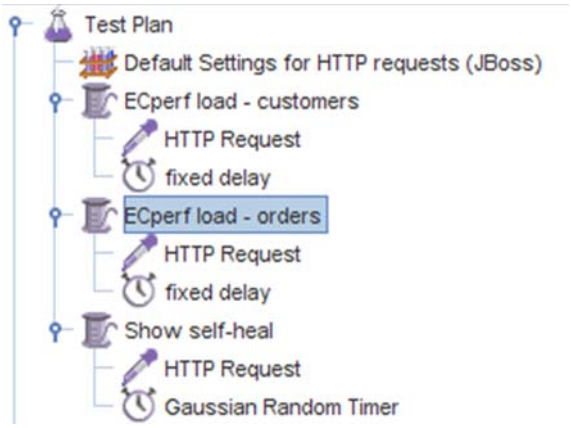

Figure 18: Configuration of JMeter used for Testing

The configuration is interpreted as follows: the test plan contains three main thread groups, which are:

- ***ECperf load – customers***: Within this thread group, ten threads are concurrently running and sending requests to the ECperf web interface. The requests are for details on customers,

which exist in the ECperf data store. Between any two requests of the same thread, a fixed delay is inserted.

- ***ECperf load – orders***: Similarly to the thread group above, this group continuously sends requests for order details. Again, a constant amount of time is waited in every thread of this group in between sending requests.
- ***Show self-heal***: The last thread group sends request to the simple test application that simulates crashes. (cf. footnote number 8 on page 46). The delays in between two subsequent requests to the self-healing test application are determined by a random timer, which will create delays of varying duration.

Every thread group contains a predetermined number of threads, which all execute the same request pattern independently from one another. All threads in all groups are started simultaneously when the start command is issued in the JMeter application. They run iteratively until the test plan is terminated with the stop command.

4.4.3 Results

At the beginning of this chapter, four *self-management* features have been presented in Table 2 on page 29 that were considered and implemented for the targeted JBoss application server. The setup and results for the tests, which were to show how each of those features have been achieved, are detailed and reasoned below.

4.4.3.1 Check the correctness and usefulness of current parameters

This first feature is the only one that doesn't need any additional application besides the JBoss application server — including the developed additional functionality — and the autonomic manager in order to be tested. The two applications were started with their default configurations. After both had initialized and were running, the *thread pool size* and *database connection pool size* parameters were modified.

The correct functioning of the AM and, therefore, the availability of the first feature could be confirmed, because every time the ratio between the two parameters went outside of the allowed range, the AM corrected the *thread pool size* at its next iteration on the self-configuration capability.

4.4.3.2 Monitor resource usage and reconfigure if necessary

In order to show the behavior of the AM in cases of scarce memory or excessive CPU usage, these two events had to be somehow created. Memory shortness can easily be caused by limiting the maximum amount of memory for the JVM in which JBoss is executed. This is done for Sun's implementation of the JVM by passing the command line argument "`-Xmx100m`", which will set the maximum heap size to 100 MB. Additionally, the ECperf benchmarking application was put into action to create work load on the system, so that it will consume memory.

With the three applications (JBoss, the ECperf load driver and the autonomic manager) running concurrently, the amount of free memory was monitored. Since the AM checked the memory status in each of its iterations, it noticed the memory shortness and took counter-measures. The functioning of this memory-monitoring was clearly observable, because the JBoss application

server continued to work properly. Under the same conditions, when the AM was turned off, JBoss crashed, because it could not recovery from “OutOfMemory” exceptions.

It is interesting to note, that the reduction of the worker threads does not have the same impact as flushing all the EJB caches. This means that the AM’s reaction will only be effective in cases, where the memory shortness is actually caused by the amount of EJB instances that are cached by JBoss. The reason for this is, that the amount of memory that each work processing thread uses is small compared to the amount of memory taken up by caching currently unused EJB instances.

The other resource monitored by the AM is CPU time. In the event of excessive CPU usage by the JVM, the AM should determine, whether this is faulty behavior and correct it if possible. Because this kind of faulty behavior was not found within the sample applications that were deployed, a custom application that would produce this kind of behavior had to be developed. The application is quite simple: it starts a thread that loops endlessly without doing any work. Such an endless loop causes the thread to use up as much CPU time as it gets assigned by the JVM’s scheduler.

Once the AM, during its routine checks of overall CPU utilization, determines, that the CPU is used excessively, it requests detailed information about all the threads. In the case of the developed sample application here, it deduces from the thread’s high CPU utilization and its stack trace, that a restart of the application to which the treads belongs is necessary. After the restart command has been issued to JBoss, the CPU utilization is back to normal. This simple heuristic can be adapted to be suitable for other applications as well.

The correct working of the CPU monitoring could be confirmed by monitoring CPU usage by hand (by means of the operating system’s process monitor) and by observing the responsiveness of JBoss to web requests through the ECperf application’s web interface.

However, there is a drawback to this capability: it only works if the application is designed to end all the threads it has created at its undeployment. JBoss does not automatically shut down all threads of an application that is undeployed. Forcing threads to stop is not possible in Java, because the `Thread` class’ `destroy()` method has never been implemented [Java6]. This capability of the AM is, therefore, only usable in situations where the application respects the design guideline mentioned above.

4.4.3.3 Revert failed applications to a functional state

This AM capability was tested using the JMeter tool presented in section 4.4.2. As shown in the tool’s configuration in Figure 18, JMeter sends requests to the sample application at random intervals based on a Gaussian distributed value. By adding a so-called “View Results Tree”-listener to the test plan, it is possible to observe the differences between the case when the AM is running (and the self-healing capability therefore active) and when it’s not.

As long as the AM is inactive, the results tree will show a couple of successful requests, before the developed sample application deployed on JBoss crashes and causes errors to be shown in the result tree. After this initial fault, only error responses are recorded.

On the other hand, when the AM was active, the result tree also showed failed requests every time the application crashed. But because the AM restarted the application and thereby returned it to a working state, JMeter's result tree indicated successful requests again after only one or two failed ones. The number of failed requests that were reported while the target application was still crashed or in the process of being restarted varied for two reasons:

- The AM's self-healing cycle time had been delayed by other AM activities.
- The load on JBoss slowed down the redeployment.

Both of these reasons were partly influenced or caused by the simultaneous execution of this test with the test of the self-optimizing capability, because during the self-optimizing test the system is under stress, and the AM also experiences more workload.

4.4.3.4 Reduce response times for client requests

The reduction of response times for client requests constitutes the AM's self-optimizing behavior and has been tested by running the ECperf benchmarking application. The set-up of that application has already been laid out in section 4.4.1. The configuration parameters that have been used for the benchmark run are shown in Table 4.

After JBoss had completed its startup sequence, the autonomic manager application has been invoked. Then, the initial values for thread pool size (TPS) and database connection pool size (DBC) were manually set through the AM's command line interface to values of 30 both. Since the AM is not allowed to change DBC, this value was constant throughout the benchmark run. Then, the ECperf benchmark was started.

Upon initiating the AM's self-managing activities, which was done about 60 seconds after benchmark startup (corresponding to the benchmarks ramp-up time), the AM tried to optimize the measured response time by adjusting the TPS value. A graph of the values obtained during this test is shown in Figure 19. The upper diagram which is labeled "Response Time" shows the mean value of all the response times measured during every interval of 4 seconds duration, similar to [FSE04] and [ZQL06]. The lower diagram shows the thread pool size as selected by the Autonomic Manager.

Table 4: Parameter Values for the ECperf run

Parameter Name	Value	Description
`txRate`	15	sets the number of threads that will concurrently send requests, thereby determining the load exerted on the SUT
`rampUp`	60	startup time for all the threads (ECperf statistics are not collected)
`stdyState`	660	actual run length of the benchmark
`rampDown`	60	shutdown time for all the threads (ECperf statistics are not collected)
`runOrderEntry`	1	activate the order client
`runMfg`	1	activate the manufacturing client
`doAudit`	0	audit the results returned to the clients

That figure clearly shows how the increase of TPS by the autonomic manager evidently reduces the measured response times. The two upwards pikes in response time that can be seen at the beginning of optimization are actually caused by the TPS increase. As at the moment, when the AM increases TPS, JBoss creates and initializes threads for all the requests that have queued up and can now be processed, because more threads are available. Since there are already as many requests queued as threads that are added, the creation of these threads creates load on the server, which in turn increases response times for the requests handled by the already existing threads.

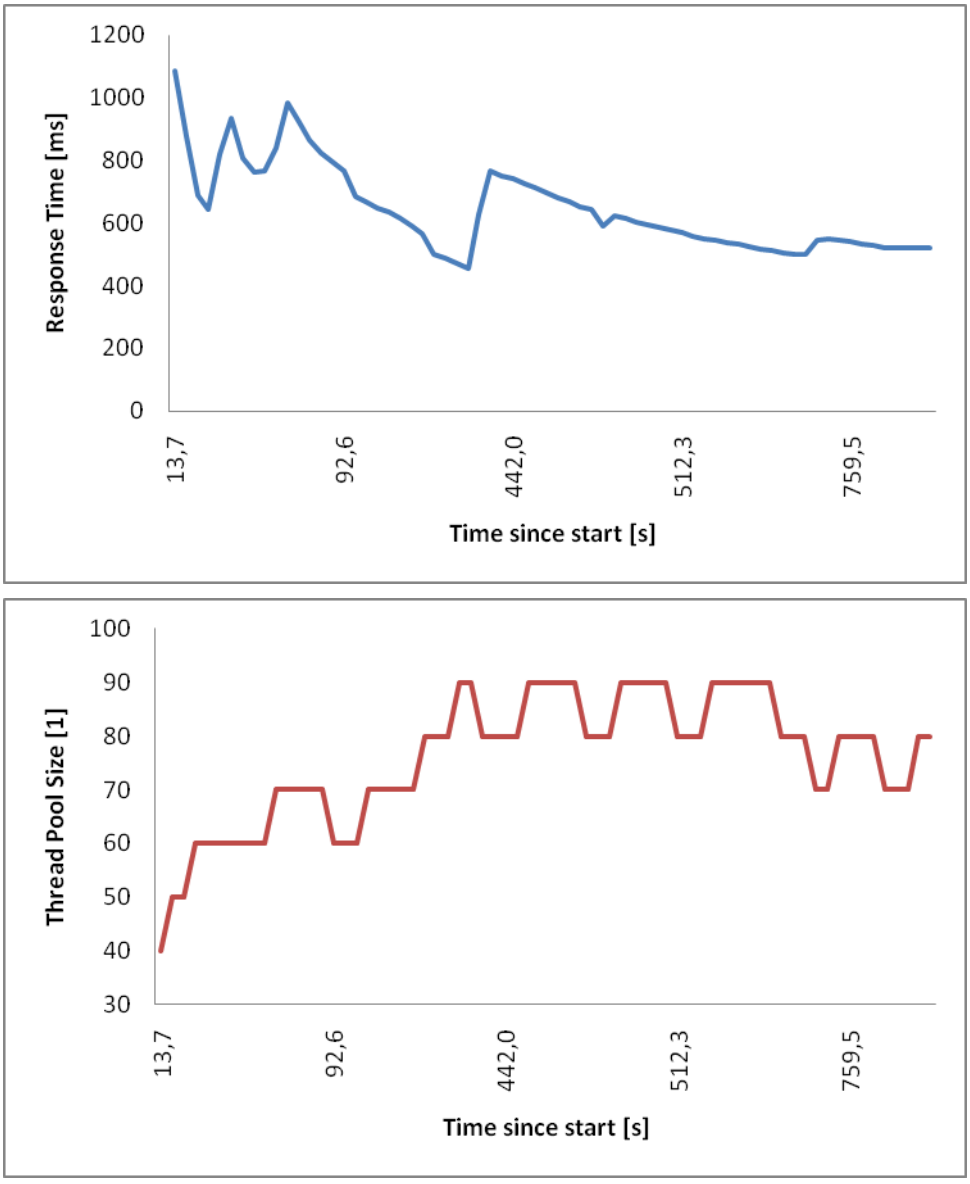

Figure 19: Testrun for Self-Optimization

The average response time increase from about 450 to almost 800 milliseconds, which can be seen in the middle of Figure 19, has a different cause. Since the maximum amount of memory that the JVM which runs JBoss is allowed to use has been constrained in this test run, the AM's self-configuration capability was triggered when the amount of available memory became low. As a

result, the EJB caches were cleared, which caused CPU load that deteriorated the response times measured.

It is because of these memory limitations that the AM finally settles for a TPS value of around 80 in this case. The constant up and down of TPS is a result of the optimization strategy. In this test, the AM was configured to use an up/down stepping size of 10, so this is the step size observed. The full list of parameters to the AM is finally visible in Table 5.

Table 5: Parameters to the Autonomic Manager

Parameter Name	Value	Description
ConfigureCYCLESECS	1.0	Interval at which the self-configuration MAPE cycle is executed (in seconds)
HealCYCLESECS	2.0	Interval at which the self-healing MAPE cycle is executed (in seconds)
OptimizeCYCLESECS	4.0	Interval at which the self-optimization MAPE cycle is executed (in seconds)
MAX_THREADS_TO_DB	3.0	Maximum ratio of threads in the pool to database connections
TPS_UPDATE_STEP	10	Value update for the optimization steps
TPS_MINIMUM	30	Absolute minimum value for thread pool size
TPS_MAXIMUM	200	Absolute maximum value for the threadpool size
TPS_UPDATE_MEMLOW	-30	Update to the thread pool size when a low-memory condition is encountered
TPS_UPDATE_DBPROBL	-10	Update to the thread pool size when a database problem is encountered
MEM_LOW	5242880	Warning threshold for low memory (in bytes)
CPU_EXCESS	0.9	Warning threshold for excessive CPU usage

4.4.3.5 Interactions among the self-management capabilities

The autonomic manager enhances the system with distinct capabilities, so the implementation of these capabilities was independently carried out and tested. But there is a certain amount of inter-action required among the capability implementations. The self-healing capability, for instance, will restart failed applications on the server. Because this creates load on JBoss, the response times measured during the restart will be degraded. The self-optimization implementation consequently has to be aware of the activity of the other self-management features. This also applies to the self-configuration capability, because the flushing of EJB caches will eventually lead to the invocation of the garbage collector, again putting additional load on JBoss.

In the scenarios presented in the case study, the autonomic manager in its self-optimization effort needs to disregard values that could be tainted by its other self-management capabilities. For a small scale scenario like the one discussed, the successful operation of the autonomic capabilities is not foiled even if there is no handling of these interactions. For larger scale applications of autonomic management, such inter-dependencies need to be thoroughly investigated.

5 Conclusion and Outlook

Autonomic computing presents a challenging vision of future computing environments. It shifts a lot of administrative work — which is now carried out by IT technicians and the people who put the systems into operation — from the customer who employs the system to the engineer that designs and develops the system. In this thesis, autonomic computing functionality was implemented for the JBoss application server. The principal AC architecture, which separates actual system functionality from the management-related tasks, was adhered to. This was done by defining the interface between the two parts as a web service based on open standards.

The chosen approach showed to be advantageous in the following ways:

- It made implementation of the server-side supportive functionality easier, because it was completely de-coupled from any AC-governed operation.
- It makes the server-side components that have been developed in this work reusable for other management applications.
- The standardized interface allows other management client applications to be used for the administration of JBoss.
- As the web service interface is itself implemented as a stand-alone component, other such interfaces can similarly be developed to add support for other protocols.

Three autonomic computing capabilities were implemented in this work:

- Self-configuration
- Self-optimization
- Self-healing

During the tests carried out in the case study it also became apparent that the functionality that was realized evidently contributes to the dependability and availability of the application server deployment. Especially one aspect, namely the monitoring of memory usage which is part of the self-configuration capability, was simple, but effective in preventing system failure.

Even if the vision of autonomic computing presents many ideas that seem unfeasible, the grand goal is definitively worth being pursued. This is not only because some of them are not too hard to realize and have great advantages. It's also because of one problem that became apparent during implementation: Expert knowledge about both the system's architecture and the relations between the many different parameters and their effects is needed in order to be able to effectively achieve management goals. As this knowledge is hard to obtain for the system administrator but directly accessible to the system's engineer, it seems just logical to make the achievement of such basic management goals a responsibility of the development effort.

Furthermore, it became clear that the different aspects of autonomic computing (expressed in the capabilities) should not only be investigated independently, but also their correlations and mutual influences have to be taken into account.

The autonomic computing functionality developed in the course of this thesis can be deployed similarly to the current human management, because the manager can reside on any machine that has network connectivity to the system-to-be-managed. Due to this, the transition from purely-human managed systems to partly-autonomous systems should be easier, because the human administrator has the autonomic manager nearby and can control and supervise its behavior.

Another approach for facilitating the transition to fully autonomic computing systems is presented in [AKF07], which models the communication between the managed system and its (human or autonomic) manager according to human speech theory. The transition is alleviated in this case, because the kind of communication partner to the managed system — human or machine — does not affect the communication protocol.

Future work can focus on integrating the aforementioned communication semantics into the autonomic computing case study presented. It will then be possible to develop a testbed for the evaluation of transition towards self-managing systems based on the functionality developed with JBoss as the example system-to-be-managed.

List of Figures

Bibliography

[ABI05] Robert Adams, Paul Brett, Subu Iyer, Dejan Milojicic, Sandro Rafaeli and Vanish Talwar. "Scalable Management," *Proc. 2nd Intl.Conf. on Autonomic Computing (ICAC'05),* 2005.

[AKF07] Edin Arnautovic, Hermann Kaindl, Jürgen Falb, Roman Popp and Alexander Szép. "Gradual Transition towards Autonomic Software Systems based on High-level Communication Specification," *Proc. Symposium Applied Computing (SAC'07), Autonomic Computing Track,* 2007. pp. 84-89.

[Ant] The Apache Software Foundation. "Apache Ant," http://ant.apache.org/.

[Apache] Apache Software Foundation, "The Apache HTTP Server Project," http://httpd.apache.org/.

[BMK04] Rob Barett, Paul P. Maglio, Eser Kandogan and John Bailey. "Usable Autonomic Computing Systems: the Administrator's Perspective," *Proc.Intl.Conf.on Autonomic Computing (ICAC'04),* 2004.

[CKZ03] George Candea, Emre Kiciman, Steve Zhang, Pedram Keyani and Armando Fox. "JAGR: An Autonomous Self-Recovering Application Server," *Proc. Autonomic Computing Workshop, 5th Intl. Workshopon Active Midleware Services (AMS'03),* 2003.

[Cla99] James Clark. "XML Path Language (XPath)," W3C recommendation, Nov 1999; http://www.w3.org/TR/xpath.

[ECperf] Sun Microsystems Inc. *ECperf(TM) Specification, Version 1.1 Final Release*, 2002.

[FR03] Marc Fleury and Francisco Reverbel. "The JBoss Extensible Server," *Middleware 2003 – ACM/IFIP/USENIX Int'l Middleware Conf.* 2003. pp. 344-373.

[FSE04] Giovanna Ferrari, Santosh Shrivastava and Paul Ezhilchelvan. "An Approach to Adaptive Performance Tuning of Application Servers," *Proc. 1st IEEE Int'l Workshop QoSAS'04,* 2004.

[FSR06] Marc Fleury, Scott Stark and Norman Richards. *JBoss 4.0 - Das offizielle Handbuch,* Addison-Wesley, 2006.

[HeH03] Hao He, "What is Service-Oriented Architecture," 30 Sep. 2003; http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html.

[HHP03] Alefiya Hussain, John Heidemann and Christos Papadopoulos. "A framework for classifying denial of service attacks," *Proc. of the 2003 conf. on Applications, technologies, architectures, and protocols for computer communications,* ACM Press, 2003. pp. 99-110.

[Hor01] Paul Horn. *Autonomic Computing: IBM's Perspective on the State of Information Technology,* Technical Report, IBM, 2001.

[ITS06] *IT Service Management Standards,* IBM White Paper, Apr 2006.

[J2EE] *JavaTM Platform, Enterprise Edition Specification,* Sun Microsystems, 2006. http://java.sun.com/javaee.

[Java6] Sun Microsystems, Inc. "Java Platform, Standard Edition 6 API Specification," 2006; http://java.sun.com/javase/6/docs/api/.

[JMeter] Apache Software Foundation, "Apache JMeter," 2006; http://jakarta.apache.org/jmeter/.

[JMX14] Sun Microsystems, Inc, "Java Management Extensions (JMX) Specification, version 1.4," 9 Nov. 2006; http://java.sun.com/javase/6/docs/technotes/guides/jmx/JMX_1_4_specification.pdf.

[Kai89] Hermann Kaindl, *Problemlösen durch heuristische Suche in der Artificial Intelligence,* Springer, Wien, 1989.

[KC03] Jeffrey O. Kephart and David M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, n. 1, 2003, pp. 41-45.

[Kep05] Jeffrey O. Kephart. "Research Challenges of Autonomic Computing," *Proc. 27th Int'l Conf. on Software Engineering (ICSE'05),* 2005. pp. 15-22.

[Kum05] Pankaj Kumar. "Web Services and IT Management," *Queue*, vol. 3, num. 6, 2005, pp. 44-49.

[LB98] Sheng Liang and Gilad Bracha. "Dynamic Class Loading in the Java Virtual Machine," *Proc. OOPSLA'98,* 1998.

[Lea00] Doug Lea. *Concurrent programming in Java : design principles and patterns,* Addison-Wesley, 2000.

[LHF05] Tiancheng Liu, Gang Huang, Gang Fan and Mei Hong. "The Coordinated Recovery of Data Service and Transaction Service in J2EE," *Proc. 29th Int'lConf. Conputer Software and Applications (COMPSAC'05),* 2005.

[Liv02] Dan Livingston. *Advanced SOAP fro Web Development,* Prentice Hall, 2002.

[LL02] Guy M. Lohman und Sam S. Lightstone. „SMART: Making DB2 (More) Autonomic,“ *Proc. 28th VLDB Conf.* 2002.

[Mic07] Microsoft Corporation. "Internet Information Services," 2007; http://www.microsoft.com/WindowsServer2003/iis/default.mspx.

[Mur04] Richard Murch, *Autonomic Computing*, IBM Press, 2004.

[Muse] The Apache Softare Foundation, "Apache <Web Services/> Project - Muse," 2007; http://ws.apache.org/muse/.

[MUWS11a] Vaughn Bullard and William Vambenepe, "Web Services Distributed Management: Management Using Web Services (MUWS 1.1) Part 1," OASIS Open, 2006; http://www.oasis-open.org/apps/org/workgroup/wsdm/download.php/20576/wsdm-muws1-1.1-spec-os-01.pdf.

[MUWS11b] Vaughn Bullard and William Vambenepe, "Web Service Distributed Management: Management Using Web Services (MUWS 1.1) Part 2," OASIS Open, 2006; http://www.oasis-open.org/apps/org/workgroup/wsdm/download.php/20575/wsdm-muws2-1.1-spec-os-01.pdf.

[MyS07] MySQL AB, "MySQL 5.1 Reference Manual," 2007; http://dev.mysql.com/doc/refman/5.1/en/index.html.

[NC407] Netcraft, "April 2007 Web Server Survey," 2 Apr. 2007; http://news.netcraft.com/archives/2007/04/index.html.

[Pat02] David A. Patterson. "A Simple Way to Estimate the Cost of Downtime," *Proc. 16th Sys.Admin Conf.* 2002. pp. 185-188.

[RFC1157] J. Case, M. Fedor, M. Schoffstall and J. Davin. "A Simple Network Management Protocol (SNMP)," IETF RFC 1157, 1990; www.rfc-editor.org/rfc/rfc1157.txt.

[RFC1441] J. Case, K. McCloghrie, M. Rose and S. Waldbusser, "Introduction to version 2 of the Internet-standard Network Management Framework," IETF RFC 1141, 1993; www.rfc-editor.org/rfc/rfc1141.txt.

[RFC1901] J. Case, K. McCloghrie, M. Rose and S. Waldbusser, "Introduction to Community-based SNMPv2," IETF RFC 1901, 1996; www.rfc-editor.org/rfc/rfc1901.txt.

[RFC2616] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, "Hypertext Transfer Protocol — HTTP/1.1," IETF RFC 2616, 1999; www.rfc-editor.org/rfc/rfc2616.txt.

[RFC3411] D. Harrington, R. Presuhn and B. Wijnen, "An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks," IETF RFC 3411, 2002; www.rfc-editor.org/rfc/rfc3411.txt.

[RHQ05] Chris Rupp, Jürgen Hahn, Stefan Queins, Mario Jeckle and Barbara Zengler, *UML 2 glasklar*, Hanser, München Wien, 2005.

[RRJ03] Mukund Raghavachari, Darrell Reimer and Robert D. Johnson. "The Deployer's Problem: Configuring Application Servers for Performance and Reliability," *Proc. 25th Int'l Conf. of Software Engineering*, 2003.

[SASE9] Sun Microsystems, Inc. *Sun Java System Application Server Platform Edition 9 Administration Guide (Beta)*, 2006.

[SOAP] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte and Dave Winer, "W3C Simple Object Access Protocol (SOAP) 1.1," 08 May 2000; http://www.w3.org/TR/2000/NOTE-SOAP-20000508.

[Ste05] Roy Sterritt. "Autonomic Computing," *Innovations in Systems and Software Engineering: A NASA Journal*, Apr. 2005. pp. 79-88.

[TM06] Vijay Tewari and Milan Milenkovic. "Standards for Autonomic Computing," *Intel Technology Journal*, vol. 10, 2006.

[Wal02] Aaron E. Walsh, *UDDI, SOAP and WSDL*, Prentice Hall, 2002.

[WSA] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Chamion, Chris Ferris and David Orchard, "Web Services Architecture," W3C Working Group Note, 11 Feb. 2004; http://www.w3.org/TR/ws-arch/.

[WSDL] Erik Christensen, Francisco Curbera, Greg Meredith and Sanjiva Weerawarana, "Web Services Description Language (WSDL) 1.1," W3C Working Group Note, 2001; http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[WSR12] Steve Graham, Anish Karmarkar, Jeff Mischkinsky, Ian Robinson and Igor Sedukhin, "Web Services Resource 1.2 (WS-Resource)," OASIS Open, Apr. 2006; http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-pr-02.pdf.

[WSRP12] Steve Graham and Jem Treadwell, "Web Services Resource Properties (WS-ResourceProperties) 1.2," OASIS Open, Apr. 2006; http:/docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-pr-02.pdf.

[XML] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler and Francois Yergeau, "Extensible Markup Language (XML) 1.0 (Fourth Edition)," World Wide Web Consortium (W3C) recommendation, 2006; http://www.w3.org/TR/2006/REC-xml-20060816.

[ZLQ03] Yan Zhang, Anna Liu and Wei Qu. "Comparing Industry Benchmarks for J2EE Application Server: IBM's Trade2 vs Sun's ECperf," *Proc. 5th Australian Coputer Science Conf. (ACSC2003)*, 2003.

[ZQL06] Yan Zhan, Wei Qu and Anna Liu. "Adaptive Self-Configuration Architecture for J2EE-based Middleware Systems," *Proc. 39th Hawaii Int'l Conf. on System Sciences*, 2006.