Monitoring and Managing Heterogeneous Middleware

Günther Rackl

Institut für Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation

Monitoring and Managing Heterogeneous Middleware

Günther Rackl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. H. M. Gerndt

Prüfer der Dissertation:

1. Univ.-Prof. Dr. A. Bode
2. Univ.-Prof. Dr. E. Jessen

Die Dissertation wurde am 13. Dezember 2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 29. Januar 2001 angenommen.

Abstract

The development and deployment of distributed applications still represents a major problem of computer science due to the complex and heterogeneous architectures of modern computing systems. The goal of this thesis is to improve the development and deployment of distributed applications by enhancing the on-line tool support for heterogeneous middleware environments. Our approach to reach this goal is to provide a monitoring and management system that is tailored to the needs of modern middleware applications.

Starting with an analysis of current middleware platforms and on-line tools for various application domains, the shortcomings of existing on-line monitoring and management approaches are worked out. Existing on-line tools mostly represent proprietary solutions for particular middleware platforms and are tailored to homogeneous scenarios, while their monolithic and static architecture limits their flexibility and extensibility.

From these insights, several requirements for the development of a monitoring and management infrastructure for heterogeneous middleware arise. Approaches for systematically managing complex environments as well as mechanisms for dealing with heterogeneity are essential in this context. Moreover, concepts supporting different kinds of on-line tools and enabling the extensibility of on-line tool environments are indispensable.

From these considerations, the MIMO MIddleware MOnitor approach is derived. MIMO is based on a multi-layer monitoring model that classifies observed entities by means of several abstraction layers, a generic monitoring infrastructure, and a sophisticated usage methodology. The prototypical implementation of the MIMO system is evaluated with respect to its applicability within current middleware platforms. Besides the observation of general purpose middleware platforms like CORBA, the application to metacomputing systems, a distributed interactive simulator, and a medical image reconstruction programme is demonstrated.

The benefits of the MIMO approach result from the systematic approach for monitoring heterogeneous environments, its capability to support tools for all on-line phases of the software lifecycle, and the extensible architecture of the monitoring system. Furthermore, the rapid tool development methodology allows to construct new tools or integrate new middleware platforms efficiently. Altogether, the results obtained within thesis represent a major contribution to an enhanced on-line tool support in heterogeneous middleware environments.

Für meine Eltern & für Beate.

Acknowledgements

This thesis would not have been possible without the support of several people. First of all, I would like to thank Prof. Arndt Bode for supporting me throughout the years and for providing an excellent research environment at the LRR. I enjoyed the great freedom to carry out my research and the opportunity to participate in various scientific conferences all over the world. In spite of his numerous obligations Prof. Bode always had time for a short discussion with me.

Prof. Eike Jessen deserves gratitude for being the co-referee of my thesis. From him I received helpful feedback in a constructive and unbureaucratic way.

The heads of the tools group at the LRR deserve thanks for encouraging me and my research within the past years. PD Dr. habil. Thomas Ludwig has contributed a lot to my scientific work and my personal development within the past years. Dr. Roland Wismüller has always been a good advisor regarding scientific and technical questions.

I would also like to thank my current and former colleagues in the tools group, Markus Lindermeier, Dr. Jörg Trinitis, Dr. Christian Röder, and Dr. Michael Oberhuber. Together, we had a lot of discussions concerning scientific as well as non-scientific topics.

My room mates Dr. Ivan Zoraja, Philipp Drum, and Martin Schulz as well as our system administrator Klaus Tilk deserve credit for innumerable technical, political, and philosophical conversations. Without them, my time at the LRR would not have been that exciting for me.

I am also grateful to Dr. Ivan Zoraja, PD Dr. habil. Thomas Ludwig, Dr. Roland Wismüller, Dr. Wlodzimierz Funika, Markus Lindermeier, Dr. Sabine Rathmayer, and Ulrich Rackl for thoroughly reviewing my manuscript, or parts of it.

Furthermore, this thesis has gained a lot from students working for the MIMO project. I would like to thank Michael Rudorfer, Alexi Stamatakis, Bernd Süß, and Lars Orta for contributing to the success of the project with their diploma theses.

Prof. Ian Foster from Argonne National Laboratory deserves gratitude for taking me up as a guest scientist in the Distributed Systems Laboratory at the ANL. Gregor von Laszewski, PhD, supported me immensly during my stay at Argonne. It was a great experience for me to participate in their research projects, and I personally and scientifically learned a lot during that time.

Last not least, I would like to thank my parents, my brothers, my love Beate, and my friends for supporting me with the great private background needed to finish such a thesis succesfully.

München, November 2000
Günther Rackl

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The spread of distributed computing systems has immensely increased over the past years. Distributed computing systems are becoming more and more important for mankind, in everyday life as well as in industrial and scientific domains. The Internet and its capabilities enable people to communicate and cooperate all over the world, while high-performance parallel computers or networks of workstations allow to solve scientific and commercial problems faster than ever.

However, development and deployment still represent a major problem of distributed applications due to their complex and heterogeneous architectures and communication mechanisms. Approaches to simplify the development and deployment processes are the usage of middleware platforms on the one hand, and the integration of on-line tools on the other hand. Middleware environments are intended to hide the heterogeneity of distributed systems and provide a platform for enabling the communication between distributed components. On-line tools, in contrast, are intended to assist developers and users of distributed environments by supporting the analysis or control of running applications.

The goal of this thesis is to contribute to an improved development and deployment of distributed applications. To reach this goal, we present an approach for the efficient construction of on-line tools for heterogeneous middleware environments. With our approach, we provide a monitoring infrastructure and a usage methodology that make it possible to efficiently and rapidly build on-line tools. Such tools can be used to support the complete "on-line lifecycle" of distributed applications, ranging from development tasks like debugging to subsequent deployment tasks like the management of running applications.

1.1 Motivation and Goals

With the advances of modern technology, the presence of computers in nearly every sector of life has drastically increased. Computers are becoming pervasive and ubiquitous, while we are moving on towards the "information society" [Eur2000a], [Eur2000b]. Advances in the field of interconnection networks and microprocessors have made distributed computing interesting for many purposes. With the increased computing and communication power, networks of workstations are becoming of interest for problem solutions formerly reserved to classical parallel computers. Another benefit of distributed computing systems is the permanent and location transparent accessibility of information, which allows users to cooperate or share

information all over the world.

From the structural point of view, there are several ways to organise distributed computing systems. An often used paradigm is the client/server principle, which distinguishes one of the participating computers as a server that provides pre-defined services to its clients. More recent approaches have abandoned the client/server principle as the server represents a central point of failure, which decreases the reliability and availability of the overall system. Instead, peer-to-peer computing systems like e.g. Gnutella [Weg2000] that do not distinguish any of the participating hosts represent a new trend for the development of distributed computing environments. Furthermore, new aspects of distributed computing have arisen with the availability of mobile devices that dynamically connect to networks via wireless communication.

Despite the technological advances and independent of the architecture of distributed systems, the development of software remains complex in comparison to software development for stand-alone systems. On the one hand, this complexity results from the diversity of components used for the construction of distributed systems, while on the other hand, complexity arises from the inherent distribution of components. Important issues in this context are the management of the available resources, the localisation of available functionality, the communication between distributed components, and the handling of failure situations.

Due to rapid technology transitions and constantly growing requirements for distributed applications, mechanisms for the efficient and reliable building of applications need to be found. A major contribution to the reduction of the development efforts is the usage of *middleware platforms*. Such platforms provide mechanisms to abstract from the underlying computing infrastructure and enable a more or less transparent interaction between application components. Nevertheless, despite the benefits of middleware platforms, further assistance is required in order to rapidly and efficiently develop and deploy distributed applications. For example, during software development, communication between components needs to be traced for debugging or performance analysis purposes. Or, during the subsequent software deployment, distributed applications need to be managed and failure situations have to be detected. The solution to these problems is the use of *on-line tools*, which are programs observing or controlling the behaviour of distributed applications at runtime.

For many middleware environments, various on-line tools exist. Such tools support development tasks like visualisation, debugging, or performance analysis, as well as deployment tasks like load balancing, computational steering, or application management. One of the main problems of on-line tools is their interference with the observed environment, as the observation itself influences the system under consideration and potentially affects its behaviour. This issue represents a general problem similar to Heisenberg's uncertainty principle [Hei1927] within the quantum theory, which has already been postulated in 1927. Thus, keeping the overhead of on-line tools as small as possible or controlling the influence on the observed system is an essential aspect for tool development.

When looking at currently available on-line tools supporting the development

or deployment of distributed applications, several decisive disadvantages can be found. Tools are mostly proprietary solutions that only work in homogeneous scenarios based on a single middleware platform. Furthermore, most tool approaches only concentrate on simple scenarios where certain aspects of a distributed application can be observed; a systematic concept for building different kinds of tools is missing. Also, tools are inflexible with respect to their extensibility or the interoperability with other tools connected to the same application.

Eliminating these drawbacks of existing on-line tools represents an important step towards improving the development and deployment process of distributed applications. Therefore, the goal of this thesis is to contribute to an enhanced on-line tool support for distributed middleware environments. This goal can be reached by designing and implementing an on-line tool concept that is tailored to the needs of modern middleware. In contrast to existing tools, our approach rests upon a systematic concept for dealing with complex and heterogeneous environments. The concepts are implemented by a generic monitoring infrastructure that serves as a basis for all kinds of tools. As a consequence, we are establishing a flexible and extensible tool environment that supports the complete on-line lifecycle of middleware applications. Our approach for monitoring and managing heterogeneous middleware therefore represents a sophisticated step towards a more efficient software development and deployment process for distributed applications.

1.2 Methodology and Outline

From the methodical point of view, this thesis shows the following structure: In a first step, we present and analyse current middleware platforms that are used for distributed applications. For these platforms, existing approaches for on-line monitoring, management, and tool construction are investigated. Our investigations yield a list of drawbacks of existing systems, which are then taken as a starting point for postulating a set of requirements for monitoring and managing heterogeneous middleware. These requirements serve as a basis for deriving a new approach for the construction of on-line tools in heterogeneous middleware environments. The derived approach represents an experimental hypothesis whose feasibility has to be proved. Therefore, we implement a prototypical monitoring system and tool environment that rests upon our approach and evaluate it by means of several real-world application scenarios. Finally, we reconsider our hypothesis and evaluate our overall on-line monitoring approach.

Consequently, the organisation of this thesis is as follows: Chapter 1 gives a general motivation of our work, presents the goals of our efforts, and summarises the research contribution.

In chapter 2, we will introduce the term “middleware” and position it within the field of distributed computing systems. A general definition of the notion of middleware will be worked out on the basis of existing points of view and further relevant criteria. Also, the terms heterogeneity and interoperability, which are key features of middleware systems, will be introduced. Subsequently, major examples for current middleware systems will be described. This is done by means of several

categories of middleware that are being used in different application domains.

Chapter 3 will introduce the research fields of on-line monitoring and management. We will define the terms “on-line tools”, “on-line monitoring” and “management”, which are essential for the remainder of the thesis. Furthermore, a general classification of tools, mechanisms for on-line monitoring, and problems arising with the monitoring process will be described. Following these general considerations, we will move on to the analysis of common approaches and systems in the field of on-line tools. As before, we will classify these approaches and systems on the basis of the application domains from which they have originated.

In chapter 4, we will work out the drawbacks of existing monitoring approaches and systems. From these drawbacks, a set of requirements for a generic monitoring approach will be derived. Here, the main goal of our requirements is to enable the construction of on-line tools for heterogeneous middleware environments with a single, powerful on-line monitoring approach.

Chapter 5 will introduce the MIMO MIddleware MOnitoring approach for constructing an on-line tool environment for heterogeneous middleware environments. MIMO represents a concrete approach to implement the abstract requirements stated in chapter 4. The overall MIMO approach is based on three fundamental ideas: First of all, an information model called the “multi-layer monitoring model” serves as a foundation for classifying entities within the observed systems. Secondly, we define a generic monitoring infrastructure that allows to build new tools or to integrate new middleware easily. And thirdly, we present a usage methodology and a tool framework describing the development of tools and the adaption of middleware platforms.

The concrete implementation of a MIMO prototype will be illustrated in chapter 6. After a short review of the tool development and usage process, the basic monitoring architecture of the MIMO prototype will be shown. The architectural point of view comprises the participating components, their interaction patterns, and further distribution and assignment considerations. Subsequently, the access and usage of the MIMO implementation will be presented. This description covers the interfaces needed to access the MIMO system, as well as procedures for deploying the monitoring system in practice. Implementation details regarding the MIMO core and its performance, instrumentation components for various middleware systems, and the MIVIS tool framework will be shown in the last section of chapter 6.

Procedures for the efficient usage of the MIMO system will be explained in chapter 7. For this purpose, MIMO’s tool development methodology establishes a rapid tool development process that can be applied to arbitrary monitoring scenarios. In order to demonstrate the applicability of the MIMO system, several real-world application scenarios for the MIMO approach and its tool development and middleware integration facilities will be described. In addition to general purpose middleware, these scenarios comprise the integration of metacomputing platforms, a distributed simulation environment, and a medical image reconstruction application.

In chapter 8, the overall MIMO approach, its implementation, and applicabil-

ity will be evaluated. Moreover, after a review of the initial problem, the results accomplished with our work will be summarised. Besides the major conceptional contributions, essential lessons learned during the development process will be elaborated.

Finally, chapter 9 contains a conclusion that summarises the major aspects of this thesis and provides an outlook to future topics.

1.3 Research Contribution

The research contribution of this thesis is to establish methods and techniques for monitoring and managing heterogeneous middleware. The resulting benefit of our work is to improve the on-line tool support in heterogeneous middleware environments. Our efforts therefore represent an essential contribution that enhances the development and deployment processes of distributed applications.

In the field of on-line tools, no other tool framework is able to fulfil the requirements of heterogeneous middleware systems. Existing on-line tools and monitoring systems are proprietary solutions tied to homogeneous middleware platforms, which cannot observe applications based on different middleware platforms simultaneously. Furthermore, existing tools or monitoring systems concentrate on particular phases of the software lifecycle and do not provide mechanisms for covering all on-line phases of application development and deployment. Extensibility with respect to the development of new tools or the integration of new middleware is hardly given because systematic usage procedures for the tool environments are missing.

Our work eliminates the drawbacks of existing approaches and provides a solution to the particular problems in heterogeneous middleware environments. The key contributions of our approach are as follows:

- Firstly, we define an abstract information model that makes it possible to classify entities existing within the various observed middleware platforms. In combination with the procedure for mapping concrete middleware platforms and the efficient algorithms to access the resulting data structures, the *multi-layer monitoring model* therefore represents a basis for the systematic monitoring of complex middleware environments.

- Secondly, we implement a *generic monitoring infrastructure* that is open to all kinds of tools and middleware. This infrastructure provides intelligent procedures for starting monitoring sessions and basic services for discovering the observed entities and their properties. Beyond this, all interfaces are kept generic, such that new middleware can be attached easily and new tools can be developed smoothly. Altogether, our monitoring infrastructure establishes a light-weight approach towards enabling communication between instrumented applications and tools on the basis of the multi-layer monitoring model. As a consequence, our infrastructure fulfils the requirement for a systematic monitoring approach that is able to support the complete on-line lifecycle and that allows to simultaneously observe heterogeneous platforms.

The applicability of our system thus ranges from development tasks like debugging to deployment tasks like application management.

- Thirdly, we propose a methodology for the efficient use of our monitoring infrastructure. Our usage methodology serves as a guideline for the systematic development of new tools or the integrating of new middleware. This is of great importance because the monitoring infrastructure provides a high degree of freedom to users due to its generic layout. Following the steps of our methodology therefore enables a *rapid tool development* process, which allows to rapidly construct on-line tools or to integrate new middleware, while an incremental extension is made possible in order to adapt to changing requirements. Thus, our methodology fulfils the requirements for a flexible and extensible monitoring approach and contributes to the coverage of heterogeneity due to its mapping strategies for concrete middleware platforms.

In addition to these major contributions, the resulting monitoring approach points out further aspects that are relevant for the on-line monitoring process. The monitoring infrastructure serves as a kind of intelligent event propagation framework, which itself can be interpreted as a kind of special purpose middleware. Dynamic processes within modern middleware applications are an important aspect that has to be taken into account. This concerns the software lifecycle of distributed applications as well as the highly dynamic behaviour of application components. Moreover, the integrative aspect of building on-line tools for heterogeneous middleware is a relevant point in this thesis. Observing the diverse platforms and applications with a single tool environment is a major contribution that significantly simplifies monitoring and management tasks.

The scientific innovation of this thesis is the concept of building a universal environment for the construction of on-line tools. While existing systems are limited either to specific middleware platforms or to specific kinds of tools, we propose an approach that integrates heterogeneous platforms and makes it possible to support the complete on-line software lifecycle. As the resulting monitoring system is complex, a usage methodology serves as a guideline for users and enables a rapid tool development process.

At a first glance, the scope of our work comprises all kinds of software tools that are used in heterogeneous environments. However, as our monitoring system itself can be seen as a middleware platform the results also affect other middleware-based distributed systems. Both the integrative aspects of our monitoring approach and the mechanisms for dealing with the dynamic behaviour of modern middleware applications can be transferred to other domains. The main insight in our context is that only a generic and universal approach can fulfil the requirements within the investigated middleware environments.

Finally, the technical and economical contribution of this thesis has to be emphasised. Many current and future information systems have and will have to solve problems concerning the integration of heterogeneous components, the organisation of dynamically changing environments, or the methodologies needed for utilising

such systems. Our monitoring system may therefore serve as a foundation for constructing the next generation middleware.

1.4 Background

The results accomplished within this thesis are based on numerous research activities in the field of on-line monitoring at the Lehrstuhl für Rechnertechnik und Rechnerorganisation at Technische Universität München (LRR-TUM). During the past years, monitoring technologies for parallel and distributed systems have been investigated and developed [BKLW2000].

The TOPSYS project [BBB+1991] aimed at the design and development of a hardware and software monitoring system running on iPSC/2 and iPSC/860 parallel computers. Later, new concepts for the construction of monitoring systems were worked out. The objective of the On-line Monitoring Interface Specification (OMIS) [LWSB1997] was to define a versatile interface between tools and the monitoring system. An implementation of the OMIS specification has been realised with the OMIS Compliant Monitor (OCM) [WTL1998], which now provides the basis for the integration of several tools in the framework of the TOOL-SET project [WLB+1997]. In this context, topics like tool interoperability [Tri1999], load management [Röd1998], aspects of non-determinism [Obe1998], or computational steering [Rat2000] have been investigated.

The insights gained in these research projects serve as a foundation for this thesis, which focuses on monitoring and managing heterogeneous middleware platforms.

Chapter 2

Middleware

The term “middleware” describes the key approach for developing distributed applications within this thesis. Therefore, this chapter gives a general definition of the term and introduces the field of middleware by its history, characteristics, examples of current systems, and possible future trends.

By looking up “middleware” in standard encyclopedias or computer science dictionaries, it is obvious that the term is relatively new as it can hardly be found in any encyclopedia, and only in most recent computer science publications. The Encyclopaedia Britannica [Bri2000], for example, has no entry for “middleware”, and several computer science dictionaries do not ([Inf1993]) or only briefly ([RP1999], p. 654) offer “middleware” as a catchword.

On the other hand, running this term through Internet search engines yields a high number of hits for this term. AltaVista [Alt2000], for example, found 243.720 pages for the search term “middleware” on Feb 7, 2000. These facts show that middleware represents an important, but still emerging research topic within computer science, which leaves a lot of challenging questions to deal with.

2.1 General Definitions

This section gives a general introduction into the field of middleware systems based on its historic development, and defines relevant terms in the research field of middleware.

2.1.1 Client/Server Systems

With the rise of distributed systems in the past, efforts to standardise all aspects of distributed computing, from the physical layer up to the application layer, have been made. But, as those standardisation efforts have shown to be very difficult to enforce, the class of *client/server* systems came up in order to deal with the still existing heterogeneity within distributed applications. With the client/server model, applications are split into tasks that get distributed among a set of nodes communicating via the network. Every component of the applications can then be executed on the node where it can be handled most efficiently. There are several similar definitions for the client/server computing model (see [Sta1998], p. 555 for examples); we will use the following one:

Client/Server Model: A computing model that is based on splitting applications into *clients* and *servers*. Thereby, servers provide functionality by means of *services*, which represent different kinds of available resources. Clients make use of these services by submitting *requests* to the servers.

This definition does not require client and server to be distributed onto different nodes, as it may in practice be useful to have client and server collocated on one machine for technical reasons. Nevertheless, the general case is surely the distribution of clients and servers among several nodes. Figure 2.1 illustrates the basic client/server computing model.

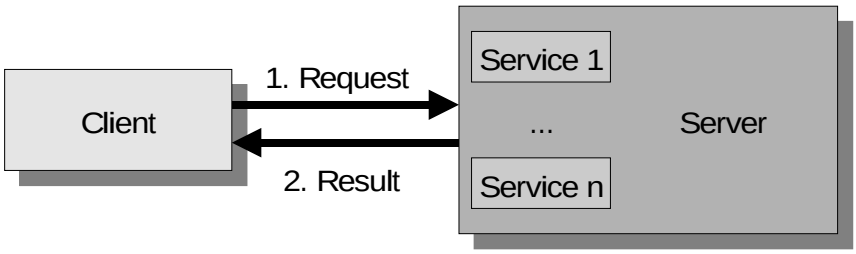

Figure 2.1 Client/Server Computing Model

2.1.2 The Gartner Model

When building client/server systems, there is great flexibility in the degree of distribution of application components. Traditional distributed applications can be separated into the three main components presentation, business logic, and data management. Depending on the distribution of these components, the *Gartner Model* ([LD1998], p. 23f) classifies distributed applications as shown in figure 2.2. Starting from the *distributed interface*, which merely adds a new graphical user interface (GUI) to an existing legacy application, several classes of distributed applications are defined. The *distributed logic* represents the classic notion of a distributed application because of its distributed business logic, which is the core part of an application. The last stage within the Gartner Model is *distributed data management*, where even the application's data bases are distributed.

2.1.3 Three-Tier Applications

The Gartner Model is a two-tier model, as it always consists of two tiers representing the client- and server-side. However, many modern applications cannot be classified with the Gartner Model anymore as they break up the three main components presentation, business logic, and data management into three separate distribution units, which yields a so called *three-tier architecture*. Thus, with this model, a new layer modelling the business processes is introduced between the desktop clients and the database servers. The main reason for this separated application-logic component is scalability within high-performing enterprise computing systems, and the

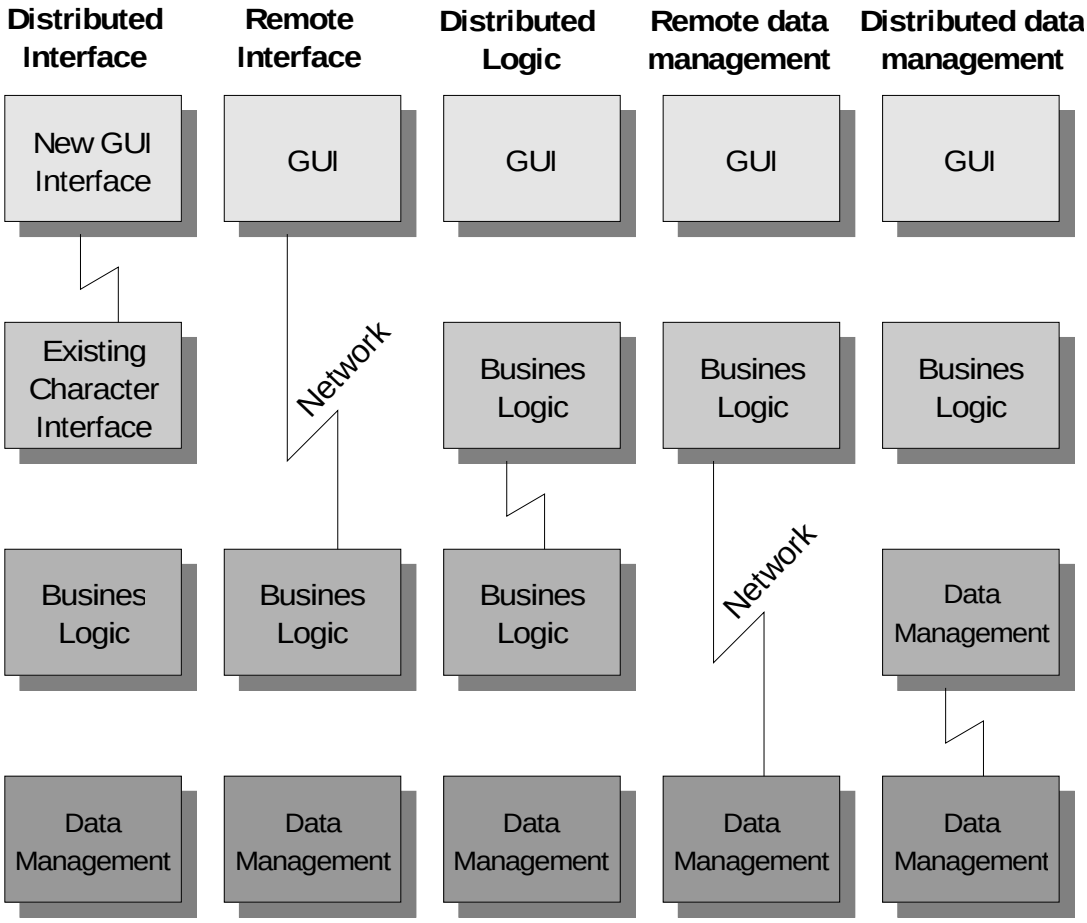

Figure 2.2 Gartner Group Distributed Application Model

encapsulation of business logic in an own component that makes it easier to handle complex procedures.

2.1.4 Middleware

To give a general definition of the term middleware, it is at first helpful to look at previous definitions made in different software development contexts:

- Originally, the term “middleware” is first mentioned in distributed operating systems literature, defining it as “standard programming interfaces and protocols that sit between the application above and communications software and operating system below” ([Sta1998], p. 563).

- In the field of client/server systems, middleware can be characterised as the “slash (/) in client/server” ([LD1998], p. 468). This expression already proposes middleware to be responsible for the interaction between client and server.

- Software engineering concepts for component-based applications define middleware as mediating software that allows to find, bind, and use services provided by components within distributed applications ([RP1999], p. 654).

Based on the general Gartner Model, middleware systems can be divided into three categories:

- *Presentation middleware* only cares for displaying data remotely. A Web browser and server communicating via the HTTP protocol (hypertext transfer protocol) can for example be classified into this category.

- *Application middleware* is used to distribute the application logic, and therefore functions as a general purpose programming platform for distributed applications. Its goal is to enable application programmers to build interacting components using middleware to abstract from given system details.

- *Database middleware* is deployed to access database management systems remotely. For example, SQL requests being sent to the DBMS and transferring back the results to the client are a typical task for database middleware.

Throughout this thesis, application middleware will be the focus of interest, although all the concepts and implementations can easily be applied to the other categories as well. Therefore, with regard to former definitions and this classification of middleware systems, the following general definition of middleware can be given:

Middleware: A software layer between operating platform and application that enables the interaction of potentially distributed application components, reaching a certain degree of transparency and independence from the surrounding runtime environments.

Thereby, transparency from the operating platform covers the network, hardware, and operating system. The kinds of transparency for the application in most cases comprise programming language, access and location transparency. This allows components implemented in different programming languages to interact without knowledge of their actual location (execution node), and syntactically equivalent to local interactions.

The implementation of middleware relies on offering an application programming interface (API) to the application components, and by using the operating platform's underlying interfaces. Figure 2.3 shows an overview of the middleware layer within the system model.

2.1.5 Heterogeneity and Interoperability

Another important aspect of many middleware systems is interoperability, which describes the concept of heterogeneous components interacting with each other. *Heterogeneity* in this sense covers a large variety of features, concerning either

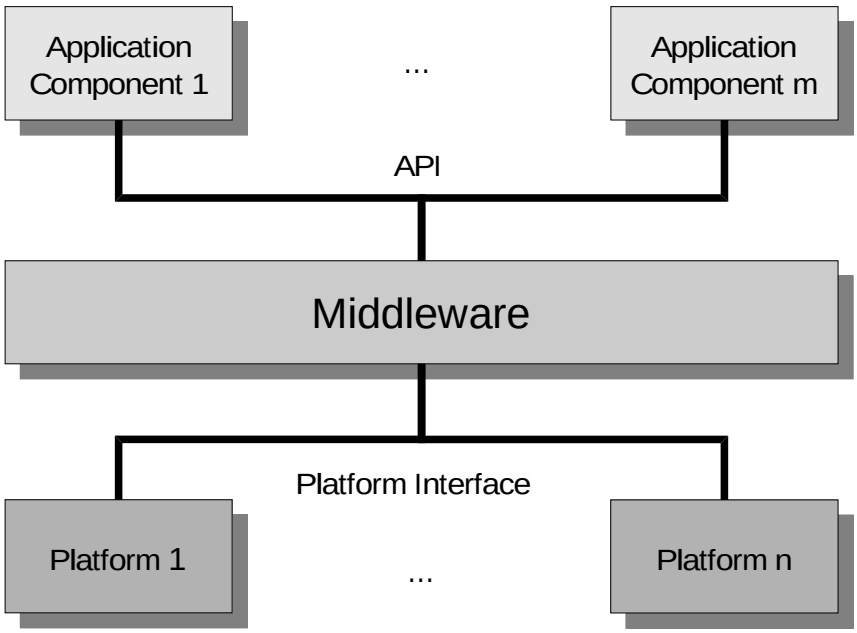

Figure 2.3 Middleware Layer between Applications and Platforms

computing platforms or application characteristics. Features regarding the computing platforms include communication networks, hardware, and operating systems; on the application side, heterogeneity mostly results from the use of different programming languages for the implementation of diverse components.

Interoperability may cover only the interaction between components using different implementations of a single middleware platform ([OMG1998c], p. 10-1ff), or it can be seen as an approach to enable different middleware platforms to interact with each other. Hence, we can define the term interoperability generally as follows:

Interoperability: The ability of components of distributed applications to interact with each other in a heterogeneous environment.

This definition covers both kinds of interoperability mentioned above. Furthermore, the notion of interoperability is also used in the area of tools ([Tri1999], p. 51), describing tools that are able to interact with each other; however, this notion is not the target of our interoperability definition.

2.2 Common Middleware Systems

In this section, we present several common application middleware systems that are either of historical, technical, or industrial relevance, and conclude with current trends and potential future developments. As the market of available middleware systems is getting more and more complex, a number of studies classifying and evaluating middleware concepts and products has been published ([Hei1999], [RE1997], [Kor1997]). These studies show that middleware development has become very diverse, and either has led to general purpose platforms emphasising the

integrative aspects of middleware, or to specialised products adapted to concrete usage scenarios. The following sections give an overview of the variety and characteristics of relevant systems.

2.2.1 Historically Relevant Approaches

The two fundamental approaches most modern middleware systems rely on are the Remote Procedure Call and the Distributed Computing Environment.

2.2.1.1 Remote Procedure Call

Remote Procedure Call (RPC) introduces the concept of calling a procedure remotely in a syntactically identical way to local procedure calls. Besides the unique calling mechanism, a main advantage of this mechanism is easy parameter handling that allows for static type checking at compile time, a feature that is not given with pure socket communication. The drawback of RPC is that despite the syntactical identity with local procedure calls the semantics is not identical. The reason for the different semantics results from the different address spaces, runtime environments, and potentially different lifetimes of the communicating processes. For example, it is not possible to pass pointers as parameters as in local C procedure calls because of the separate address spaces.

The implementation of RPC is based on a description of the interface with the RPC language, from which stubs and skeletons for the client- and server-side are generated with an RPC compiler. These stubs and skeletons take over the task of marshalling and unmarshalling parameters on both sides, and care for the communication that relies on the TCP/IP or UDP protocols. Hence, with RPC, a certain degree of access and location transparency can be reached.

RPC was the first system with these properties, and therefore represents a foundation for most subsequent middleware platforms. It was first implemented in 1984 [BN1984], but the most popular implementation is Sun RPC, which was delivered in 1985 as part of Sun's *Open Network Computing* (ONC). In addition to the RPC module, ONC contains a library for external data representation (XDR) that provides functionality for platform independent data encoding. One of the main applications of RPC is Sun's Network File System (NFS). Today, RPC and XDR are standardised as the Internet Engineering Task Force (IETF) standards RFC 1831 [Sun1995a] and RFC 1832 [Sun1995b]. For more details on RPC, see e.g. [LS1994].

2.2.1.2 Distributed Computing Environment

The Distributed Computing Environment (DCE) is a development of the Open Group [Gro2000][1]. Its purpose is to provide a distribution platform for client/server applications. DCE defines an architecture model for applications that is placed on top of the local commodity operating systems. It defines a set of *services* that are

[1] Formerly *Open Software Foundation* (OSF)

hierarchically structured, so that higher level services can always use lower level services.

The lowest-level service on top of the local operating system is the *Thread Service*; it is based on the POSIX standard 1003.1c [NBF1996], and defines an API for lightweight processes. This API contains functions for creating, manipulating, synchronising, and deleting threads, and serves as a basis for asynchronous communication. The basic communication service placed above the Thread Service is the Remote Procedure Call described in the previous section. On top of RPC, several higher-level services like the *Time Service*, *Directory Service*, *Security Service*, and *Distributed File Service* are available. These services provide functionality for different kinds of general purpose tasks required in most distributed applications. DCE environments themselves are divided into so called *cells*, which represent administrative units comprising users, machines, or applications in any arbitrary configuration.

The main drawbacks of DCE are its reliance on RPC as the only communication mechanism, and its non-object-oriented design (although there are object-oriented extension to DCE). Nevertheless, an important contribution of DCE is its concept of splitting middleware functionality into a set of variably usable services. Moreover, the decentralised and therefore scalable approach of building organisational cells is of importance for subsequent developments. These concepts can be found in many modern middleware platforms that will be discussed in the following sections.

2.2.2 Parallel Programming Environments

Another important, but very different form of middleware are parallel programming environments used for *high-performance computing and networking* (HPCN). The main goal of these platforms is an efficient communication between parallel processes, while transparency and ease of use originally play a secondary role.

First implementations of parallel programming environments are based on the *message-passing* paradigm, an approach with which all communication is based on exchanging messages. The only fundamental communication operations are the sending and receiving of messages.

Later systems pick up the *distributed shared-memory* (DSM) paradigm, which provides an illusion of shared memory within a distributed environment. DSM systems provide the advantage of an improved location transparency and are therefore easier to program, but have problems to reach the efficiency of message-passing systems.

2.2.2.1 Message-Passing Systems

The two most popular implementations of message-passing systems are Parallel Virtual Machine and the Message-Passing Interface.

Parallel Virtual Machine. Until recently, *Parallel Virtual Machine* (PVM) [Sun1990] [GBD+1994] was the de-facto standard for general purpose message-passing libraries. It can be deployed on both dedicated parallel computers and heterogeneous workstation clusters.

PVM consists of a programming library supporting different kinds of functionality: Process control functions create new application processes (called *PVM tasks*) within the distributed system, interprocess communication functions allow processes to interact by exchanging messages, and environment control functions allow to configure the environment, e.g. by defining the set of possible execution nodes. The execution of parallel processes is controlled by *PVM daemons* running on each participating node, and all machines running a PVM daemon constitute the virtual parallel machine; communication within PVM on workstations is implemented using both UDP and TCP.

The advantages of PVM are its portability and its applicability in heterogeneous environments. Further benefits are the abstraction of providing one single virtual machine composed of several nodes, and the possibility to dynamically manage computing nodes and processes.

Message Passing Interface. In contrast to PVM, which provides an implementation of a message-passing library, the *Message-Passing Interface* (MPI) [SOHL+1999] only defines a standard message-passing interface that can be implemented by vendors of high-performance computers or clusters of workstations; this gives vendors the opportunity to build MPI implementations with special optimisations for their hardware and communication infrastructure. Besides various messaging and process management functionality, MPI allows to define message contexts and communication topologies. Currently, MPI is established to be the new standard among message-passing platforms.

2.2.2.2 Software Distributed Shared-Memory Systems

Message-passing systems have traditionally been deployed on *loosely-coupled* hardware, as for processes with disjoint address spaces exchanging messages is the simplest way to communicate. On *tightly-coupled* hardware, in contrast, *shared-memory* programming is enabled because memory is equally accessible to all participating processors. For the programmer, shared-memory programming is easier because no explicit data movement operations are required, and synchronisations are carried out by the runtime system to arbitrate conflicting memory accesses.

The *Distributed Shared-Memory* (DSM) model now tries to apply the shared-memory programming model on loosely-coupled hardware, combining the advantages of both systems: Loosely-coupled hardware is more scalable, easier to build, and inexpensive, and with the shared-memory abstraction, a more beneficial programming model is given.

In *Software DSM systems*, the DSM abstraction is provided by an additional DSM middleware layer that provides a DSM API for the programmer, and maps these operations to communication and synchronisation operations on the underlying message-passing environment. Typical software DSM implementations are TreadMarks [ACD+1996] and Orca [BBH+1998]. The main issue of most software DSM implementations is performance: Programming on the higher-level shared-memory abstraction reduces the programmer's influence on optimising communication, and hence, a trade-off between efficiency and simplicity has to be found. For

more details on Software DSM systems, see [ZRS+2000], [Zor2000].

2.2.2.3 Summary

Both message-passing and software DSM systems are initially developed for high-performance computing. Efficient execution and communication on heterogeneous hardware is the main goal, while interoperability and portability have to be maintained. Therefore, parallel programming environments can be classified as low-level middleware, providing only a low degree of transparency regarding the interaction of distributed components. This holds true particularly for message-passing systems, but also in the case of DSM systems transparency is only reached at a rather low level of memory access. Higher-level services, for example for dynamically looking up distributed components, are missing.

2.2.3 Metacomputing Infrastructures

Metacomputing infrastructures extend the parallel high-performance computing paradigm to geographically distributed resources. This is either necessary when local resources are no more sufficient for solving hard problems, or when the distribution of resources is inherent, for example in multidisciplinary collaboration environments. The resulting infrastructures are often referred to as *grids* [LK1999] [FK1999], which represent higher-latency and lower-bandwidth wide-area interconnections.

The goal of metacomputing systems is not to replace parallel computers, but to extend the parallel programming model for usage within geographically distributed environments. To achieve this, they provide services supporting specific requirements of metacomputing, including e.g. resource management, security protocols, information services, fault tolerance mechanisms, or communication services. These services are adapted to common problems of distributed resource usage, which include authentication and authorisation problems due to different administrative domains, resource management and allocation problems in large environments, or communication problems arising from different latency, bandwidth, and reliability within the grid.

Thus, metacomputing infrastructures lift parallel high-performance computing to a higher level of abstraction by adding services targeted at geographically distributed systems.

Globus. Globus [FK1998] [Glo2000] represents an advanced implementation of a metacomputing infrastructure. The *Globus toolkit* provides basic services and consists of several modules. These include functionality for resource management (Globus resource allocation manager GRAM), communication (Nexus), security (Globus Security Service GSS), information (Metacomputing Directory Service MDS), fault tolerance, and remote data access (Globus Access to Secondary Storage GASS). The services are independent of each other and can be used in any arbitrary combination.

The design of the Globus toolkit reflects the main problems of metacomputing environments:

1. Autonomous institutions: Resources are managed by different institutions that reside in different administrative domains.

2. Resources are managed using different resource management systems.

3. In most cases, applications require different distributed resources to be allocated at the same time (resource co-allocation).

4. It has to be possible to monitor the progress of applications at runtime in order to be able to react on changing resource requirements.

Globus deals with these problems using GSS for authentication and authorisation in different domains, whereas GRAM allows to use different resource management systems and add new ones, and MDS enables on-line monitoring.

Summing up, metacomputing infrastructures represent middleware for high-performance computing that extend parallel programming environments for geographically distributed computing by enriching it with a set of services particularly targeted at computational grids.

2.2.4 Distributed Object-Oriented Systems

This section gives a survey of the class of object-oriented middleware. After starting with the general principles of this class, three major representatives for object-oriented middleware will be described and compared in more detail.

2.2.4.1 General Principles

Generally, Distributed Object Computing (DOC) [RZB1999] integrates object orientation, client/server architecture, and distributed computing. Applications are decomposed into a set of objects, either providing server functionality or acting as clients. These objects interact by invoking methods on requested target objects, identical to the traditional paradigm of object-orientation.

Interfaces. In order to make the functionality offered by server objects understandable and available to other objects, DOC systems describe services by defining their *interfaces* using an interface definition language (IDL). This IDL description lists all operations that are available for a certain type of service, and serves as a basis for the generation of client and server stubs for the client and server object implementations. This generation is performed by an IDL compiler that maps the IDL description to various programming languages, in which the actual objects are implemented. IDLs are declarative languages similar to C++ class declarations and Java interfaces and describe attributes and methods available in server objects.

Broker Mechanism. DOC systems make use of *globally unique object references* to identify objects within a shared global object space uniquely and issue method calls upon the reference. The component that is responsible for the creation of unique object references and delivering method calls to appropriate objects is called the *broker*. Brokers are logically centralised components that are able to resolve object references. Thus, client objects are not required to know the actual location

of the server object they are addressing with an object reference, so that the location transparency is managed by brokers. Brokers are implemented according to the *broker design pattern* [BMR+1996] that is depicted in figure 2.4. The broker design pattern encompasses the client, server and broker classes and their proxies. When transferring messages between different platforms additional bridges may be used to carry out protocol conversions.

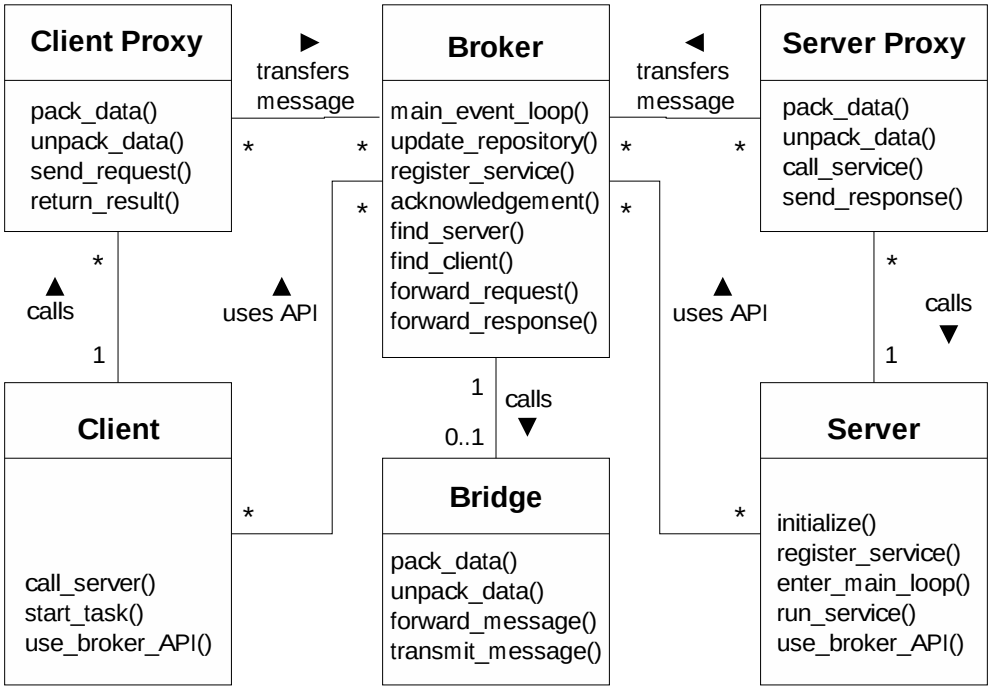

Figure 2.4 Broker Pattern

Technically, when a method call upon a given object reference is issued, the DOC system has to detect whether the call is local or remote. In the remote case, parameters need to be marshalled, a message to the remote object has to be sent, and when the method call returns the results have to be demarshalled. Incoming requests on the server side are processed using an analogous proceeding. In most cases, this functionality is realised using *proxy objects*, which are local representatives of the server objects on the client side and vice versa. These proxies carry out the respective actions and enable the access transparency for local and remote method calls. Therefore, the combination of the broker and proxy mechanisms can be seen as the basic communication mechanism in DOC systems.

Figure 2.5 shows the structure of the *proxy design pattern* [GHJV1995]. A particular service is implemented by the original object. As clients can not access the original directly – it may for example reside in a different address space – they make use of proxies that provide the same interface as the originals do. To achieve the proxy functionality, both the original and the proxy are derived from an abstract original that only specifies the required interface. The proxy keeps a reference to the original object and is able to forward client requests to it and deliver the results

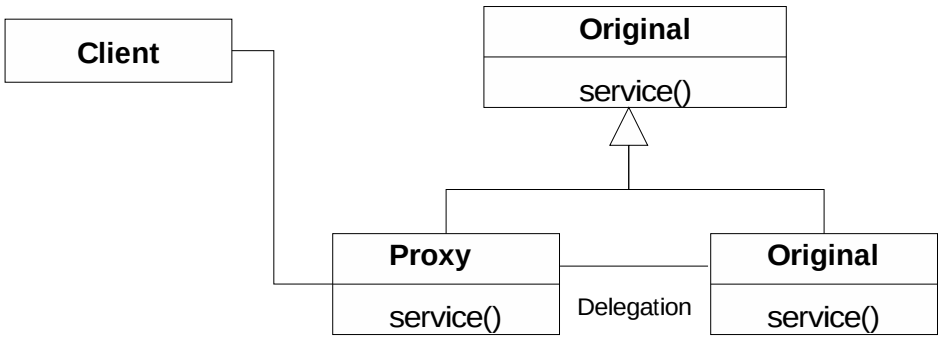

Figure 2.5 Proxy Pattern

back to the client. In addition, proxies take over the task of packing and unpacking data and the communication with the broker.

Finding and Accessing Objects. One of the basic conceptual questions in DOC is how clients can obtain object references to access requested services. To solve this problem, most DOC systems implement some kind of *naming service* at which server objects can register themselves by giving their name and their object reference. The scenario for obtaining object references of a remote object using a naming service is as follows: When a server is started it creates its objects providing given service types and registers those objects at the naming service. Clients contact the naming service, give a service name and obtain the object reference to the requested service. Once the reference has been obtained, the client may invoke methods that implement the requested services.

When a client has obtained an object reference from an appropriate server object it can issue method calls on it. The steps carried out when a method on a given target object reference is invoked is shown in the UML collaboration diagram in figure 2.6. First, the client calls the method at the client proxy, which represents the local representative of the server object. The client proxy resides in the same address space as the client itself and provides an interface identical to the server object's interface. The proxy takes over the task of marshalling data and passing the resulting request to the broker. The broker can logically be seen as a single component responsible for delivering the request to the server object represented by the given object reference.

On the server side, the server proxy unmarshalls the request and delivers it to the actual server object that carries out the method call and passes the results back to the server proxy; the way back to the client is analogous to the call. The illustrated scenario shows a common solution in which the broker is physically distributed among the concerned nodes and where no additional bridges are involved. In practice, the internal implementation of the broker is left to the developer and should not influence the programming model. In large client/server environments, clients and servers may be developed and compiled independently. The interface of a service is the only common information clients and servers need to share.

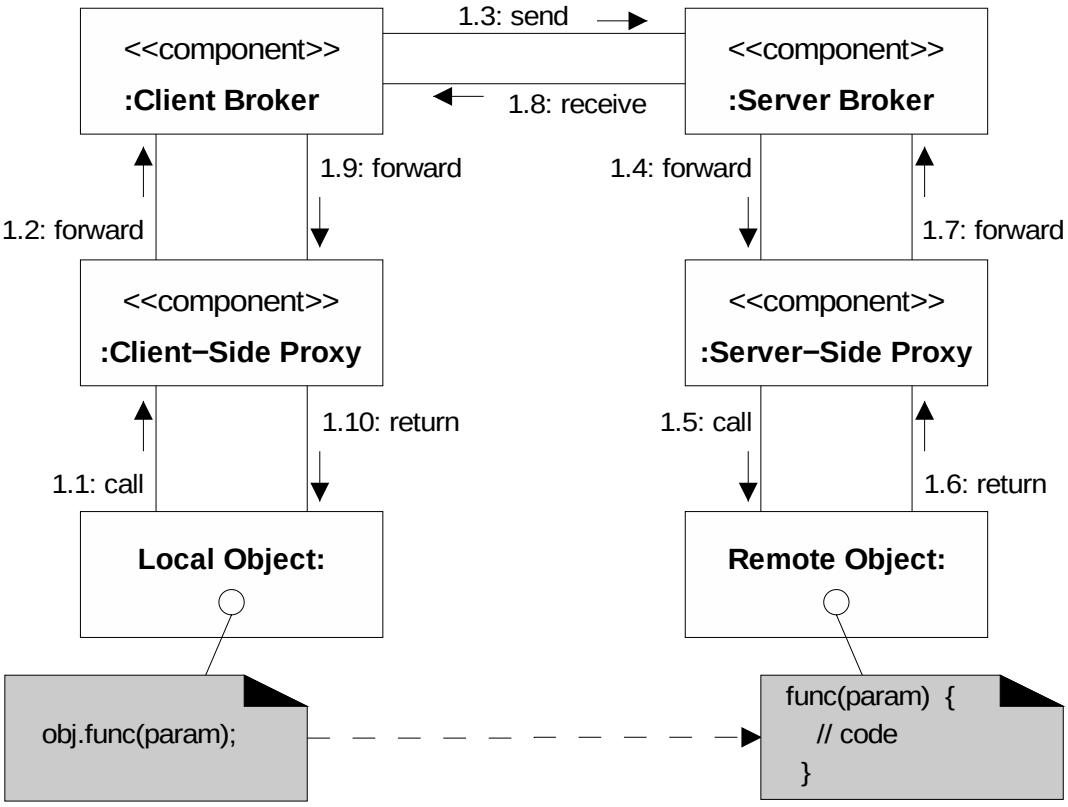

Figure 2.6 Remote Invocation

Garbage Collection and Persistence. As object references may be distributed among a set of clients or saved in a file for later usage, a general strategy for determining whether references to a given object exist somewhere in the system cannot be realised (Pacific Ocean Problem [Hen1998]). Instead, some implementations keep track of the number of live client references (e.g. by counting the number of live connections), while others employ some cooperative mechanisms in which clients notify the server when they don't need the object reference anymore.

Since servers are often considered to be long-living entities, their object references should be persistent, i.e. they should be kept valid over time, and clients can access the service by using the same object reference even if the server has been shut down or deactivated in between. A problem that occurs in this context is not to lose the server's state when deactivating it. A possible solution is to externalise the server object's state by storing it on disk or in a database – when reactivating the server, the object state can then be restored by internalising its state again.

The following sections present OMG CORBA, Sun Java RMI, and Microsoft DCOM, three major examples of current DOC systems.

2.2.4.2 CORBA

The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG). The aim of CORBA is to define a general framework for the construction of distributed object computing systems. Therefore, in a first step, the Object Management Architecture (OMA) reference model identifying and classifying the object types in a DOC system was proposed [OMG1997a]. As shown in figure 2.7, the model consists of five components.

The actual applications are represented by the application objects. These objects make use of the other object categories that provide a supporting environment. The central component enabling the interaction between components is the object request broker (ORB). Object services, common facilities, and domain interfaces are further object classes offering common services that are frequently needed in DOC applications. Object services provide very general low-level services, e.g. naming and trading, whereas common facilities offer higher-level functionality, and domain interfaces are services closely adapted for specific application areas.

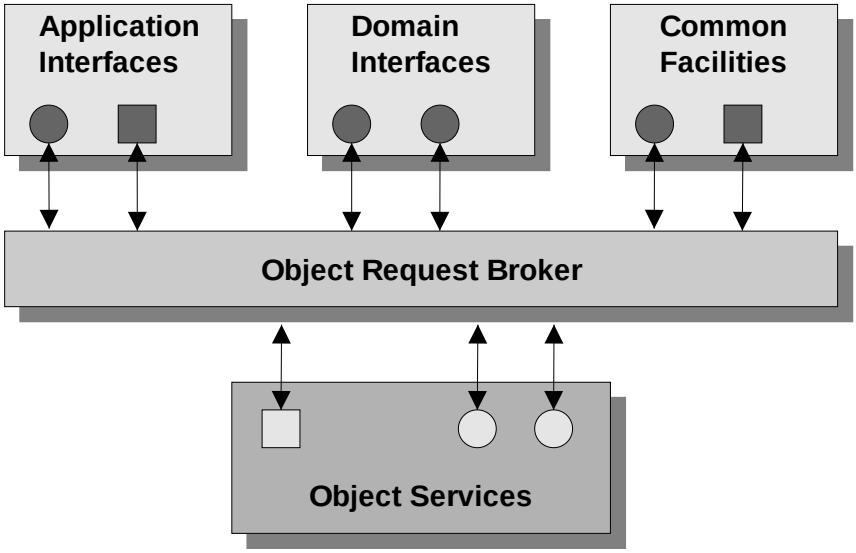

Figure 2.7 OMA Reference Model

Based on the OMA reference model, the CORBA specification [OMG1998c] describes the ORB and related mechanisms in detail. Furthermore, there are separate specifications for the object services [OMG1997b], the common facilities [OMG1995], and the domain interfaces.

IDL Interfaces and Language Mappings. In CORBA, interface definitions are made using a declarative *interface definition language* (IDL). Besides the common types and parameter passing methods, CORBA IDL fully supports exception handling and interface inheritance. An important feature of CORBA IDL is its language independence that allows to define the interface without touching implementation aspects. From the IDL description, an IDL compiler generates mappings to various programming languages, in which the actual object implementation can be realised.

Information extracted from the IDL file is used to generate stubs and skeletons for the client and server object implementations[2]. Additionally, the interface can be registered in an *interface repository* to dynamically construct requests at runtime without making use of the client stub; moreover, the *implementation repository* is used to store information about the activation of the actual object implementation.

Interoperable Object References. CORBA's way for uniquely addressing objects within the system is the usage of so called *interoperable object references* (IORs). When a new object implementation for a given service type is instantiated, it is the object adapter's task to choose a globally unique object reference. The ORB interface provides operations for converting IORs to strings and back, so that a simple mechanism for passing around IORs is given. As objects are only accessible by their object reference, CORBA objects are generally passed by-reference when used as parameters in method calls. An extension to CORBA allowing to pass objects by-value is currently being addressed by the OMG [OMG1998b].

According to the CORBA philosophy, IORs should be persistent in principle. The mechanism for reaching this goal is as described before: Objects should store their state before being deactivated and restore it when getting activated again. However, current ORB implementations treat object activation and deactivation in a rather proprietary way.

Finding Objects. As explained before, an issue in all DOC systems is to find objects supporting a given interface. In CORBA, this problem is delegated to the CORBA services that specify a *naming service* with the purpose of registering objects by giving a name and the respective IOR. Furthermore, hierarchical name spaces can be defined in order to structure large object spaces. Moreover, another CORBA service aimed at finding objects is the *trading service* that allows to register objects by giving specific properties instead of names, which represents an advanced method for locating services.

ORB Interoperability. As different vendors may develop different implementations of the CORBA specification, interoperability between ORBs from different vendors is an essential issue. A main problem of the first version of the CORBA specification was the missing protocol as well as data formats for the communication between ORBs. The OMG has addressed this issue by introducing the *general inter-ORB- protocol (GIOP)*, which specifies an exact transfer syntax and a standard set of message formats for ORB interactions on any connection-oriented transport medium. A specialisation of GIOP is the *Internet inter-ORB protocol (IIOP)*, which specifies how GIOP is mapped to TCP/IP connections. Furthermore, other *environment-specific inter-ORB protocols (ESIOPs)* are defined for interaction with other computing infrastructures like e.g. DCE.

Object Services. CORBA follows the concept of splitting the core communication infrastructure (ORB core) from additional services. Services are defined in a separate specification [OMG1997b] and include basic functionality like naming,

[2] In CORBA, the term "client" is used for the client implementation, whereas the term "object implementation" always refers to the implementation of a server object.

trading, and event handling, as well as more advanced services like lifecycle management, security, or transactions.

The advantage of this separation of the services from the ORB core is that it allows a modular and incremental construction of CORBA systems. Small, lightweight ORBs can be used for environments where no advanced services are needed (e.g. embedded systems), whereas fully featured ORBs can be deployed in more demanding environments (e.g. enterprise computing systems).

2.2.4.3 DCOM

The Distributed Component Object Model DCOM [Mic1998], [EE1998] is Microsoft's distributed object infrastructure. It was developed as an extension to the Component Object Model (COM) that enables the interaction of objects on a local machine.

Interfaces and COM Objects. To specify interfaces, DCOM uses its own interface definition language called MIDL. The Microsoft MIDL compiler is used to generate proxy and stub code[3] from these definitions. In contrast to CORBA, where source code for different languages is generated by the IDL compilers, DCOM defines a binary standard for generated interfaces that determines the physical layout of function tables for compiled interfaces.

COM objects are not considered as objects in the classical sense, they can rather be seen as components composed of a set of classical objects offering a set of interfaces to the clients. A special feature of COM is *aggregation*, which allows to integrate interfaces of different objects into one COM object transparently for the clients. The identification of objects within the distributed environment is realised by using Globally Unique Identifiers (OSF identifiers with a length of 128 bits) for classes and interfaces. COM classes are identified by class identifiers (CLSIDs), and interfaces are addressed by interface identifiers (IIDs). A COM object presents its functionality to the outside via its IIDs, where each interface has its own IID, and consequently a single COM object can possess several IIDs. This is different to CORBA and Java, where each object only possesses a single object reference through which all functionality is provided.

Accessing objects via IIDs can represent a major drawback because it is impossible to address a single instance of an object, as several objects providing the same interfaces can have the same IID; this problem can only be avoided by adding additional identifiers called *monikers* to objects.

Finding and Accessing Objects. Accessing a COM object from the outside is only possible by calling methods upon its IID. COM supports both static and dynamic invocation of methods. When the interface is known at compile-time the usual static invocation is applicable. For dynamic invocation, COM provides an approach called *automation*. Using this approach, the server COM object needs to support an interface called `IDispatch` that offers functionality for querying the available methods and issuing dynamic calls. Parameters are passed in a platform

[3] In DCOM, the client side proxy is called "proxy" and the server side proxy is referred to as "stub".

independent format called *network data representation*. When interface identifiers have to be passed, they are sent by means of so-called object references `OBJREF` that contain the full information to identify the server, object, and interface.

Object creation and location in DCOM is effected using object factories associated with COM classes. To locate these object factories, information about a COM class has to be contained in the Windows Registry, where the CLSID functions as a key. This information can be inserted into the Windows Registry either programmatically or manually. A *service control manager* (SCM) is responsible for looking up the local registry and contacting the respective object factory.

2.2.4.4 Java RMI

The Remote Method Invocation (RMI) [Sun1997] service is an extension for Java to enable distributed object computing. Like CORBA, RMI is also based on the broker and proxy patterns explained in the general concepts.

Interfaces and Remote Objects. *Interfaces* in RMI are described as standard Java interfaces that have to extend the interface `java.rmi.Remote`, and therefore no additional interface definition language is required. Functionality declared in a remote interface is implemented by classes that extend `java.rmi.server.RemoteObject` and its subclasses, and clients may access the remote objects only by using their remote references.

Finding and Accessing Objects. Parameter passing in RMI is realised with the Java `Serialisable` interface. Local objects that need to be passed as parameters have to implement the `Serialisable` interface and are passed by value. Remote objects are passed by-reference, making use of the possibility to transfer a serialised version of the proxy to the client. Consequently, multiple copies of the proxy can exist within a single client's virtual machine. RMI's ability to transfer proxy code to clients allows to set up and invoke requests dynamically in an advantageous way. In order to get information about the actual methods available and the respective parameters needed, introspection (Java Reflection) can be used to extract relevant information from the proxy code.

The problem of finding objects is realised by a naming component in RMI. On each node, an RMI registry daemon is running that functions as a name server for objects on its node. The registry provides a flat name space that allows to register objects by their name and remote reference. From the outside, the RMI registry is accessible via uniform resource locators (URLs) specifying the host name, port number, and name fields.

2.2.4.5 Summary and Future Directions

From the example systems described in this chapter, CORBA represents the most far reaching approach because it defines a complete framework for the construction of object-oriented applications. RMI's disadvantage is the limitation to Java programs only, and DCOM is mainly suited for Windows operating systems (although there are implementations for Unix, too), and lacks of historical drawbacks resulting from the incremental introduction of several features. More information and

additional overviews and comparisons of distributed object platforms can be found in [ZRS+2000], [PS1998], [CHY+1998], [TK1998].

Current work carried out in the field of CORBA contains the CORBA-3 standard [Vin1998] that improves and extends CORBA over several dimensions. Another big issue is the definition of the CORBA Components specification, which leads towards a more generalised framework [Joh1997b] for developing distributed applications.

2.2.5 Enterprise Middleware

A relatively old category of middleware that has probably existed much longer than the term middleware itself is enterprise middleware. This category comprises middleware that is mainly deployed in large enterprise information systems, for example in the financial sector (banks etc.) and insurances. Important characteristics of such information systems are a huge amount of interactive clients (dialog systems, terminals) accessing shared data with a high actuality. This results in a high rate of operations and a high data throughput, generated by a high number of small, but concurrent transactions, where synchronisation of parallel accesses is necessary. Moreover, the interactive usage requires short system response times.

All these requirements are achieved by providing highly available and fault tolerant environments, which rely on an appropriate underlying middleware. Transaction processing monitors and message-oriented middleware are the two main representatives of this category.

2.2.5.1 Transaction Processing Monitors

Transaction processing monitors (TP-monitors, TPM) provide an execution environment for transaction-oriented systems. Generally, a *transaction* is a set of operations that modify certain resources. In order to maintain a proper system state in concurrent environments, transactions have to fulfil the so called *ACID* properties [GR1993]:

- *Atomicity:* Transactions are considered as the smallest, indivisible units; this means that either all operations of a transaction are carried out, or none of them (“all-or-nothing” principle).
- *Consistency:* At the end of a transaction, the system has to be in a consistent state. If not, the complete transaction has to be undone (“roll back”).
- *Isolation:* This property requires that concurrent transactions do not influence each other (serialisability).
- *Durability:* When a transaction has been completed, the system state is modified permanently, i.e. it will e.g. not be set back by a system failure anymore.

TPMs provide a runtime environment that allows to guarantee the ACID properties. They are therefore often referred to as the “operating systems of transactions”.

The key components of transaction-oriented systems are the transaction managers and resource managers. *Resource managers* provide operations with ACID properties on single resources, for example data base systems, files, or queues. *Transaction managers* care for the overall coordination of transactions by applying a so called *two-phase commit*: Transactions are split into two phases, where in the first phase operations on the single resources are committed, but not finally executed, so that a roll back is still possible. If all operations can actually be carried out, they get finally carried out in the second phase. If at least one operation could not be executed, a roll back of the complete transaction is made.

The behaviour of TPMs has been standardised in the X/Open Distributed Transaction Processing (DTP) Model [X/O1991]. Figure 2.8 illustrates the components of the X/Open models and their interactions.

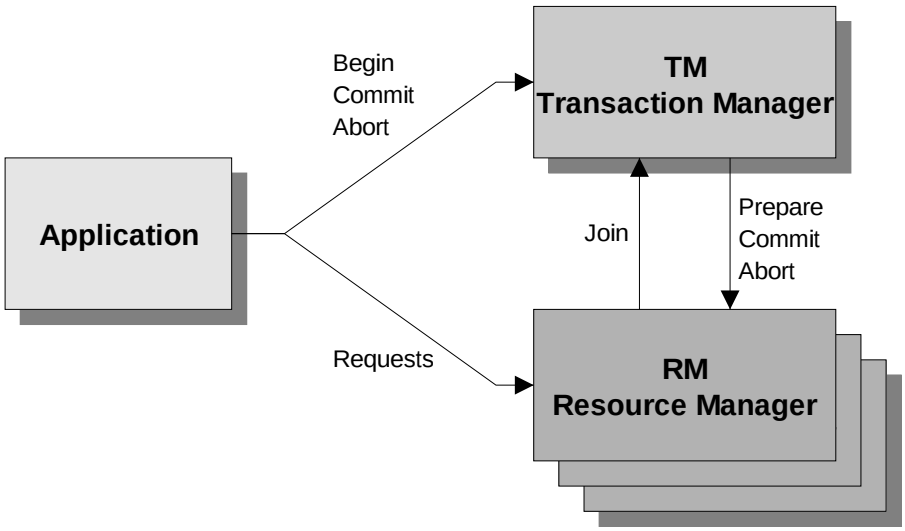

Figure 2.8 X/Open Transaction Processing Model

Hence, TPMs represent a rather high-level form of middleware, in which advanced goals have to be achieved. Mechanisms supporting the integrity of complete enterprise information systems are provided in addition to high performance, availability, and scalability. This goes much further than pure communication middleware described in the previous sections.

2.2.5.2 Message-Oriented Middleware

The second enterprise middleware platform is message-oriented middleware (MOM). The basic principle of MOM is to transfer data by means of messages that are put into a *queue* by the sender, and read from the queue by the receiver at an arbitrary time. Within MOM, two sub-categories can be distinguished:

- *Message-passing:* This kind of MOM is based on direct application-to-application communication. Messages from the sender are directly transferred to the receiver, and for the applications, no queues are visible.

Message-passing is only possible in connection-oriented systems, where a connection between sender and receiver has to be established before data transfer.

- *Message-queuing:* With message-queueing, data transfer is implemented by using explicit queues. In this case, communication is connection-less, i.e. the receiver needs not be active at transmission time because messages can be stored in the queue until the receiver becomes active.

The main benefits of MOM are the asynchronous and connection-less communication with message-queueing systems. In large wide-area distributed networks, this is a key feature as application components may not be active or reachable at all times, such that decoupling senders and receivers increases system performance and stability.

Other relevant features of MOM include guaranteed message-transfer, which provides a method for senders to make sure that asynchronously sent messages have actually arrived at the receiver component. Furthermore, fault tolerance is of crucial importance within MOM systems. This includes the detection of incomplete or modified messages as well as recovery after system crashes on any hop between sender and receiver.

The most important usage scenario for MOM are certainly batch processing systems, where asynchronity and guaranteed transfer are the main aspects. Considering the level of abstraction, on the one hand MOM resides on a rather low level as messages are the only communication entity, but on the other hand, its applicability in heterogeneous areas including host-based systems as well as workstation environments puts it onto a higher level of interoperability.

2.3 Summary

In this section, we have defined the term “middleware” starting from distributed computing and the client/server model. It has become clear that middleware comprises a lot of different implementations for rather diverse application scenarios. According to the users’ needs, the key trade-off between universality, complexity, and performance has to be found. The more specific and tailored to given application scenarios a middleware is, the better is the performance that can be reached. On the other hand, ease of use and simple programming models in most cases object to performance, too.

The platforms described in this section include traditional approaches that have built a foundation for current implementations, as well as middleware for high-performance computing and enterprise middleware, which represent fairly application domain specific systems. Finally, distributed object-computing platforms represent the most general purpose middleware; they have been explained in more detail as they are of interest for the implementation of the monitoring system that will be shown later. Table 2.1 summarises the characteristics of the described middleware according to the following properties[4]:

[4]The range of ratings is “$-$”, “0”, “$+$”, “$++$”, standing for poor, medium, good, or very good

Middleware / Criteria	RPC	DCE	PVM MPI	DSM	Meta-computing	DOC	TPM MOM
Abstraction Level	0	+	−	0	+	++	+
Universality	+	+	−	−	0	++	0
Performance	0	0	++	0	+	0	+
Overhead	+	−	+	+	0	+	−
Portability	0	+	0	−	0	++	+
Interoperability	−	0	−	−	−	++	+

Table 2.1 Comparison of Middleware Characteristics

- *Abstraction level* characterises the ease of use of a middleware for the programmer and the availability of advanced services.
- *Universality* indicates the applicability of a middleware to different scenarios. The more generally applicable a middleware is, the higher is its universality.
- *Performance* evaluates the runtime efficiency of the execution of applications using this middleware.
- *Overhead* denotes the general effort needed to deploy this middleware. The more light-weight middleware is, the lower is its overhead. Hence, a small overhead results in a good rating here.
- *Portability* characterises the applicability of this middleware in heterogeneous environments, i.e. without considering interactions to other middleware products.
- *Interoperability* finally stands for the interaction with other middleware platforms, including legacy integration.

The table shows that middleware with a higher universality, abstraction level, and interoperability tends to have a higher overhead and a lower performance. More specialised middleware can partially compensate this through its adaption to given application scenarios. Nevertheless, this rating cannot represent a completely extensive survey of middleware, as the variety of products is enormous, even within

fulfilment of a criteria.

the single categories. Nevertheless, it can be concluded that distributed object-computing systems tend to be the most general approach for middleware up to now and act as an integrative platform for other concepts due to their high interoperability and universality.

Trends. Current trends for middleware platforms lead to two distinct directions: On the one hand, optimised platforms for specific application domains like high-performance computing are developed. On the other hand, big effort is put into integrative middleware like CORBA, with the goal of improved interoperability between different middleware products and the integration of legacy applications. Exemplary indicators for such developments are e.g. Object Transaction Processing Monitors (OTPM), which represent a combination of DOC and TPM functionality. This tendency towards highly interoperable and integrated systems justifies a new need for tool concepts and environments that are able to deal with such complex systems.

Chapter 3

Monitoring and Management

This chapter introduces general concepts, mechanisms, and systems for monitoring and managing distributed environments. Developing and maintaining complex applications has become impossible without a sufficient tool support. Monitoring and management systems provide the basis for on-line tools, which are a major technique for handling the on-line phases of the software lifecycle. As we will see, approaches for monitoring and management within the middleware landscape are very diverse, but we can extract common concepts that represent the foundation for a general monitoring and management approach.

3.1 General Principles

Before going into detail with current implementations of monitoring and management systems, we will introduce their fundamental terms and mechanisms in this section.

3.1.1 Software Development and On-line Tools

Basically, *tools* are programs used to support or simplify the development and deployment of applications. This comprises software development tools like integrated development environments with analysis and design tools, GUI builders, and debuggers, as well as *on-line tools* that are applied to the developed application at runtime. In contrast to *off-line tools* that only collect information at runtime for later analysis, the characteristic of on-line tools is to allow immediate manipulations of the running program in addition to data collection. This leads to the following definition:

On-line Tools: On-line tools are tools applied to an application during runtime. They have the capability to collect data for later analysis as well as the potential to manipulate the current execution of the application.

Further characteristics of tools are whether they work in a centralised or distributed way, or whether they are interactive or automatic ([Lud1998], p. 20). In the course of this thesis, when talking about tools, we will be referring to on-line tools if not explicitly mentioned otherwise. Examples for such on-line tools are performance analysis tools, debuggers, visualisation tools, load balancers, or computational steering tools (see [Tri1999], p. 25ff), i.e. tools applicable as soon as there is a running prototype of the application.

Moreover, an appropriate tool support becomes more important with the growing complexity of applications and environments in which applications are deployed. Therefore, tools play a significant role in the field of software development for parallel and distributed systems [Bod1995], [Wal1995]), including middleware-based applications.

3.1.2 Monitoring

In order to implement tools as described before, mechanisms to collect required information from the observed system or to influence it are needed. Every tool could implement this functionality on its own, but this would cause a high effort and potentially cause problems concerning the coexistence, integration and interoperability of different tools. Therefore, in most cases an additional monitoring abstraction layer separating the tools from the observed applications is introduced. This layer is implemented by a *monitoring system* (or simply monitor) that provides basic functionality for accessing and manipulating the observed applications. This monitoring system processes commands or requests from tools and possibly provides the synchronisation of concurrent tools. Figure 3.1 illustrates the resulting three-tier model consisting of tools, the monitoring system, and the monitored applications.

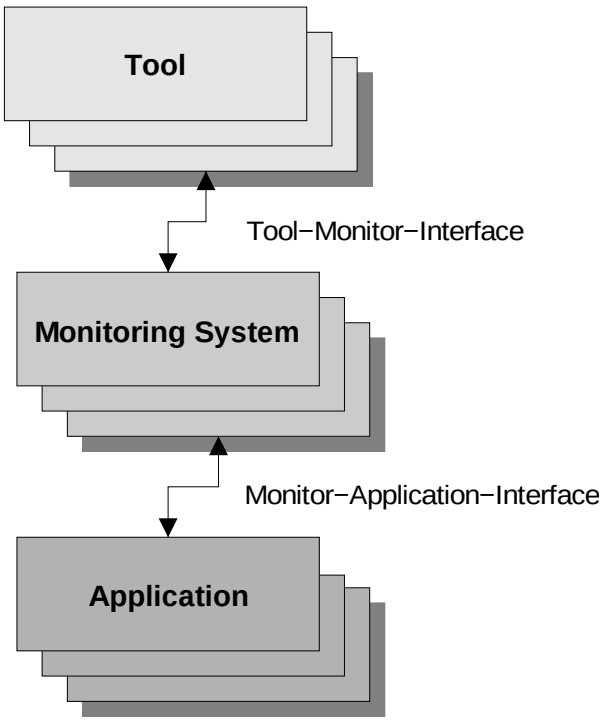

Figure 3.1 Three-tier Monitoring Architecture

The interface for accessing monitoring functionality from the tools is called the *tool-monitor interface*, and the monitoring interface needed to access and control applications is called the *monitor-application interface*. As for tools, it can be distinguished between *off-line monitoring*, which only allows the observation of

programs without manipulating them [JLSU1987] and *on-line monitoring*, which also allows manipulations. [Tre1996]. Hence, on-line monitoring can be defined as follows:

On-line Monitoring: On-line monitoring comprises techniques and mechanisms to dynamically observe and potentially manipulate applications at runtime.

On-line monitoring can consequently be seen as a technical precondition for building on-line tools. As for tools, the term "monitoring" will be referring to on-line monitoring unless otherwise mentioned.

3.1.3 Management

Another important term in the field of software development and deployment is *management*. General definitions see management as *all activities undertaken to reach a business-goal oriented effective and efficient deployment of a distributed environment and its resources* [HAN1999]. Here, management on different levels of interest can be distinguished: *Network management* deals with communication services and network components, *system management* covers resources of end systems, and *application management* is responsible for distributed applications and services. Functionality carried out in the management process can be classified into the following categories, which are also called *management functions*:

- *Configuration management* cares for the adaption of systems to specific deployment conditions.
- *Fault management* is responsible for discovering, isolating, and fixing abnormal behaviour.
- *Performance management* has the aim of realising given quality-of-service requirements.
- *Accounting and user management* covers administrative tasks concerning naming, authorisation, and billing functions.
- *Security management* comprises the analysis of given threats and the implementation of appropriate mechanisms to protect the system.

These management functions are implemented by *management tools*, which can be classified as on-line tools that make use of a monitoring system to execute their tasks. Hence, our definition of management can be formulated in the following way:

Management: Management covers all activities required to establish management functions, thereby making use of management tools and on-line monitoring techniques.

Other definitions of monitoring and management use the terms in a different way: They define monitoring as a way only to observe a system without manipulating it, and management as the manipulation facility based on monitoring [MS1995]. Nevertheless, we will proceed with the above definition throughout this thesis.

3.1.4 Classification of Tools

As we have seen that tools with different functions are attached to different phases of the software lifecycle, we can give a general classification of tools now. Figure 3.2 shows a rough partitioning of the software lifecycle into three major phases:

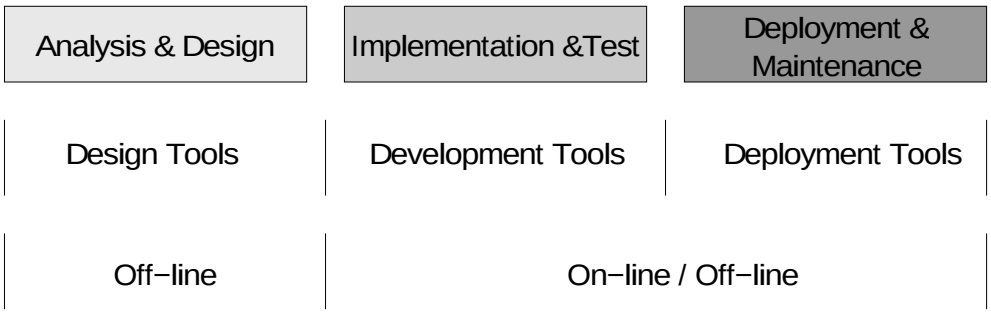

Figure 3.2 Coarse Software Lifecycle and Tool Types

- *Analysis and design phase:*
  In this phase, no on-line tools are required since there is no running prototype of an application. Tools used for this phase are classified as *design tools*, including e.g. modelling tools, code generation tools, or GUI builders.

- *Implementation and test phase:*
  During implementation and testing, tools like debuggers, visualisers, or performance analysis tools are applied. We classify these tools as *development tools*, as they are mainly used with running prototypes, but before delivery of the software product.

- *Deployment and maintenance phase:*
  After completion of applications, they are deployed in a suitable operating environment. During this phase, *deployment tools* care for a smooth operation of single applications and the complete environment. This includes management tools as well as application-specific tools. For example, performance management can balance load within the complete environment, but single applications may also use their own load balancing mechanisms [SR1997], [RS1997], [Rac1997].

As management tools are applied during the deployment phase and implement more advanced features concerning the whole operating environment, they can be categorised as high-level tools compared to development tools used for implementation and testing, which are mainly concerned with issues of single applications. We will therefore refer to tools using lower-level system information, like e.g. debuggers, as *low-level* tools, and to management tools operating on more abstract entities as *high-level* tools.

3.1.5 Monitoring Mechanisms

Monitoring an application requires techniques to gather information from a running program and to perform manipulating operations on the program. Basically, the two general approaches are hardware-monitoring and software-monitoring. Hardware-monitoring provides the only possibility of not influencing the running application at all, but is much harder to realise and less flexible. We will therefore concentrate on software-based monitoring techniques in the following.

3.1.5.1 Events and Actions

All monitoring techniques rely on the detection of *events* that are relevant for the program behaviour. Of course, the notion of relevance depends on the specific tool and application; therefore, monitoring systems allow tools to define relevant events dynamically. With these events, certain *actions* are triggered, i.e. commands that are executed when the event occurs. These commands can either carry out data collection, or also manipulate the running program. The mechanism of events and actions is also called the *even-action paradigm*.

3.1.5.2 Instrumentation

Instrumentation represents the basic approach to implement software-monitoring techniques. The idea of software instrumentation is to insert additional code carrying out monitoring tasks at appropriate positions. Depending on the module of code-insertion, several instrumentation techniques are distinguished (see [Tre1996], p. 26ff, [STK1996], [SMSP1996]):

- *Application instrumentation:*
  This technique covers all mechanisms that instrument the application itself. It can be implemented by *source-code instrumentation*, which inserts monitoring-code at appropriate positions within the application sources. This approach is very flexible and portable, but the sources have to be available and recompiled. *Object-code instrumentation* in contrast adds or replaces code inside the compiled application objects. The approach only requires relinking and no recompilation, but is technically harder to implement. Portability is also an issue when object formats are not identical on different platforms, but with standardised platforms like Java where the object format is identical on all platforms, portability is given.

- *Runtime environment instrumentation:*
  The second approach for software-monitoring is runtime environment instrumentation, which modifies components of the runtime environment instead of the applications. It is mostly implemented by instrumenting libraries linked with applications or the operating system. Moreover, the technique also allows for *middleware instrumentation*, as most middleware platforms are based on libraries or system calls that implement middleware functionality.

The advantage of instrumenting the runtime environment is transparency for the applications, i.e. applications need not be aware of the presence of a monitoring

system at all. Application instrumentation has the advantage of allowing very specific and domain-dependent code insertions that can be more efficient than general runtime instrumentation. In practice, several instrumentation techniques are often combined.

3.1.5.3 Wrapping

A technique that is commonly used to instrument middleware is called *wrapping*, which works as follows: An entity that has to be monitored is renamed, and a new entity with the name of the original entity is added to the system. Now, when clients access the monitored object, they automatically call the new monitored object. The monitoring object can carry out monitoring actions and finally call the original object. Hence, this approach is completely transparent for clients and does not need to modify the original entity (except renaming it). The advantage of this approach is its simplicity, the drawback is its limited usability because operations inside the wrapped entity cannot be observed. Nevertheless, in the field of middleware wrapping is an advantageous technique as basic entity access events need to be monitored frequently.

3.1.6 Problems of Monitoring

There are several issues that have to be taken into account when software-monitoring systems are used for observing application behaviour. Some issues concern sequential as well as parallel systems, while others are only of interest when parallel systems are being monitored. Among the most important issues are the following:

- *Monitoring overhead:*
  As mentioned above, one of the main problems of software-monitoring is its impact on the observed system. Any kind of instrumentation requires additional actions to be carried out, resulting in a delay of the original program. This delay causes problems especially for performance analysis, where execution times have to be measured as exactly as possible. Therefore, the *overhead* caused by the monitoring system should be kept small.

- *Global time and event ordering:*
  In distributed systems, the time of the local clocks may differ from machine to machine. This influences the ordering of events, when using time as an ordering criteria. Therefore, either a mechanism for synchronising these clocks or an additional event ordering mechanism needs to be found.

- *Non-deterministic behaviour and races:*
  In a distributed system, non-deterministic behaviour occurs when several parallel components compete for output from or to another component. This has the consequence of potentially different program executions with a different partial event ordering for identical input data. Situations where two or more components compete for the state of another component are called

races ([Obe1998], p. 25ff). This leads to non-reproducible program executions, which causes a problem e.g. for debugging distributed applications.

- *Heterogeneity of the observed applications:*
  When monitoring heterogeneous application components, different kinds of instrumentation may be required. Difficulties for the monitoring process result from different characteristics of these instrumentations, either coming from functional or overhead related issues of the instrumentation.

Generally, issues of monitoring systems depend on the abstraction level on which monitoring is carried out. For example, performance analysis of high-performance computing systems is much more influenced by delays and overhead than high-level management tools. Vice versa, issues of heterogeneity are mostly an issue of management and not of low-level tools. Hence, the selection of instrumentation techniques requires great care, depending on the requirements of tools using the measured data.

3.2 Common Approaches and Systems

Depending on the application domain and the respective tools, currently existing monitoring and management systems are very diverse. This section gives an overview about common monitoring and management systems and approaches. We classify those systems into the categories for parallel programming environments, distributed object systems, application management systems, and network management systems.

3.2.1 Monitoring Systems for Parallel Programming

In this section, we discuss approaches for monitoring parallel programming environments. As explained in the previous chapter, these systems are particularly related to message-passing and distributed shared-memory systems.

3.2.1.1 OMIS – On-line Monitoring Interface Specification

The *On-line Monitoring Interface Specification* (OMIS) [LWSB1997] is a joint project between the Lehrstuhl für Rechnertechnik und Rechnerorganisation, Technische Universität München, Germany (LRR-TUM), and Emory University, Atlanta, GA, USA. Its main goal is to develop a standard for runtime tool development in parallel and distributed systems.

OMIS is the first system to follow the idea of separation of monitoring system and tools through a standardised interface. Consequently, tools can be developed platform independently, while only the monitor has to be ported to different platforms. This has the advantage of bringing n tools to m platforms with an effort of $n + m$ instead of $n * m$, because tools only need to be developed once, and the monitoring system needs to be ported to every platform. Without this proceeding, every tool would have to be ported to every platform.

The OMIS interface is based on requests being sent from tools to the monitor, and on the event-action paradigm for event processing. The system model of OMIS relies on a hierarchy of the abstract objects nodes, processes, threads, message queues, and messages. Objects themselves are referred to by using tokens, which can be converted into other types by token expansion. For example, when a request requires a thread token as an argument, but a process token is given, this process token is automatically broken up to all threads within the context of the process.

The basic system architecture of OMIS is based on the three-tier model shown in figure 3.1. Additionally, OMIS allows to be extended by tool extensions or monitor extensions. These extensions can be used to implement distributed tools or to support specific programming environments. Based on the OMIS standard, the *OMIS compliant monitor* (OCM) is an implementation of the OMIS concept [WTL1998], and the first version of OCM is built with support for PVM and therefore called OCM/PVM. The access of OCM to application data is effected using instrumentation of parallel programming libraries or other specialised runtime libraries. Concerning the distribution of requests and the gathering of results, OCM implements a node distribution unit (NDU), which is the only component interacting directly with tools. The NDU is responsible for accepting requests from tools, distributing them to a set of local monitors residing on every participating machine, assembling collected data, and passing the result back to tools.

Evaluation. All in all, the benefits of OMIS/OCM are its approach to define a standard interface for tools, thus reducing the complexity of tool development, and introducing the separation of tools, monitor, and applications. Also, the token concept with its expansion mechanisms represents a major progress for tool development. Nevertheless, from the middleware point of view, OMIS/OCM is a rather low-level approach, targeted mainly at high-performance message-passing programs. It is not designed for usage in middleware environments, where heterogeneity and integration of management functionality are of interest, too.

3.2.1.2 Coral

An example for a monitoring system supporting DSM is *Coral* [Zor2000]. Developed at LRR-TUM, a current implementation of Coral is targeted at the TreadMarks DSM library [ACD+1996], but easily adaptable to other DSM middleware.

Coral's principal approach is to provide the abstraction of shared memory to parallel tools. Therefore, the monitor has to manage DSM mechanism transparently, hiding DSM internals from tools. The architecture of Coral is based on a client/server model, the event-action paradigm, and a layer model decomposing monitoring functions into three units.

Due to performance reasons, the implementation of Coral internally uses a proprietary message-passing communication library. Local communication between the monitor and the applications is effected using shared memory segments. Advanced monitoring functions like object migration and checkpointing are integrated by specific migration or checkpointing frameworks that are used by the Coral core.

Evaluation. The main contribution of Coral is its concept of allowing tools to operate on the same abstraction level as the actual middleware. This enables application-level monitoring of DSM programs, in addition to lower-level monitoring of the DSM middleware itself.

3.2.2 Tools and Monitors for Distributed Object Computing

Tools and monitoring systems for distributed object computing environments are rather diverse. To a great extent, proprietary tools for specific DOC implementations are supplied by the vendors. Additionally, tools and monitors are often restricted to certain DOC categories, for example CORBA. A generic monitoring system supporting heterogeneous DOC environments with several different DOC categories does not exist. Furthermore, most systems merge tools and monitor into one monolithic system, hindering easy tool construction or extension. Most approaches described in the following passages deal with CORBA systems, because for other DOC middleware, only few sophisticated on-line tools or monitors exist.

3.2.2.1 CORBA Assistant

CORBA Assistant is a management system for CORBA developed by Fraunhofer-IITB [Fra1997]. The goal of CORBA Assistant is to provide an application management facility that can be part of a larger integrated management environment. The design of CORBA Assistant is based on the following principles:

- *Managed objects:*
  In order to get an external management view on a resource, managed objects are introduced. Managed objects represent wrappers around resources that have to be managed, adding functional extensions for management purposes to the normal interface of the managed resources. This includes attributes of the management resource (for example version numbers or component-specific counters), management operations like resetting the managed objects, or asynchronous event notification functionality.

- *Management protocols:*
  CORBA Assistant follows the principal architecture of common management systems: An agent is responsible for providing a management view upon managed objects by implementing the respective functionality. The functionality is accessed by a management tool (or manager) that processes information retrieved from a set of agents. The interaction between agents and manager is based on CORBA communication mechanisms.

- *Management Information Bases (MIBs):*
  Management Information Bases (MIBs) are an important concept for representing information about the managed system (see section 3.2.4.3). MIBs are defined using a formal notation and can be read by any management tool as the format is standardised. The MIB defined by CORBA Assistant

contains information about running processes implementing CORBA clients and servers, as well as more detailed data about the CORBA objects existing within clients and servers. This allows for observing current system configurations (e.g. the distribution of CORBA objects among the participating hosts), collecting accounting data (e.g. CPU usage or data volumes being transferred), or managing performance (e.g. measuring response times or throughput).

An implementation is currently only available for the Orbix CORBA product, and based on the Orbix filter mechanism, which is not CORBA compliant. The implementation of the managed objects concept relies on the following instrumentation techniques:

- *Object implementation phase:*
  Here, instrumentation code needs to be added to the source of object implementations (source code implementation).
- *Linking phase:*
  A management library has to be linked with the instrumented and compiled objects.

The object definition (CORBA IDL) of server objects is not influenced by the above instrumentation and applications can be started as without management facility. Together with CORBA Assistant, the *Orbas* management tool is delivered. Orbas allows to browse the MIB, display management attributes and performance graphs.

Evaluation. Summing up, the main advantage of CORBA Assistant is its usage of a CORBA MIB in a standard format, which enables different tools to be built upon the management agents. On the other hand, the large overhead caused by using heavy-weight MIBs causes performance problems and prohibits the application for high-performance domains. The major disadvantage of CORBA Assistant is its restriction to Orbix applications only. While an extension to other CORBA implementations could be realised with minor effort[1], the integration of other middleware platforms is not feasible. This represents a main limitation of CORBA Assistant with respect to monitoring systems for heterogeneous middleware platforms.

3.2.2.2 AppMan

In the *AppMan* project [KDW+1999] at Fachhochschule Wiesbaden, Germany, a management system for complex distributed CORBA-based applications is being developed. Its major goal is to integrate application and network management functionality, primarily concerning configuration, event, and performance management. The current AppMan implementation itself is implemented using CORBA.

Information Model. The basic idea of AppMan is to define an information model describing the structure of the observed system. As it is impossible to define a general model for all kinds of applications, AppMan focuses on CORBA applications and the respective communication model.

[1] By replacing Orbix filters with interceptors compliant to the CORBA standard.

The AppMan information model defines the four layers application, CORBA, system, and network as fundamental entities. Every layer can be subdivided into its composing elements: Applications consist of components and subcomponents; the CORBA layer comprises server objects, object adapters, and the local ORB subcomponent; the system is based on processes and hosts; the network finally consists of nodes, subnetworks, and networks. Based on this model, concrete applications are represented by instances of the presented entities and their relationships. Moreover, the overall information model is divided into the two views static and dynamic view.

Architecture and Implementation AppMan's architecture relies on an *event management service* (EMS) responsible for the communication within the management system. Through the EMS, sensors, evaluators, a correlation engine, management automata, and actors interact in the following way:

- *Sensors* extract management relevant data from the observed system. They can be implemented using different kinds of instrumentation, or by using other existing information sources like e.g. SNMP.

- *Evaluators* provide the transformation of raw data from the sensors to meaningful metrics. They include e.g. computation of minima, maxima, average values, or deviations.

- The *correlation engine* filters the incoming events with respect to the information model, such that only relevant events are passed to the affected components.

- *Management automata* are tools automatically processing filtered events and generating management actions. These automata can be nested, work in parallel, or be meshed.

- *Actors* are the last element in the event chain and execute management actions that are issued by the automata in order to implement feedback to the observed system.

In addition to these components, a topology service is used to store static structural data of the environment.

Evaluation. AppMan's concept of generic sensors enabling the integration of different information sources is one of its main benefits, as this approach represents a foundation for monitoring heterogeneous middleware. Also, the possibility to add different kinds of tools to the EMS is an open and extensible solution.

However, the exclusive concentration of the information model on CORBA limits its applicability essentially. Also, the decomposition of the four layers of the information model into the given subcomponents is more or less arbitrary and may not be useful for some application domains.

3.2.2.3 OrbixManager

OrbixManager [Ion2000] is a vendor proprietary management tool for the IONA Orbix CORBA implementation. Its purpose is to simplify administration and management of a CORBA environment. OrbixManager relies on managed objects that are controlled by the management tool through so called *management agents*. Standard management information bases are used to describe the characteristics of managed objects. The architecture of OrbixManager consists of four components:

- *Management library:*
  This library has to be linked with Orbix applications and contains an interface for the management service to access management functionality; hence, the management agents are implemented within this library.

- *Management service:*
  This component represents the actual monitoring system, implemented by a single stand-alone application mediating between the management library and the management tool.

- *Management tool:*
  The only tool available with OrbixManager is the management tool that provides a GUI for data measured by the underlying monitoring system.

- *SNMP proxy:*
  In order to make use of SNMP-oriented information sources or to deliver data to other SNMP-tools, the SNMP proxy, acting as a gateway between CORBA and SNMP, is needed.

An advanced feature of OrbixManager is the possibility to observe single threaded pure clients. Observing this kind of clients represents a problem for monitoring systems as they can hardly be monitored without severe interference. OrbixManager's solution is to keep a proxy for such clients within the management service, which is contacted by the management library (running within the client process) at specific points defined by a "rendezvous policy".

Evaluation. OrbixManager is a proprietary tool delivered from an ORB vendor for its CORBA implementation. Due to the non-intrusiveness of the approach that results in a minimal overhead, the performance of the monitoring and management system is good. However, this can only be reached by using Orbix internal interfaces that are not available for any other CORBA implementation; hence, portability to other middleware platforms is hardly possible.

Furthermore, the functionality of OrbixManager is limited to features of the management tool that is tightly coupled with the management service. Although communication within OrbixManager relies on CORBA IIOP, it is not possible to access functionality from outside as the interfaces are proprietary and not documented.

Consequently, as there is no separation between tool, monitor, and application instrumentation through defined interfaces, the SNMP proxy represents the only

technique for interaction with other tools or applications. This is not sufficient for building flexible tools or the adaption of other ORBs or middleware products to the monitored environment.

3.2.2.4 Inprise AppCenter

AppCenter is a vendor proprietary application management tool-set for the Inprise Visibroker ORB and the AppServer integrated development environment [Inp2000].

AppCenter follows a model-based approach to manage distributed applications: With an application-modelling environment, application-focused views of the environment can be defined. Starting with the topology of an application, groups representing the building blocks of a distributed application can be defined, and dependencies between components can be modelled. Further relationships between components allow to model fault tolerance by specifying backup objects that are automatically activated when the primary object goes down, or, on the other hand, containment relationships that specify components being included in other components, requiring the container to be started before the contained entity.

The basic architecture of AppCenter consists of three parts:

- The *user interface* is a Java-based console from which all activities of the management system are controlled. The console can be customised to show different views of the modelled applications.

- A *core management service* contains the business logic of AppCenter; it stores the application model and is responsible for the execution and coordination of management functions.

- *Management agents* are located on every participating host and take care of the local execution management tasks.

Hence, in general monitoring terms, the user interface represents a tool, the core management service comprises the monitoring system itself, and management agents contain the instrumentation code.

The functionality available within AppCenter comprises the following areas:

- Active management functions allow to start or stop applications or components. From the predefined model, actions necessary to start applications can be derived from dependencies and containment relationships.

- Performance monitoring creates built-in performance statistics of modelled application objects.

- Load balancing mechanisms enable activation of a variable number of servers, and the assignment of clients to least-loaded servers.

- An event-action mechanism allows to filter specific events and trigger actions – like e.g. sending email or starting other processes – with these events.

Moreover, AppCenter provides the possibility of exporting or importing application models using an XML description, and defines an SNMP MIB to interact with other management systems. Also, in addition to CORBA objects it is possible to monitor DCOM or EJB (Enterprise Java Beans) components.

Evaluation. AppCenter represents the most far reaching approach for a proprietary tool coming from an ORB vendor. Its ability in principle to integrate CORBA, DCOM, and EJB demonstrates a step towards open tool environments as well as enabling SNMP connections. Also, XML export and import functions for application models make it possible to process these data with other tools.

However, reasonable application modelling with AppCenter can only be carried out in collaboration with Inprise Visibroker and AppServer because the systems are tightly integrated. The integration of other ORBs, or DCOM and EJB components not based on Inprise products is left open. Like with other proprietary products, the tool-monitor interface is not open and the integration of own instrumentation techniques is not possible, which makes it impossible to build own tools or inspect other middleware.

3.2.2.5 Silk for CORBA

The Silk for CORBA [Seg2000] software is a third-party family of products for testing, monitoring, and controlling distributed CORBA applications. It consists of four separate tools implementing different tool functionality:

SilkPilot. SilkPilot is a tool for validating the behaviour of CORBA servers. It is a GUI tool that allows to connect to CORBA servers and to issue requests based on the interface descriptions that are automatically retrieved from a CORBA Interface Repository. A sequence of method calls issued during a session can be converted into Java source code for later replay.

SilkObserver. SilkObserver is a tool for tracing and monitoring distributed communication. It is based on the instrumentation of clients and servers by linking additional observer libraries with them, a setup editor that allows to configure filters for specific methods to be traced, a viewer for graphical display of the data, and a data recorder combined with a parser for off-line replay and analysis of measured data.

SilkPerformer. SilkPerformer provides a load testing facility for CORBA servers. Load testing is based on IIOP traffic observation with a so called Internet Recorder, which is integrated into the socket library and records IIOP traffic before transmitting it to the actual clients.

The IIOP packages can be replayed exactly as recorded or reconfigured to generate different traffic profiles. Reconfiguring and changing invocation parameters requires additional effort as parameter types cannot be determined by observing IIOP only. Therefore, SilkPerformer is able to extract parameter information from the interface definitions, which enables parameter types to be associated with actual IIOP packages.

With a workload definition wizard, it is then possible to generate traffic seemingly coming from different machines with a variable number of concurrent users and a variable transaction frequency. During the load tests, results can be presented with a GUI tool showing virtual users, transaction status, response times, data throughput, and occurring errors.

SilkMeter. SilkMeter makes it possible to implement security and accounting functions for CORBA servers. Security relies on a usage control manager and server controlling the access to specific objects on the basis of secret keys. Access rights can be given according to several different access control schemes like floating, node-locked, time-limited, or personal licenses.

Accounting functions are implemented either by postage- or gas-metering: With the former, only a predefined number of accesses to a certain resource is allowed, while with the latter, the user is billed after making use of the resource. With SilkMeter, both kinds of metering are available either for users, groups, or hosts. As already mentioned, most Silk functionality is integrated by linking additional libraries with the clients and servers. Furthermore, for advanced functionality like accounting, extra source code needs to be added to clients or servers.

Evaluation. Compared to other tools and monitors described before, Silk for CORBA brings some non-standard features like access control, accounting, or load testing to the spectrum of relevant DOC tools.

However, its approach of delivering four stand-alone tools that have to be applied separately to an application is a drawback that reflects the lack of a versatile monitor on which to construct tools by using an open API. Moreover, only the Orbix and Visibroker CORBA implementations are supported by Silk, and there is no easy way of adapting other platforms. Thus, the overall functionality of Silk for CORBA can be classified as good, whereas the general architecture is its limitation.

3.2.2.6 Others

In addition to the systems described in detail, there are other approaches or products worth mentioning, either because of specific features, or for the sake of completeness.

The *MODIMOS* Managed Object-Based Distributed Monitoring System [ZL1995] aims at delivering an adaptable platform for visualising heterogeneous distributed applications. It has the interesting concept of so called *plugs* used to adapt new platforms to a given monitoring system. This enables an easy integration of new systems by defining standard interfaces for plugs.

For handling events within CORBA monitoring and management systems, object-oriented frameworks deriving all events from a single root class have been proposed [Sch1997]. By specifying common properties of all event classes, a more generic event handling mechanism can be implemented utilising this approach.

Furthermore, there are other proprietary tools delivered by middleware vendors. For example, *BEA Manager* [BEA2000] is an administration tool for the BEA WebLogic CORBA-based middleware. We do not discuss all of these tools in detail as their functionality is similar to the features of OrbixManager or Inprise AppCenter

vendor tools already analysed.

All of the above-metioned systems are mainly targeted at CORBA middleware. This can be explained either with a lacking tool support of other middleware platforms, especially DCOM and Java RMI. For DCOM, there is only very little online tools support. Although there is an integrated debugger with most development platforms, there are no tools specifically designed for distributed applications. For Windows 2000, a tool named *AppCenter Server* coping with monitoring and management issues is announced, but not yet available [Ewa2000].

For Java RMI, the lacking tool support can be explained with its increasing convergence towards CORBA. With *RMI over IIOP* [IBM2000], is is possible to connect RMI-based applications to CORBA environments, hence allowing RMI components to be monitored and managed with CORBA tools to a certain degree. *RMI over IIOP* relies on the Java-to-IDL [OMG1999] and Objects-by-Value [OMG1998b] efforts of the OMG, tackling with the problems of mapping Java interfaces to CORBA IDL and introducing the call-by-value semantics to CORBA.

3.2.2.7 Summary

With the described systems for monitoring and managing DOC platforms, we have shown a great variety of tools and functions useful for DOC environments. Among the interesting tool functionality are basic features like visualisation, performance monitoring, load distribution, and interface testing, as well as advanced functions like load testing, access control and accounting, and generic event-action patterns.

The usage of standard techniques like SNMP or MIBs is a benefit of some systems that enables the interaction with other management systems, although this is not sufficient for building both development and deployment tools. Specific solutions, for example OrbixManager's approach of keeping proxies for single threaded clients, are useful but not easily transferable to other platforms. Other techniques based e.g. on IIOP are more general, but still limited to CORBA or RMI systems. Nevertheless, the idea of building tools on standard protocols is a step towards open tool environments.

From the architectural point of view, all the approaches follow the general structure of tools, monitoring system, and instrumentation, although this separation is mostly implemented in a proprietary way and does not allow the insertion of additional components. The fusion between tools and monitor on the one hand, or monitor and instrumentation on the other hand is the major drawback of all investigated systems. Here, the demand for improvement is given in order to cope with the characteristics of heterogeneous middleware.

3.2.3 Enterprise Tools and Standards

There is a large variety of monitoring and management tools for enterprise middleware, resulting on the one hand from the variety of enterprise middleware itself, and from a large number of third-party vendors offering tools for any application scenario on the other hand. Because of the large variety of products, we have picked two characteristic representatives to illustrate the concepts of enterprise monitoring

and management frameworks. Our proceeding does not cover the whole spectrum of available products, but is sufficient for our middleware-oriented point of view of this category.

3.2.3.1 BMC Patrol

The BMC Patrol [BMC2000] product line comprises a large set of tools in the field of distributed application management. In our context, one of Patrol's key concept called *knowledge modules* is of interest. Knowledge modules are application specific components used to connect various operating systems, networks, network management systems, middleware, or applications to the Patrol platform. For example, knowledge modules for common operating systems like Unix, OS/390 or Windows NT are available, as well as for enterprise middleware like MOM products or Transaction Processing Monitors or common database systems like Oracle or Informix.

The architecture of BMC Patrol is agent-based, where the agent on the managed system makes use of the respective knowledge module in order to be able to manage its components. Tools for a variety of tasks depending on the actual environment are provided with Patrol, including standard tools for visualisation or performance management as well as domain specific tools for the knowledge modules.

Evaluation. BMC Patrol is a very powerful set of tools for enterprise environments. Its functionality comprises a lot of areas, from low-level network management up to high-level database management. However, the consequence is an enormous complexity that only pays off for complex environments. Despite its concept of separating knowledge modules from the core system, Patrol is a very heavy-weight product that requires high effort to be deployed. Moreover, the focus of Patrol tools is on the deployment phase only, whereas development tools are of little interest for Patrol. As for most commercial products, the interfaces of Patrol are not open (except an SNMP gateway), which prohibits an easy extension to other middleware platforms not supported by BMC. However, this would be of particular interest for integrating own, non-standard middleware that will not be supported by BMC because of their minor significance. Nevertheless, the overall approach of application specific knowledge modules can be seen as a smart possibility for integrating heterogeneous components.

3.2.3.2 LoadRunner

Mercury Interactive LoadRunner [Mer2000] [Kac1999] is a load testing tool for distributed enterprise environments. Similar to the SilkPerformer tool described in section 3.2.2.5, LoadRunner is built to produce load for testing servers, but - in contrast to SilkPerformer - is not limited to CORBA platforms.

The architecture of LoadRunner is based on a controller used to set up, configure, and start load tests. With a virtual user generator, different user characteristics can be defined and configured to reflect specific profiles. During the load tests, performance data are collected and can be analysed off-line after the completion of the tests.

The advantage of LoadRunner is its availability for diverse computing platforms, including general purpose protocols like CORBA-IIOP, FTP, HTTP, enterprise middleware like TPMs, and domain specific software like SAP or Oracle.

Evaluation. For LoadRunner, the same arguments as for BMC Patrol apply. LoadRunner is a proprietary tool available for a large set of domains, which only pays off for complex environments. Its benefit is the support of a large number of communication protocols, but the lack of a generic interface for connecting own protocols limits its general usability.

3.2.3.3 Application Response Measurement

Due to the difficulties resulting from proprietary approaches for monitoring and managing enterprise environments, attempts to create a vendor-neutral interface for observing business transactions have been undertaken. Starting as independent projects at Tivoli Systems and Hewlett Packard, the activities were merged into a working group for defining a general Application Response Measurement (ARM) interface. The resulting API is now standardised by the Open Group [Gro1998], and free software development kits are available, too.

Architecture. ARM's architecture consists of *ARM agents* and the management applications making use of these agents. ARM agents run on every participating host and consist of two parts: An *ARM API subagent* is dynamically linked to the instrumented application and runs within the address space of the application. This subagent communicates through inter-process communication mechanisms with its local *ARM processing subagent* that runs in another process. The management applications can then contact the ARM processing sub-agents to retrieve information.

The ARM API and implementation. ARM specifies the two APIs ARM V1 and ARM V2 [Joh1997a]. Version 1 contains basic functionality and was later extended to version 2 providing enhanced functions. Basic functionality covers initialising the agents, defining applications and transactions, and generating start- and stop-events for transactions. Version 2 allows to ship additional information, including parent/child relationships for nested transactions, or further data describing transactions more exactly.

The API reflects the general design considerations for ARM, which are simplicity, low runtime overhead, and extensibility. Also, an implementation of the API should not use extra threads for communication with the local processing subagent, as this would increase program complexity and introduce a variability in the measurements due to unpredictable thread processing behaviour.

In addition to the ARM API, the standard also defines a data format for the ARM operations. This is necessary for enabling management applications to interact with all kinds of ARM agents, independent of the respective vendor.

Use Cases. There are several use cases for which ARM is appropriate. The simplest case of monitoring beginning and end of single transactions is straightforward. A more demanding task arises when nested transactions are considered: In

this case, parent/child relationships can be used to extract information about the resulting hierarchy of transactions. This hierarchy can be useful for analysing nested transactions and by determining the relationships between sub-transactions.

ARM's technique to detect these correlations between transactions is simple but powerful: When a transaction is started, the ARM agent assigns a *correlator* to this transaction and returns it to the caller. When this parent transaction initiates a child transaction, the correlator is passed with every request in this transaction. A management application can collect all the correlators and reconstruct the transaction hierarchy from these data. In practice, this technique can be used efficiently by turning on correlators only for specific transactions that have to be monitored. Every time a child transaction receives a request with a correlator, it propagates the use of correlators, thus spreading the correlators through the complete hierarchy.

Other use cases for ARM include performance analysis by sending performance data with transaction events, or supervision of transaction success by sending diagnostic messages with transaction-stop events.

Evaluation. The ARM idea for monitoring transactions is a simple but powerful instrument for observing transaction-oriented systems. Its benefits are simplicity, flexibility, and openness. From a general point of view, the drawbacks are its limitation to observation only (as no manipulations are possible), its orientation to transaction-based applications, and the missing higher-level architecture describing patterns for the collaboration between management applications and agents.

However, basic ideas of ARM – especially the correlation mechanisms – represent intelligent concepts that can also be applied to general middleware monitors.

3.2.3.4 Summary

We have seen that monitoring and management systems for enterprise middleware are often large, heavy-weight systems with a far reaching functionality tightly adapted to existing platforms and applications. From our general point of view, the main drawback of such systems is the missing openness to integrate other middleware or build own tools. Other approaches like ARM, in contrast, show that openness and flexibility are the key issues for successful tool development for any kind of middleware. Basic approaches adopting various applications, protocols, or platforms within one tool environment are therefore a key concept for building tools for heterogeneous systems.

3.2.4 Network Management Approaches

Another important category of on-line monitoring systems are network management systems. On the one hand, network management approaches are historically relevant as they have been existing for a long time, starting from early local area network management and growing to large telecommunication network management systems. On the other hand, network management is still an ongoing research topic, as network management systems tend to merge with application management systems, resulting in an *integrated management* approach.

From the monitoring perspective, network management represents a form of on-line monitoring where similar problems like distribution, synchronisation, and heterogeneity come up. Although those issues are regarded in a slightly different context, the general principles remain the same.

In the following, we introduce major network management concepts and standards that are relevant from our monitoring-oriented view. Network management is a very voluminous topic, thus this overview cannot cover the complete network management area extensively as this would go beyond the scope of our focus.

3.2.4.1 Management Architectures

Basically, network or integrated management approaches are defined using management architectures describing a framework for management relevant aspects ([HAN1999], chapter 4). These aspects are usually classified into four categories:

- The *information model* defines the methods for modelling and describing managed entities.
- The *organisational model* determines the roles of actuators within the management system and their ways of cooperation and interaction.
- The *communication model* defines communication concepts, including communicating partners, protocols with syntax and semantics of management information, and integration into the overall framework.
- The *functional model* splits management functionality into sub-categories, for example the management functions explained in section 3.1.3.

The coarse structure of management architectures can be recognised in many existing management systems, although they are often not visible to the users, who only access the management system through a standardised management API.

3.2.4.2 OSI-Management and TMN

The OSI-management framework is defined within the ISO-OSI reference model [ISO1994] [ISO1989] and represents a reference architecture for management systems based on the four categories introduced above. OSI-management relies on the OSI layer model to describe the management-relevant aspects. Its information model is completely object-oriented, which allows for a comprehensive structural description of management information bases (MIBs). The organisational model assumes a distributed cooperative management and defines manager and agent *roles* for actuators within management systems. The communication protocol recognises three communication mechanisms for systems management, layer management, and protocol management, and the functional model classifies functionality into the five groups mentioned in section 3.1.3.

Telecommunications Management Network (TMN) is a management architecture explicitly tailored to manage public networks (PSTNs). It is standardised by the International Telecommunication Union (ITU) [ITU1997] and closely related to

the OSI-management approach. Important aspects of TMN are its orientation towards large and complex networks, and the introduction of a separate management network called TMN-overlay network.

3.2.4.3 Internet Management: SNMP

The most popular management architecture is the Internet management architecture, often designated as SNMP-management because of its Simple Network Management Protocol (SNMP). Internet management and SNMP are standardised by the IETF RFCs 1155, 1213, and 1157 [RM1990], [MR1991], [CFSD1990].

The information model (*structure of management information* SMI), exists in the two versions SNMPv1-SMI and SNMPv2-SMI, with version 2 being an extension of version 1. Internet management uses a client/server model for the propagation of management related data: *Management agents* provide the collection of information or manipulation of managed resources, while the *management station* (or manager) represents the management application interacting with a set of agents.

A critical aspect of management systems is the description of the properties of the managed resources. For this purpose, Internet management uses so called Management Information Bases (MIBs). Every agent keeps an *agent-MIB* containing variables describing characteristics of its managed node. These variables are called *management objects* (in a non-object-oriented sense). The structure of an agent-MIB is determined through the general *Internet-MIB* that lists all management objects, their meaning, and how the are identified. For vendor-independent usage of such MIBs, it is necessary to uniquely define the Internet-MIB. Therefore, a global identification tree identifying all management objects in a hierarchical manner has been standardised.

As mentioned above, the communication protocol between agents and manager is SNMP. With SNMPv1, only four operation types are defined:

- Reading information from an agent is implemented using a `get-request` operation that can be issued by a manager. As a parameter, only the respective MIB-entry needs to be specified.
- Stepping through an agent-MIB can be carried out with `get-next` operation, which provides the next entry from the agents MIB.
- Write access to agent-MIBs is granted with a `set` operation.
- Asynchronous events can be tracked using the `trap` operation, which notifies the manager in case the given event occurs.

In addition to the available operations, SNMP also defines the exact format of messages being exchanged between agent and manager. SNMPv2 extends SNMPv1 by adding encryption mechanisms and further request types like e.g. bulk requests allowing to retrieve larger amounts of data with one request. However, due to its complicated security mechanism, SNMPv2 could not be established for general usage. Currently, a new SNMPv3 standard unifying different endeavours is being undertaken, but not yet finished.

3.2.4.4 Summary and Evaluation

Network and integrated management approaches have shown to be very flexible, but large-scale and heavy-weight concepts for managing distributed environments. Their focus is still on network aspects, although application integration becomes more and more important. From our on-line monitoring perspective, a disadvantage is the concentration on deployment tools only, since development aspects are only of minor interest for management systems.

Moreover, the network management approaches are often too heavy-weight to be easy to use and to develop tools rapidly. The SNMPv1 approach is an exception as it is very general through the usage of standardised MIBs and the simple API. However, advanced tool functionality may not be implemented with SNMP, for example when synchronisation of distributed events is required; in this case, a lot of work is left to the tool developer.

3.3 Conclusion

Starting with a definition of on-line tools, on-line monitoring, and management, we have given a general introduction into the area of our interest in this chapter. After illustrating criteria for classifying tools and introducing basic monitoring mechanisms and problems, we have proceeded with a survey of common on-line monitoring approaches and implementations.

The systems analysed in our survey exhibit a wide range of monitoring aspects, which we have organised according to the kind of middleware they are built for. Similar to the previous chapter, tools and monitoring systems can be classified into the categories tools and monitors for parallel programming, distributed object systems, enterprise middleware, and network management.

Another attribute for classifying the described systems is the type of their approach. Some approaches represent specifications or standards for monitoring, while others are more or less proprietary solutions for specific purposes. In the category of specifications, we have the following approaches:

- OMIS
- ARM
- OSI-Management/TMN
- SNMP

Mostly, these systems are higher level concepts emphasising the integration aspects of distributed environments; in this sense, OMIS represents an exception within this category. The category of proprietary approaches contains the following systems:

- Coral
- CORBA Assistant
- AppMan
- OrbixManager

- Inprise AppCenter
- Silk for CORBA
- BMC Patrol
- LoadRunner

These products fall mainly into the category of distributed object systems and enterprise middleware. For DOC, this is reasonable as the field is rather new and no standardisation efforts have been completed yet; for enterprise middleware, the solutions have had a long history and have grown while being deployed within enterprises, where no interest in generalisation is given. Nevertheless, these products are not completely proprietary and provide communication capabilities with other systems, e.g. through SNMP support.

From the analysis of the various monitoring approaches, the question about the origins of this diversity arises. A main reason lies in the diversity of the main focus of interest for the different monitoring systems. The diversity itself is not the fundamental problem, but leads to a different *granularity* of monitoring for the respective middleware, from which different approaches for monitoring are derived. For example, while for parallel programming environments, non-intrusiveness and high performance are key issues, this is not a main aspect for network management, where time is considered with a more coarse granularity[2]. Considering the main entities of the analysed systems, the granularity shown in Table 3.1 results.

Monitor	*Granularity*
OMIS, Coral	OS Processes
CORBA monitors	CORBA Objects
Enterprise monitors	Transactions
Network management	Managed objects

Table 3.1 Granularity of Monitoring Systems

While parallel programming tools exhibit the finest granularity residing on operating system process level, main entities for other systems are becoming more and more coarse, growing from CORBA objects to transactions and managed objects (which often cover hardware entities like machines or network components). Of course, these values do not hold for any monitor, and exceptions do exist, but a general trend can certainly be derived.

Concluding with this analysis, the question remains why there is no generic monitoring system capable of dealing with all kinds of middleware. This issue will be our guideline for the next chapters, where we will present a novel approach for dealing with the described problems.

[2]This does not mean that time is not important for network management at all, but in general, the required resolution is lower.

Chapter 4

Requirements for a Generic Monitoring Approach

In this chapter, we will establish requirements for designing a generic monitoring system. As we have seen in the previous chapter, several monitors for different kinds of middleware exist. These systems offer similar functionality and rely on similar concepts, but they are not interchangeable and none of them is universally applicable. Hence, in this chapter we will provide the foundation for a generic monitoring system that is able to deal with different kinds of middleware and that allows to develop different types of tools.

Starting from the analysis of existing tools and monitors, we will expose the drawbacks hindering them from providing a foundation for a generic monitoring system. Subsequently, we will elaborate key criteria that have to be fulfilled by a generic monitor. These criteria define a basis for the design and implementation of a new monitoring infrastructure that will be illustrated in the following.

4.1 Drawbacks of Existing Systems

In part, the disadvantages of specific monitoring systems have been explained in the previous chapter. Here, we will summarise and substantiate these shortcomings in a systematic way.

4.1.1 Proprietary Solutions

A major drawback of many monitors is that they are proprietary solutions. Such systems are developed by vendors of specific middleware platforms, tailored to the requirements of that platform and its foreseen application domains, and can hardly be applied to other scenarios. The main reasons for these limitations are as follows:

- Interfaces used within the monitoring system are not open, i.e. they are not documented and not publicly available. Mostly, interfaces between tools and monitor as well as interfaces between instrumentation and monitor are affected by this fact. Therefore, communication with other monitors or applications is hardly possible.

- Communication within the system is not based on common communication standards or protocols. Here, the usage of non-portable and operating sys-

tem dependent protocols (e.g. relying on binary data encodings) makes the integration of other information sources hard.

Such limitations do not hold for all existing systems, as standards like ARM or network management approaches are often based on general definitions. Nevertheless, for many other systems like most CORBA monitors, these restrictions definitely hold.

4.1.2 Homogeneous Scenarios

The second shortcoming of most monitoring approaches is their restriction to homogenous systems only. Monitoring systems and tools concentrate on the kind of middleware they are built for and are not able to handle data coming from other middleware platforms. Hence, the drawbacks in this case include two aspects:

- First, the monitoring systems cannot be adapted to be used with other platforms than those they were originally developed for.

- Second, monitoring systems are not able to observe more than one middleware platform simultaneously.

This lacking adaptability results from a missing general information model on which the monitor would have to be constructed. An appropriate information model has the task of abstracting from the concrete characteristics of monitored entities in order to enable a more general classification of collected data.

Moreover, the homogeneity of monitoring systems is also reflected in the tools provided with the monitors. Due to the drawbacks of the monitoring systems, tool functionality is consequently limited to specific functions, too. Hence, these tool functions are closely related to the middleware platform and predefined application scenarios.

4.1.3 Simple Scenarios

Except for the network management approaches, most other monitoring concepts only cover simple scenarios and environments: Monitored environments only involve one single application that may consist of a set of distributed, but uniform components.

More complex scenarios like three-tier architectures being executed in a heterogeneous environment that dynamically includes different operating environments are not considered. The reason for this shortcoming lies in a more or less static layout of the complete monitoring environment that does not allow for easy and scalable integration of new components. Furthermore, the observation of several independent applications or components within a single environment is not covered, or at least only one application per tool can be handled as tools are not capable of handling multiple applications at a time.

4.1.4 Missing Interoperability Support

Another important disadvantage of common monitoring approaches is a lacking support of interoperability concepts. In our case, interoperability support covers two distinct concepts:

- First, interoperability can be seen in the field of heterogeneous components interacting with each other. A current example for this case are mixed CORBA- and DCOM-applications communicating through a CORBA-DCOM bridge. Here, interoperability support for the monitoring systems denotes the ability of the monitoring system to observe this “inter-middleware” communication.

- Second, interoperability support can be seen in the sense of *interoperable tools*, i.e. tools simultaneously working on a single application or component while interfering with each other in a controlled manner (see section 2.1.5).

Both kinds of interoperability are not supported by most monitoring approaches. From the systems described in the previous chapter, only OMIS/OCM implements tool interoperability, while only ARM and the network management approaches are in principle able to handle application interoperability.

4.1.5 Inflexible Architecture

A final, but nevertheless decisive drawback concerns the architecture of the monitoring systems. From the architectural point of view, none of the monitoring approaches shows a clear separation between tools, the monitoring system core, and the instrumented applications. Only OMIS/OCM presents a well-defined tool-monitor interface and the possibility to extend the interface, but all the other systems are either monolithic implementations or do not treat architecture details at all. However, a clear design of the core monitoring system is a crucial aspect for several reasons:

- From the tool’s point of view, well-defined and generic tool-monitor interfaces provide a facility for implementing different kinds of tool functionality without influence on the monitoring system (see OMIS/OCM).

- From the observed applications point of view, generic instrumentation-monitor-interfaces provide a means for the easy and flexible integration of new middleware platforms.

Hence, taking these two aspects together would result in a flexible and extensible monitoring approach. However, current systems do not show this kind of interfaces, but provide a heavy-weight and monolithic monitor that does not allow the easy integration of new middleware or the flexible building of new tools adapted to any kind of middleware.

The flexibility of using only parts of the monitoring system would allow to observe only interesting aspects of an application and hence improve performance, while extensibility would make it possible to add new middleware and therefore improve reuse capabilities of the monitor core.

4.2 Criteria for a Generic Monitoring Approach

From the drawbacks of existing systems and approaches, a set of requirements for a generic monitoring system can be derived. Here, we are listing the requirements in an abstract way, while a concrete implementation will follow in the next chapters. Altogether, our requirements aim at the construction of integrated tool environments for complex middleware systems [RWMB1998], [ZRL1999].

4.2.1 Systematic Concept for Complex Environments

Most monitoring approaches suffer from a lacking systematic concept for dealing with complex environments. Complexity in middleware applications occurs in many shapes:

- Firstly, complexity can be caused by a large number of components that have to be observed by a monitor. In this case, scalability issues gain relevance as the monitoring overhead should not increase disproportionately for performance reasons. A solution for the scalability problem is to introduce a hierarchy for components within the observed system, which enables a systematic classification of monitored entities and allows to generate definable views on the overall application system.

- A second source of complexity comes from the observation of multiple applications with a single monitoring system. A monitor therefore has to be able to handle different application architectures and implementations. For example, software architectures like two- or three-tier applications have to be monitored as well as integrated legacy components that may be implemented differently and show a completely distinct software architecture.

As a consequence of this complexity of monitored environments, an implementation of a monitoring system should be based on a monitoring infrastructure that defines a generic monitor core and enables a flexible usage within complex environments.

In addition to a suitable monitor core, a *monitoring framework* for complex environments is essential. When building tools on the basis of a generic infrastructure, a framework [Joh1997b] defining the core components and access and usage patterns is required for the implementation of tools and the integration of any kind of middleware. Furthermore, relying on a monitoring framework, the definition of a *tool development methodology* is useful for enabling a rapid and efficient design and implementation of tools. This development methodology can be seen as a recipe for constructing tools for a given middleware using the given monitoring infrastructure. Its target is to establish a process for rapidly adding new tools and middleware to an existing monitoring environment without interfering with other running tools. Up to now, none of the existing monitoring systems has offered a precise tool development methodology that provides the possibility for a flexible tool development and middleware integration. Therefore, the tool development methodology that will be substantiated in more detail in chapter 7 represents an important step towards a more general and efficient tool development process.

4.2.2 Tool Support for all On-Line Phases

Current monitoring approaches mostly concentrate on tools for specific phases of the software lifecycle. Therefore, they are not suited for the implementation of tool functionality belonging to other phases, although no significant differences between these phases exist. Figure 4.1 shows an overview of the software lifecycle for parallel and distributed applications (see [Bod1995], [Lud1998]). For on-line tools, only the late development (debugging, test, etc.) and deployment phases are relevant.

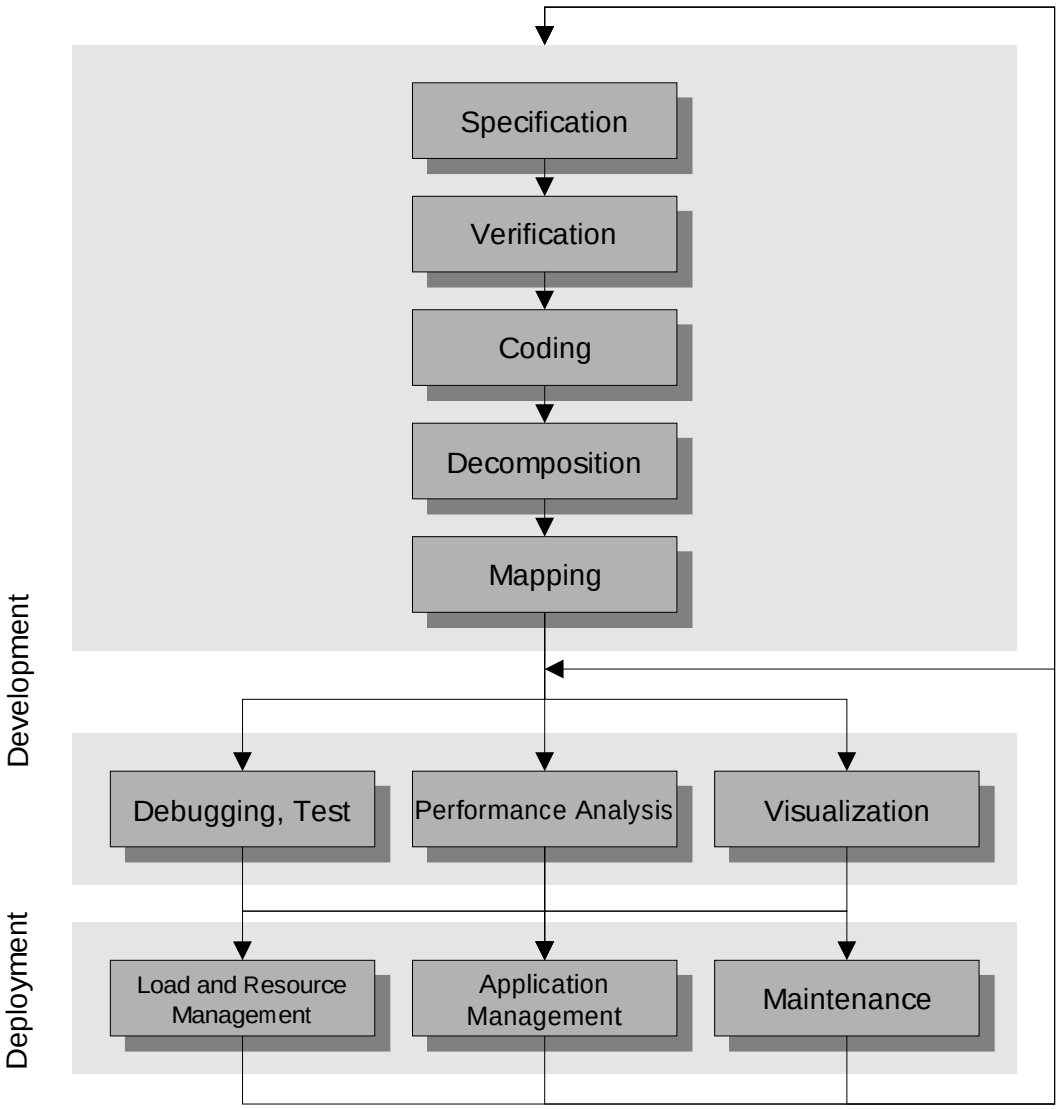

Figure 4.1 Lifecycle for Distributed Software Development

A major requirement for a generic on-line monitoring approach is to support both tools for the late development phase and the subsequent deployment phase. As a consequence, the structure of the monitor has to enable observation of entities with a different *granularity*. While during the development phase, fine granular entities

are of interest e.g. for debugging or performance tuning, in the deployment phase a more coarse granularity observing application components or complete applications is of importance. In other words, this means that a generic monitoring systems has to be laid out for low-level development tools as well as for high-level deployment tools. Using a monitor with such capabilites simplifies software development and deployment enormously as all kinds of tools can be implemented within a uniform infrastructure, which makes programming and operation easier.

Another aspect of supporting both the development and deployment phases is the ability to integrate classical monitoring functions and management functions as described in section 3.1.3. The only problem that remains to be solved is the complexity arising from the support of both lifecycle phases. But, as already explained in the previous section, a hierarchical view on the observed system in combination with a generic monitoring infrastructure integrates well into these ideas. Thus, a separation between development- and deployment-related data can easily be made through a hierarchical view on the system, which makes it possible for tools to concentrate on relevant facets without being bothered by other data.

4.2.3 Cover System Heterogeneity

A significant aspect for monitoring middleware environments is to take into account the existing system heterogeneity. As already mentioned, heterogeneity occurs in a variety of shapes, including the following aspects:

- Hardware
- Operating systems
- Programming languages
- Middleware platforms
- Application-specific protocols and standards

When heterogeneous environments are monitored using a single monitoring system as explained above, it has to be assured that heterogenous platforms can be monitored *simultaneously* with a single instance of the monitoring system. Consequently, the monitoring system has to be able to handle heterogeneous components simultaneously and present the respective data to tools. Tools themselves can either concentrate on a single middleware platform and ignore heterogeneity, or be built for usage within heterogeneous environments.

As a key criteria for simultaneous monitoring of heterogeneous components, the usage of a *general information model* for middleware-based systems results from the consideration of the various data types coming from different middleware platforms. The justification for an abstract and general information model derives from comparability and interoperability considerations, as comparability on the one hand is necessary to correlate data from different platforms, and interoperability allows to monitor inter-middleware communication. From the conceptual point of view,

the information model has to be general enough to comprise all possible characteristics of observed entities and allow a mapping of its abstract entities to concrete instances of the respective middleware platforms. Moreover, a hierarchical structure of the information model is obvious because of its ability to advantageously classify information within complex systems.

Another important feature of a general information model is its contribution to the aspect of integrating the classical monitoring and network management "worlds" illustrated before. Due to the fact that network monitoring approaches are frequently based on information models defined e.g. through management information bases (MIBs), introducing a similar model adapted for middleware platforms and covering classical monitoring as well as management functions represents a step towards an integrated monitoring and management concept.

Finally, heterogeneity also comprises the aspect for the monitoring systems itself to be applicable within a heterogeneous operating environment, i.e. to be executable on various hardware and operating systems.

4.2.4 Flexibility and Extensibility

In addition to the concepts presented up to now, flexibility and extensibility are important criteria for a generic monitoring approach. Concerning flexibility, two main features have to be taken into account:

- At first, the core monitoring system has to be designed in a way that does not restrict the type and amount of data to be retrieved from an observed system. Hence, the semantics of measured data should not influence the monitor core at all, except for basic data that are essential for fundamental monitoring tasks. As an example, events indicating the generation or destruction of monitored entities can be essential for the overall operation of the monitor, but application specific data should be propagated to interested tools without intervention of the monitor.

- Secondly, it has to be ensured that multiple tools can be applied simultaneously to an observed environment. This includes independent tools and applications as well as tools operating on the same monitored entities and therefore possibly influencing each other. Here, interoperability concepts have to be taken into consideration, allowing tools for example to communicate with each other to synchronise their actions or to inform each other.

Bearing these features in mind, a monitoring system is prepared to observe most imaginable scenarios. By touching as little semantics as possible within the monitor core, it is e.g. enabled to observe applications based on middleware as well as the middleware platforms themselves with a single system, but using different tools. The simultaneous application of these tools then allows to gain deeper insight into the whole environment.

Regarding extensibility, there are further criteria a generic monitor approach has to fulfil:

- On the one hand, extensibility with respect to the observed middleware has to be given. This can be ensured by using a general information model and generic interfaces for instrumentation-code accessing the monitor core. A general information model can be used to map and classify any kind of middleware to the abstract model, while generic interfaces provide the technical integration of middleware platforms. Hence, the combination of the information model and generic interfaces enables an easy and efficient integration of new middleware platforms, also during ongoing deployment of the monitoring system.

- On the other hand, extensibility regarding the tools making use of the monitoring system has to be guaranteed. At first, this implies the possibility to dynamically add new tools to the monitoring environment. Like before, this feature can be accomplished by defining a general tool-monitor interface that acts as an entry point for all tools. However, in practice it is often required to enrich existing tools with additional functions. In this case, it is advantageous to build tools on another, separate tool framework that is able to add new tool components easily to existing tools. Again, the process is simplified if all tools are based on the general information model defined by the monitor core.

Summing up, it can be concluded that flexibility and extensibility can be reached by making the monitor core as generic as possible, and thereby preserve a simple access and usage through defined access interfaces and a general underlying information model.

4.3 Summary

Based on the preceding discussions that have been initiated by an analysis of drawbacks of existing monitoring systems, our main requirements for a generic monitoring approach can be summarised as follows:

1. Definition of an infrastructure and access framework in order to handle system complexity.

2. Support of all on-line lifecycle phases by enabling the monitoring of entities with different granularity.

3. Coverage of heterogeneity by introducing a general information model that allows simultaneous monitoring of different platforms.

4. Achieve flexibility and extensibility that provides the possibility to dynamically add new tools and new middleware without affecting the monitor core.

Of course, it has to be kept in mind that the more general a monitoring approach is, the higher the overhead of monitoring gets. Therefore, a trade-off between performance and complexity of the monitor on the one hand, and the universality and functionality of the approach on the other hand has to be found. From the preceding

requirements, it can be concluded that our approach to handle this problem relies on the concept of defining a generic monitoring infrastructure: A small and light-weight monitor core with generic interfaces and a general information model serves as a foundation for integrating diverse middleware and building various tools. The development of tools and the adaption of new middleware is simplified by providing access and usage frameworks for the monitor core, and by defining a methodology for the development processes. With this approach, a configurable monitoring approach that can be customised for usage within various middleware scenarios is given.

In the subsequent chapters, we will illustrate the design and implementation of the MIMO monitoring infrastructure, which represents a concrete application of the general concepts acquired above.

Chapter 5

The MIMO Approach

This chapter introduces the *MIMO MIddleware MOnitor* approach for monitoring heterogeneous middleware. After having postulated a set of requirements for a generic monitor in the previous chapter, MIMO presents a concrete approach for implementing a monitoring systems that complies with the stated requirements.

Starting with the introduction of the *multi-layer monitoring approach*, one of the fundamental aspects of the overall MIMO approach is illustrated and enforced by a formal description. Additionally, a general monitoring infrastructure will be introduced, and the need for a methodology for developing tools and integrating new middleware will be substantiated. Altogether, these aspects establish a basis for the overall monitoring infrastructure and framework; details concerning the monitor architecture and its components will then be discussed more deeply in the subsequent chapters.

The basic idea of the MIMO approach is to define a versatile monitor core that only stores information concerning the current system state, i.e. data about the existence of monitored entities. More specific data, e.g. describing attributes of monitored entities can then be queried by directly contacting the instrumentation component that is responsible for a certain entity. The multi-layer monitoring model enables a structured and systematic classification of existing entities, while the monitoring infrastructure defines how further information can be queried from specific entities. The usage methodology supports a concerted development and integration of tools and middleware platforms.

5.1 Multi-Layer Monitoring

The core of the MIMO approach rests upon the multi-layer monitoring (MLM) model, which we will derive in this section. After the introduction of the abstract information model and its formal foundation, we will show the integration of concrete middleware platforms.

5.1.1 Hierarchical Information Model

As we have seen in the previous chapter, a common information model for the representation of monitored environments is needed to monitor heterogeneous middleware systems. Such an information model provides the basis for classifying information retrieved from observed platforms by defining a concept for the description

of monitored entities. From our requirements, we can derive a model that fulfils the following properties:

- The possibility to classify monitored entities with a different granularity and on different abstraction levels is given.
- Steps for the observation of complex environments are undertaken.
- The observation and classification of heterogeneous entities with a single model is made possible.

As a consequence of the first and second requirement, a *hierarchical information model* is appropriate because of its ability to handle complex systems in a clear and structured manner. From the third requirement, the need for a generic model that can be customised for usage within various middleware scenarios can be derived. Therefore, MIMO's information model is based on a hierarchical and generic model of the observed environment.

Obviously, the definition of a common information model the monitoring system is built on also influences the layout and usage of tools. In our case, the hierarchical information model suggests a hierarchical tool layout and usage; this means that tools should either operate on single levels of the information model, or exploit the information model's hierarchy when accessing multiple levels of the model.

5.1.2 Abstract Entity-Relationship Information Model

MIMO's description of the monitored environment relies on an *entity-relationship model* (see [Dat1995], ch. 12). In our scenario, *entities* represent "distinguishable things" that are being observed by the monitoring system. In this term, "distinguishable" expresses the fact of possessing a unique identity in our context, and "things" comprise all kinds of modelled objects including software- and hardware-components on different abstraction levels. Besides their identity, entities hold a set of properties (or attributes) containing further information. Several entities can be linked through *relationships* that define associations with specific semantics among entities.

To define a general model for middleware environments, it is necessary to determine relevant entities and their relationships. On the one hand, as the model should be applicable to all kinds of middleware platforms, a general model has to be *abstract* in order not to limit its scope of usage. On the other hand, entities from concrete middleware platforms have to be classified according to this model. Therefore, the MIMO approach defines an abstract information model that only contains as little restricting preconditions as possible and a procedure to map concrete middleware platforms to this abstract model.

The advantage of this proceeding is to enable the simultaneous monitoring of heterogeneous middleware because all platforms are mapped to a single abstract information model. An alternative approach would be the definition of a customised model for every platform; however, the disadvantage of these models would be the loss of generality, accompanied by with a missing comparability of entities from different middleware platforms.

5.1.3 The Multi-Layer Monitoring Model

From the described preconditions and the analysis of common middleware in chapter 2, MIMO's generic information model can be derived. However, as there still remains a certain degree of freedom, the following layers have been chosen with respect to practical monitoring tasks; hence, they should exploit as much information as possible from the observed environment, and thereby allow to concentrate on specific details without losing the monitoring context and system overview.

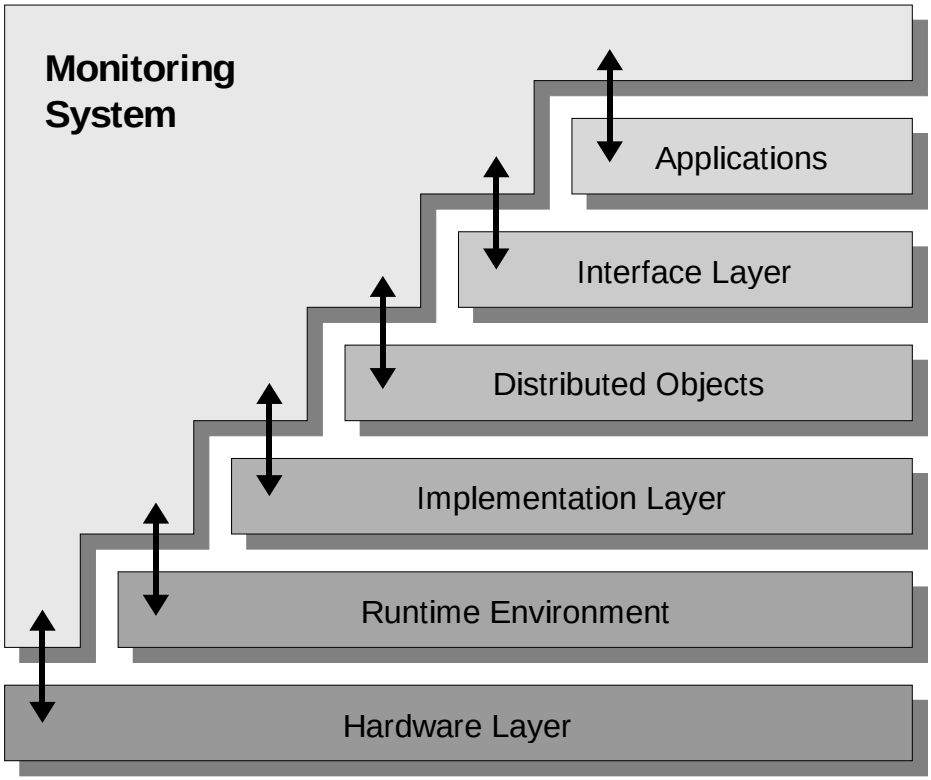

Figure 5.1 MIMO Multi-Layer Monitoring Model

Figure 5.1 illustrates the resulting multi-layer monitoring (MLM) model [Rac1999]. The hierarchical model consists of six layers representing different abstraction levels, into which information from the observed environment is classified. Within an application, the whole functionality offered by components is described by interfaces; these interfaces are defined in any kind of definition language that are represented by the interface layer. The subsequent implementation of the functionality described by these interfaces is effected by objects within the distributed object layer. These objects may still be considered as abstract entities residing in a global object space. In order to enable communication between the distributed objects, middleware is required, and in particular a mechanism to define and uniquely identify objects within the object space is needed. Hence, objects identified by these middleware-specific object identifiers are stored in the distributed objects layer. Furthermore, as objects on the distributed object level are still abstract entities, they need to be implemented in a concrete programming language. This implementation

of the objects is considered in the implementation layer. Finally, the implementation objects are executed within a runtime environment that can be an operating system or a virtual machine on top of an operating system that is being executed by the underlying hardware nodes; thus, the lowest two layers represent entities within the runtime environment and the underlying hardware.

As a consequence, for the tools making use of a MLM-based monitoring system, the following facts are relevant:

- Firstly, it is possible to gather data on all abstraction levels in order to serve as an information source for all kinds of on-line tools.

- Secondly, the mappings between the different layers are of great importance. As all entities within a specific layer are mapped onto appropriate entities within the layer on the next lower level until the hardware layer is reached, keeping track of these mappings is essential because the relationships between entities in two adjacent layers are not necessarily one-to-one relationships.

The tools making use of the monitoring system may be very diverse and therefore operate only on specific abstraction levels, while for other tools, mappings between layers can be of special interest. Our multi-layer monitoring approach closely reflects the structure of the middleware environment and is therefore well suited for both classes of on-line tools discussed above.

Considering the relationships between entities in adjacent layers, the abstract MLM-model expresses only weak assertions about how the different entity types are related to each other: Applications *contain* a set of interfaces that are *offered by* distributed objects. The objects are *implemented by* entities in the implementation layer, which in turn are *running within* runtime entities that are *executed on* specific hardware. Moreover, these relationships are not quantified, i.e. they have to be assumed as $m : n$ relationships. For concrete middleware, these relationships are specialised with the concrete entity types for this middleware. Hence, the abstract MLM-model in a first step introduces as little semantics as possible for the relationships in order not to limit the generality.

5.1.4 Formal Description

After having started with an informal description of the multi-layer monitoring model and its derivation, we now introduce a formal model to describe MLM-based monitoring systems and algorithms in more detail.

5.1.4.1 Basic Formalism

The MLM-model can be seen as a set of layers L containing information about existing entities within the observed system. In our case, the set of layers is

$$L = \{L_0, \ldots, L_{\Lambda-1}\}$$

where Λ is the number of layers, with $\Lambda = 6$ in for the MLM-model[1].

The *system state*, i.e. the set of all entities known to the monitor at a time is called S, with

$$S = (E, R)$$

where E denotes the existing entities within all layers:

$$E = \{E_0, \ldots, E_{\Lambda-1}\}$$

and R denotes the relationships originating from these layers:

$$R = \{R_0, \ldots, R_{\Lambda-1}\}$$

Evidently, for every Layer i, E_i denotes the set of entities within this layer:

$$E_i = \{e_{i,0}, \ldots, e_{i,n_i-1}\}$$

where n_i is the number of entities residing in layer i.

Moreover, R_i denotes the relationships for layer i:

$$R_i = \{r_{i,0}, \ldots, r_{i,n_i-1}\}$$

Here, every entity $e_{i,j}$ with $i \in \{0, \ldots, \Lambda - 1\}$ and $j \in \{0, \ldots, n_i - 1\}$ has a set of up- and down-relationships linking them with respective entities in the layers $i - 1$ and $i + 1$ (of course, for $i = 0$ there is no up-link and for $i = \Lambda - 1$ there is no down-link). Hence, the relationships $r_{i,j}$ for entity $e_{i,j}$ are given by

$$r_{i,j} = (U_{e_{i,j}}, D_{e_{i,j}})$$

where

$$U_{e_{i,j}} = \{e_{i-1,l_{i,j,0}}, \ldots, e_{i-1,l_{i,j,u_{i,j}-1}}\}$$

for $i = 1, \ldots, \Lambda - 1$, and

$$D_{e_{i,j}} = \{e_{i+1,m_{i,j,0}}, \ldots, e_{i+1,m_{i,j,d_{i,j}-1}}\}$$

for $i = 0, \ldots, \Lambda - 2$.

Here, $u_{i,j}$ and $d_{i,j}$ denote the number of up- and down-links for entity $e_{i,j}$; consequently, $U_{e_{i,j}}$ and $D_{e_{i,j}}$ are the sets of entities in the layers $i - 1$ and $i + 1$ being linked with $e_{i,j}$, and $l_{i,j,0}, \ldots, l_{i,j,u_{i,j}-1}$ and $m_{i,j,0}, \ldots, m_{i,j,d_{i,j}-1}$ are the indices of the linked entities in the upper and lower layers. In the following, we do not distinguish between the relations R_i for $i = 0, \ldots, \Lambda - 1$ with respect to their potentially different semantics; for the sake of simplicity, R only models a general relationship without reflecting the underlying semantics.

To illustrate the introduced notation, figure 5.2 shows a fragment of a set of entities and their links to the adjacent layers. The up- and down-links for entities $e_{i,j}$ and $e_{i,j+1}$ are marked by arrows to the respective entities in the layers $i - 1$ and $i + 1$.

[1]We continue to take Λ as the number of layers, although we never change the number of layers throughout this thesis.

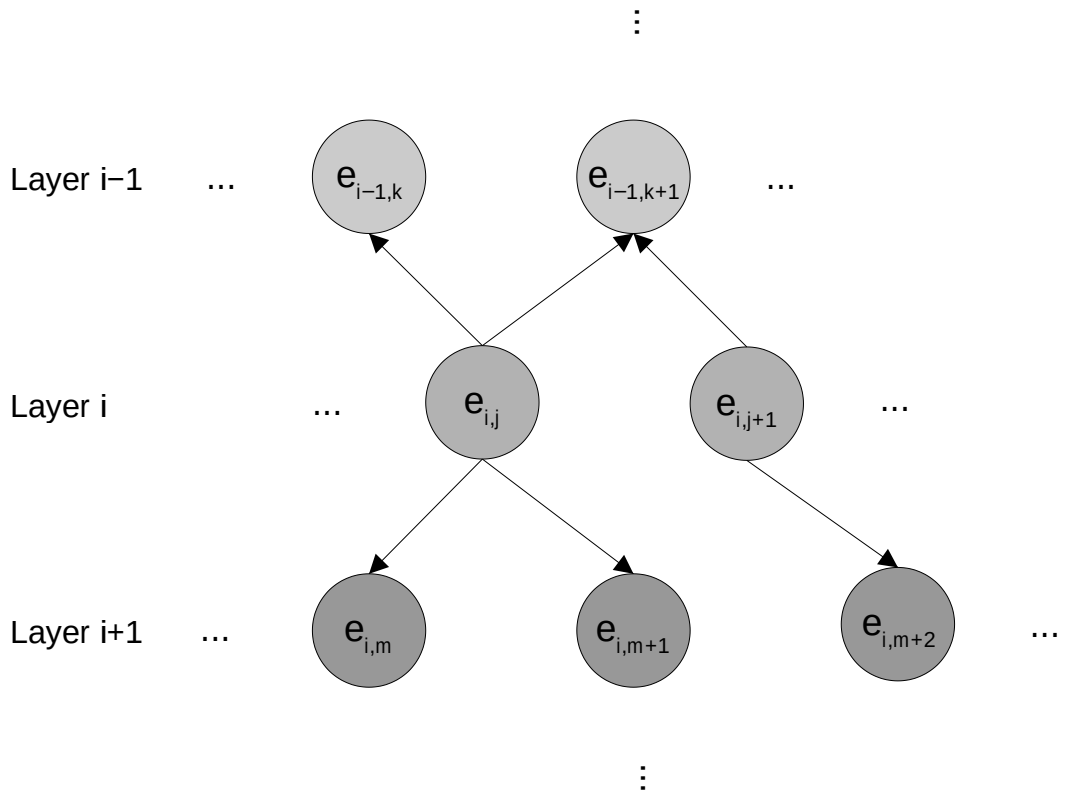

Figure 5.2 Entities and Relationships

As the MLM-model does not define a concrete semantics for the relationships between entities, the only statement that can be made is that relationships between entities are always *symmetric* by definition, i.e.

$$\forall \text{entities } e_i, e_j \in E : e_j \in U_{e_i} \Leftrightarrow e_i \in D_{e_j}$$

This property can be deduced from the general definition of the entity term and the layers; if it does not hold, the consistency of the system state is violated.

5.1.4.2 Relationships between Entities

Another aspect of our model is that all entities are *weak entities* in the sense of the entity-relationship model, because no entity can exist on its own without possessing at least one up- and one down-link (except those on the topmost and lowest layer, which do not have an up- or down-link, respectively). Consequently, a relationship $\mathcal{R}$ linking any two entities within two different layers of the model can be defined[2]. Two entities $e_{i,j}$ and $e_{k,l}$ are related through $\mathcal{R}$, if the following holds:

$$e_{i,j}\mathcal{R}e_{k,l} \Leftrightarrow \exists z = \{z_1, \ldots, z_n\}(n > 1)$$

with

$$z_1 = e_{i,j} \wedge z_n = e_{k,l}$$

[2] R only lists relationship between entities in *adjacent* layers.

and

$$z_{m+1} \in D_{z_m}(k > i) \text{ or } z_{m+1} \in U_{z_m}(k < i)$$

for $m = 1, \ldots, n-1$. Two entities within the same layer can never be in $\mathcal{R}$ as no relationship between two entities of the same type exists.

The computation of $\mathcal{R}$ is similar to the computation of the *transitive closure* ([Sed1988], p. 473ff) for a given relation; we just have to take a directed graph consisting of all entities and either the D or U relation for the edges. The difference to our case is that we do not need to compute the complete relation, but find all entities that are related to a certain starting entity in one given layer. Moreover, we could not store the complete relation anyway because it dynamically changes with the system state; to deal with this problem, a mechanism that incrementally updates the transitive closure would have to be found, but this would cause too much effort that cannot be justified by performance improvements. Hence, the complexity of $O(n^3)$ for computing the transitive closure combined with a dynamic update is not appropriate to solve our problem. In section 5.1.6, we will present an algorithm to compute $\mathcal{R}$ for a single entity, as well as other MLM-related algorithms.

5.1.4.3 Hierarchy Structure

Finally, it should be pointed out that the whole hierarchical information model does not necessarily represent a *tree structure*, because it is possible that entities on a lower level of the hierarchy are connected to more than one entity on a higher level. This is of practical relevance e.g. when instances of distributed objects offer more than one interface, or when several processes are executed on a single machine.

However, the system state described by the MLM model still represents a hierarchy: The graph G representing the system state results from entities being vertices and either the D or U relation being used for the edges (here, the usage of one of the relations D or U is sufficient because the information contained in D and U is redundant; we only use it for a more convenient description). As G is a directed, acyclic graph, a partial ordering of the entities is given, and G consequently represents a hierarchy.

5.1.5 Application to Concrete Middleware

The next step after formulating a formal foundation for the abstract information model is the application of this model to concrete middleware platforms.

5.1.5.1 Mapping of Concrete Middleware

Now that we have explained the multi-layer monitoring model and its formal foundation, we can go into detail with the integration of concrete middleware platforms and the MLM approach. The proceeding for monitoring a middleware platform is as follows:

- In a first step, the entities existing within the middleware need to be analysed and classified into a set of abstraction layers.

□ The resulting abstraction layers need to be mapped to MIMO's MLM model.

The result of the first step is a layer model C with p layers describing the relevant entity types of the concrete middleware μ:

$$C = \{C_0, \ldots, C_{p-1}\}$$

Obviously, C depends on the middleware platform and on the goals of the monitoring process, as the relevance of certain entity types corresponds to the tools processing the resulting data.

For the second step, a function M that maps C to MLM's layer model needs to be defined:

$$M_\mu : C \to L$$

with

$$M_\mu(C_i) = L_j$$

$\forall i \in \{0, \ldots, p-1\}$ and $j \in \{0, \ldots, \Lambda - 1\}$. To yield a valid mapping, M_μ has to be a *function*, i.e. map every C_i to a single L_j.

There are no other preconditions that M_μ has to fulfil, but the following cases should be mentioned for clarity:

1. $p < \Lambda$:
   As the number of layers in the concrete middleware is smaller than the number of layers in the MLM model, not all MLM layers can be filled with data. Thus, in this case empty layers emerge, which need to be treated properly; the proceeding for this case will be shown in section 5.1.5.3.

2. $p = \Lambda$:
   In this case, the mapping from C to L can be bijective, but does not necessarily have to be. If M_μ is bijective, every layer in C exactly corresponds to a layer in L, which is a standard case that can be applied conveniently.

3. $p > \Lambda$:
   If the number of layers in the concrete middleware is bigger than the number of MLM layers, more than one layer in C needs to be mapped to a single layer in L. Hence, M_μ cannot be injective anymore because at least two layers in C are mapped to the same layer in L. This case is not forbidden, but may lead to a more complicated system model.

In practice, only the first two cases are of relevance, while the third case should be avoided in order not to complicate monitoring tasks.

5.1.5.2 Example: CORBA

As a simple example for our theoretical background, the mapping of the CORBA middleware to our MLM model can be shown. Table 5.1 shows the definition of the CORBA layers and their mapping to the MLM model. The example is straightforward because our CORBA model shows six layers that can be mapped easily to the MLM model.

Multi-Layer Monitoring Model	*CORBA layer*
Application layer	CORBA applications
Interface layer	IDL interfaces
Distributed object layer	CORBA objects
Implementation layer	Object implementations
Run-time layer	OS processes and threads
Hardware layer	Nodes

Table 5.1 CORBA to MLM Mapping

An important aspect for the mapping is the unambiguousness of the entities in their respective context. This means that entities have to be uniquely identifiable within their layers. For CORBA, the identification of entities can be implemented as follows:

- Applications are identified by their name[3].
- CORBA Interfaces are uniquely described by their interface name that is derived from the IDL description.
- CORBA objects can be uniquely addressed by their *interoperable object reference* (IOR).
- Object implementations, processes, and threads can be uniquely identified by their memory address, process or thread identification number on the respective machine.
- Nodes are identified by their IP address.

The only problem arising here is that identifiers coming from two different middleware platforms must not interfere with each other. This is a general issue that has to be kept in mind during the deployment of the monitoring system; a simple heuristics to circumvent difficulties is to prepend identifiers with the name of the respective middleware platform (which is in general uniquely defined).

Of course, depending on the monitoring scenario, other models of CORBA applications can be defined. Our example represents a relatively general model of CORBA applications that can be used for most common monitoring tasks, without being specialised to any details of CORBA.

5.1.5.3 Handling Empty Layers

As mentioned before, empty layers in the information model can occur if a layer model for a certain middleware contains less layers than the MLM model. This case requires special treatment, because otherwise the structure of the relations between entities would be destroyed. Our formal model requires that every entity only keeps

[3]This identification by name is the only possibility for nearly all middleware platforms, but as the number of applications is generally small, this is not an issue.

references to entities in adjacent layers. If a layer is empty, we therefore have to introduce *dummy entities* that allow to skip the original MLM layer.

For example, assuming that a layer L_i is kept empty because there is no mapping from a certain middleware to L, a set of entities

$$c = \{c_0, \ldots, c_{i-1}, c_{i+1}, \ldots, c_{\Lambda-1}\}$$

has to be inserted into E. If so, a dummy entity $d_{c_{i-1},c_{i+1}}$ has to be inserted with c, yielding

$$c = \{c_0, \ldots, c_{i-1}, d_{c_{i-1},c_{i+1}}, c_{i+1}, \ldots, c_{\Lambda-1}\}$$

for the insertion into E. The dummy element links c_{i-1} with c_{i+1} and acts as a surrogate for the missing entity. Apparently, different dummy entities need to be generated for each different pair of entities in order not to modify the original relationships. If more than one layer is kept empty, all these layers need to be filled with newly generated dummy entities. With this proceeding, the case for $p < \Lambda$ can be treated properly without influencing the following algorithms.

5.1.6 Algorithms for Accessing the MLM

Based on the formal description of the MLM model, we can proceed to the fundamental algorithms for accessing the defined data structures. First, we will show how to insert and delete entities from the MLM model, and subsequently we will introduce a mapping algorithm that computes the $\mathcal{R}$ relation from section 5.1.4.2.

5.1.6.1 Insertion

Figure 5.3 gives the basic algorithm for inserting entities into the MLM model. The input for the insertion algorithm is a list of entities $e = \{e_{0,j_0}, \ldots, e_{\Lambda-1,j_{\Lambda-1}}\}$. It is not possible to insert single elements of e because the relationships between the entities need to be preserved. Nevertheless, the possibility to insert only parts of e is given, if the two adjacent entities for a given new e_{i,j_i} are included and already existing in the MLM model. This case is straightforward and similar to the algorithm described here, so we concentrate on the case of inserting entities for all layers of the MLM.

The proceeding of the algorithm is simple: It traverses all layers of the MLM and inserts the respective entity e_{i,j_i}, if it is not yet contained in layer L_i. Additionally, the relations for every entity are updated in a second step.

For simplicity, the algorithm does not contain arrangements for handling empty layers as described above. Nevertheless, in practice this problem is solved by keeping elements of the input empty for non-existing layers, while the algorithm has to introduce dummy elements for these elements. The complexity of the algorithm is linear with the number of layers. Inside the layers, the effort of the lookup operations depends on the internal representation of the sets, and can be implemented efficiently e.g. by using hashing mechanisms.

Algorithm **insertEntity**:
Input An entity list $e_{0,j_0}, \ldots, e_{\Lambda-1,j_{\Lambda-1}}$ to be inserted
System State $S = \{E, R\}$
Output Modified system state $S = \{E, R\}$

begin
  for $i := 0$ **to** $\Lambda - 1$ **do**
    // INSERT e_i, IF NOT ALREADY IN E_i:
    if $e_{i,j_i} \notin E_i$ **then**
      $E_i := E_i \cup e_{i,j_i}$;
      $U_{e_{i,j_i}} := \emptyset$;
      $D_{e_{i,j_i}} := \emptyset$;
    fi
    // ADD UP- AND DOWN-LINK:
    if $i \neq 0$ **then**
      $D_{e_{i-1,j_{i-1}}} := D_{e_{i-1,j_{i-1}}} \cup e_{i,j_i}$; // ADD DOWN-LINK
      $U_{e_{i,j_i}} := U_{e_{i,j_i}} \cup e_{i-1,j_{i-1}}$; // ADD UP-LINK
    fi
  od
end

Figure 5.3 Insertion Algorithm

5.1.6.2 Deletion

At first glance, the deletion of an entity from the system state is straightforward. A given entity $e_{i,j}$ is looked up in the elements E_i of layer i and removed, if found. But, in a second step, the relationships of the deleted entity also have to be removed. This concerns the entity's U and D relations that are deleted with the elements itself, and also the related entities D and U relations, which are redundantly stored. Here, the case of $e_{i,j}$ being the last related entity for an entity e in an adjacent layer can occur. If e has no more relationships to layer i, it has to be deleted because all entities are weak entities. Consequently, the deletion of e can be implemented by a recursive deletion call. An example for this propagation of the deletion process is shown in figure 5.4 for entity $e_{i,j}$; the bold lines mark the influenced entities that have to be removed as a consequence of $e_{i,j}$'s removal. The resulting deletion algorithm is illustrated in figure 5.5.

5.1.6.3 Related Entities

As already mentioned before, an algorithm for computing the $\mathcal{R}$ relation is required for conveniently querying the current system state. The problem is to compute all entities in layer l that are related to a given entity $e_{i,j}$. We have explained before that the problem is similar to the transitive closure that can be computed with Warshall's algorithm [AHU1975b], [AHU1975a]. But, as we operate on a dynamically changing relation and only need to get the related entities in a given layer and for a

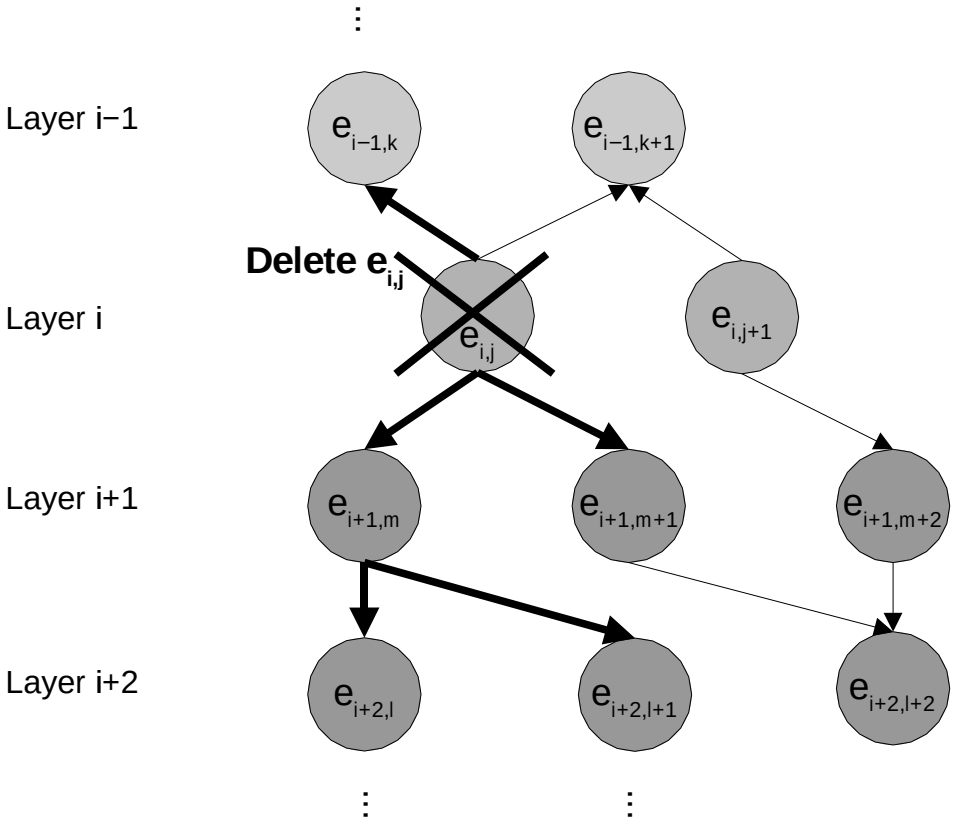

Figure 5.4 Deletion and Propagation Process

single starting element, our algorithm shown in figure 5.6 works differently.

As we have a given hierarchy consisting of several layers, we only have to traverse these layers and expand the relationships starting from our initial entity, until we reach the target layer. The only decision to be made at he beginning is whether to move upwards or downwards in our hierarchy, from then on it is sufficient to compute the set M_k of related entities in the respective layer k. When the target layer l is reached, M_l naturally contains the resulting set of related entities for $e_{i,j}$.

This algorithm represents one of the fundamental mechanisms for quickly and efficiently accessing information stored in the MLM model because it allows for a directed navigation through a complex system model, especially if only partial aspects of the overall system are of interest for certain tools.

5.1.7 Summary

In this section, we have introduced the multi-layer monitoring model, which represents an abstract, hierarchical information model that is tailored to our monitoring purposes. After an informal description, we have developed a formal description for the basic model, the integration of concrete middleware, and fundamental access algorithms.

These principles of the MIMO approach will be taken up again when the imple-

Algorithm **deleteEntity**:
Input An entity $e_{i,j}$ to be deleted
System State $S = \{E, R\}$
Output Modified system state $S = \{E, R\}$

begin
// REMOVE ENTITY FROM E:
$E_i := E_i \setminus e_{i,j}$;
$R_i := R_i \setminus r_{i,j}$;
// REMOVE DOWN- AND UP-LINKS TO DELETED ENTITY;
// CHECK WHETHER ENTITIES IN ADJACENT LAYERS NEED TO BE REMOVED:
// UP-LINKS:
if $i \neq 0$ **then**
foreach $e \in U_{e_{i,j}}$ **do**
$D_e := D_e \setminus e_{i,j}$;
if $D_e = \emptyset$ **then** $deleteEntity(e)$; **fi** // RECURSIVE CALL
od
fi
// DOWN-LINKS:
if $i \neq \Lambda - 1$ **then**
foreach $e \in D_{ei,j}$ **do**
$U_e := U_e \setminus e_{i,j}$;
if $U_e = \emptyset$ **then** $deleteEntity(e)$; **fi** // RECURSIVE CALL
od
fi
end

Figure 5.5 Deletion Algorithm

mentation of the monitoring system is described in detail.

5.2 Monitoring Infrastructure

The second fundamental concept of the MIMO approach is the MIMO monitoring infrastructure. From the requirements stated in the previous chapter, we address the following points with our monitoring infrastructure:

- Definition of a suitable framework that allows to handle the system complexity.
- Achievement of flexibility and extensibility that makes it possible to dynamically add new tools and middleware.

The main aspects for the definition of the monitoring infrastructure are the definition of a core framework, enhanced by general interface definitions, and a universal event model.

Algorithm **relatedEntitities**:
Input An entity $e_{i,j}$ and a layer l to which $e_{i,j}$ should be mapped
System State $S = \{E, R\}$
Output A set of entities M_l containing the related entities for $e_{i,j}$

begin
// NOTHING TO DO IF MAPPED TO THE SAME LAYER:
if $i = l$ **then**
$M_l := \{e_{i,j}\}$;
return M_l;
fi
// SET INCREMENT OR DECREMENT:
if $i < l$ **then** $inc := 1$; **else** $inc := -1$; **fi**
// GO THROUGH LAYERS AND BUILD MAPPINGS:
$M_i := \{e_{i,j}\}$;
for $k := i + inc$ **to** l **step** inc **do**
// EXPLORE UP- OR DOWN-LINKS OF ENTITIES IN ADJACENT LAYER:
$M_k := \emptyset$;
foreach $e \in M_{k-inc}$ **do**
if $inc = 1$ **then** $M_k := M_k \cup D_e$; // ADD DOWN-LINKS
else $M_k := M_k \cup U_e$; // ADD UP-LINKS
fi
od
od
// M_l NOW CONTAINS LIST OF MAPPED ENTITIES
return M_l;
end

Figure 5.6 Related Entities Algorithm

5.2.1 Core Monitoring Framework

The first element of the monitoring infrastructure is the definition of the core monitoring framework. A framework determines the main components of the monitoring environment and the respective usage patterns [Joh1997b], where the level of abstraction is higher than that for separate design patterns (which can be constituting elements of frameworks).

Our basic framework for the MIMO monitor consists of a three-tier model of the monitoring system as shown in section 3.1.2. A logically central monitor core provides an entrance point for tools and instrumented middleware. It offers appropriate interfaces and provides a correct propagation of incoming events.

The usage pattern for tools basically consists of three phases:

1. Attach to the monitor.

2. Send requests and get responses synchronously or asynchronously.

3. Detach from the monitor.

The detailed mechanisms and implementation possibilities for these steps are illustrated in more detail in the next chapter, where we describe the implementation of the monitor.

From the application side, the situation is not as trivial because applications are usually not laid out for being monitored, and thus mechanisms collecting data from them and interacting with the monitor core have to be found. The code taking over this part is called *instrumentation* and has to be integrated into the application. With our general framework, we cannot make any assumption about how the instrumentation gathers data from the application, how it potentially manipulates it, and how the code is integrated, because of the large amount of possibilities existing here. The only distinction we can make at this point is whether instrumentation is added transparently for an application or not. In case the application is instrumented transparently, we call the instrumentation component an *intruder*, and in case it is integrated non-transparently, we call it an *adapter*. Hence, assuming the existence of an intruder or adapter that is able to gather data from and influence the observed application, the usage pattern of the monitor core is analogous to the one for tools:

1. Attach to the monitor.

2. Send events to and receive commands from the monitor core.

3. Detach from the monitor.

Usually, attaching to the monitor is carried out when an application starts, and detaching indicates the termination of an application.

The monitor core itself is responsible for establishing the connection between tools and the instrumented applications; this includes the propagation of requests to the concerned intruder or adapter, as well as transferring back the events to the tools. Consequently, the information the monitor core has to store is state information about the observed environments, so that requests and events can be routed correctly to the concerned components.

5.2.2 Interfaces and Events

Up to now, no information has been given about how the communication between tools, monitor core, and instrumentation code is carried out. Basically, we have to distinguish between synchronous and asynchronous communication patterns. For synchronous communication, we can rely on interfaces with a request/reply semantics, where the requesting component is blocked until the result is passed back. For asynchronous communication, interfaces cannot be used as the requesting component proceeds with other tasks and results have to be passed by means of events; hence, an event-based communication mechanism needs to be introduced in addition to interfaces.

For reaching a flexible and extensible layout of the monitoring infrastructure, the interfaces and event models need to be generic in order to enable the integration of diverse middleware and the construction of different tool types.

5.2.2.1 Generic Interfaces

The monitor core offers interfaces for both tools and instrumented applications. The interface for tools is called the *tool-monitor interface* and has to provide the following basic operations:

- *Attach and detach operations:*
  As mentioned before, these operations are required to announce the existence of new tools or the termination of a tool. From the monitor core's point of view, tools need to have identifiers, e.g. for asynchronously passing back results.

- *Request operation:*
  This operation represents the only way for a tool to issue requests. It has to be possible to request certain functionality synchronously or asynchronously, depending on the usage case.

Even more importantly, the content of the request may vary from tool to tool, depending on concrete monitoring tasks. Obviously, the monitoring core can only be designed to answer a set of common requests concerning state information it has actually stored. As this information cannot be sufficient for tools, the proceeding is to issue requests for certain user-defined events, which are being generated from suited instrumentation code and passed to requesting tools without semantical processing by the monitor, which only forwards the events to interested tools.

As a consequence of the generic request-operation, the need for a generic event model originates. Hence, the interfaces only enable the interaction between tools, monitor core, and instrumentations, while concrete details are left to the event-based communication.

5.2.2.2 Generic Event Model

The event mechanism resulting from the above discussion is used for asynchronous communication between tool and monitor on the one hand, and instrumentation and monitor on the other hand. As requests coming from tools are issued synchronously because all activity originates from the tools, no asynchronous communication from the tool to the monitor core is required. Vice versa, from the monitor to the tool, asynchronous communication is required in any case to pass back events for issued requests. Hence, for the tool-monitor interaction, only one event channel from the monitor to the tool has to be established.

For the instrumentation-monitor communication, in contrast, two event channels for both directions are necessary as asynchronous events emerging from the observed application have to be propagated to the monitor, while control commands from the monitor to the instrumentation may also be passed asynchronously in order not to block the monitor.

Concerning the content of the generated events, we have to distinguish two types of events:

- Events describing basic changes important for the current system state have to be predefined and interpreted by the monitor core. For example, events

indicating the generation of new entities or the deletion of entities belong to this event type. The monitor needs to analyse and interpret these events and update its system state according to this information. This is necessary for coordination purposes, as requests or commands only have to be propagated to concerned entities; without general knowledge of the system state, this could not be accomplished.

- Events not containing critical, state changing information are not of interest for the monitor core. We call these events *user-defined events*, which are not interpreted by the monitor, but only propagated to tools that have been registered for these events by preceding requests.

Thus, the generic event model allows to introduce any kind of event; the monitor core does not associate any semantics with them. The only demand is that the general format of event descriptions is well-defined. Event descriptions therefore have to contain the event type and its parameters in a fixed format, such that they can be dynamically included into a running monitoring system without requiring any changes to it. Of course, for a reasonable usage of user-defined events, instrumentation and tool need to agree on those events; we will explore this condition in detail in section 5.3.

5.2.3 Conclusion

In this section, we have presented MIMO's monitoring infrastructure, which is based on the definition of a monitoring framework, generic interfaces, and a generic event model. The monitoring framework relies on a three-tier model clearly separating tool, monitor, and monitored applications, for which elementary access patterns to the monitor core have been defined. The interfaces are designed for synchronous communication with the monitor core, while event-based communication is used for asynchronous propagation of data.

Altogether, the idea behind the chosen approach is to put as few preconditions as possible into the monitor core in order not to limit its scope of usage. Rather, the monitor core should be considered as an intelligent component for the exchange of requests and events between tools and instrumented applications. In this sense, the monitor core can be seen as a sort of intelligent "router" for request and event information, which can be flexibly deployed in any scenario.

5.3 Usage Methodology and Tool Frameworks

The last important concept of the MIMO approach concerns the efficient deployment of the MIMO infrastructure. As the monitoring infrastructure represents a rather complex environment, a proceeding for the usage of the system is required. This includes technical issues as well as guidelines for users who wish to develop new tools or integrate new middleware into the monitoring environment.

Our concept to enable an efficient usage of the MIMO infrastructure is based on two features: Firstly, a usage methodology giving guidelines for a concerted devel-

opment of tools and the integration of new middleware is defined, and secondly, a reusable and adaptable framework for easy GUI tool construction is presented.

5.3.1 Usage Methodology

As explained above, the generic layout of the MIMO infrastructure provides a high degree of freedom to develop new tools and to integrate new middleware. We have seen that, in addition to the basic requests and event types, user-defined event types serve as a basis for the implementation of new tool functionality. However, the development of tools and instrumentation code is closely related and a methodology for optimal coordination of the development tasks is therefore advantageous.

Our proposed usage methodology for the MIMO infrastructure consists of the following major steps:

1. Definition of tool functionality.

2. Definition of required data to be retrieved from the monitored system in order to fulfil tool functionality.

3. If no mapping of this concrete middleware to the MLM model exists: Definition of the mapping to the MLM model.

4. Definition of events and commands needed for monitoring the application.

5. Implementation of instrumentation code and tool.

It can be seen that the approach starts with the definition of the tool functionality and the necessary events and commands derived from it. If this middleware has not been used with MIMO before, a mapping of its layer model to the MLM model is required; this mapping depends on the usage scenario and can either be general or specialised for the given monitoring tasks. Based on these definitions, tool and instrumentation have to be implemented, which can in principle be done simultaneously. At this point, we cannot make any statements about how the instrumentation has to be implemented as there are a lot of possibilities that are preferable under certain circumstances. The most important aspect is that tool and instrumentation have to be adjusted to each other, while the monitor core only serves as an intelligent way of interaction between them. We will substantiate this methodology later with a concrete implementation of the monitoring system.

5.3.2 Tool Frameworks

When we look at a set of common tools for distributed systems, it is remarkable that a given amount of basic functionality is present with most tools of tool sets. This includes for example basic visualisation capabilities for gaining an overview of the observed system, or logging functions. Beyond this, tool functionality can be very diverse and depend highly on the type of tool and middleware. Consequently, as we wish to monitor heterogeneous middleware with our approach, a tool framework

offering these basic tool functions on the one hand, and being dynamically extensible with any kind of tools functionality on the other hand shows to be profitable. Especially for GUI tools, basic visualisation capabilities are needed in nearly any case and serve as a starting point for further functions. Thus, the concept of a tool framework simplifies tool development considerably and therefore integrates well with our usage methodology because both aim at a systematic and rapid tool development process. We will go into further detail with our tool framework when we describe a concrete implementation.

5.4 Summary

In this chapter, we have introduced the fundamental concepts of the MIMO MIddleware MOnitor approach. MIMO's goal is to design and implement a monitoring system that fulfils the requirements stated in the previous chapter. To accomplish this, MIMO relies on three basic pillars depicted in figure 5.7:

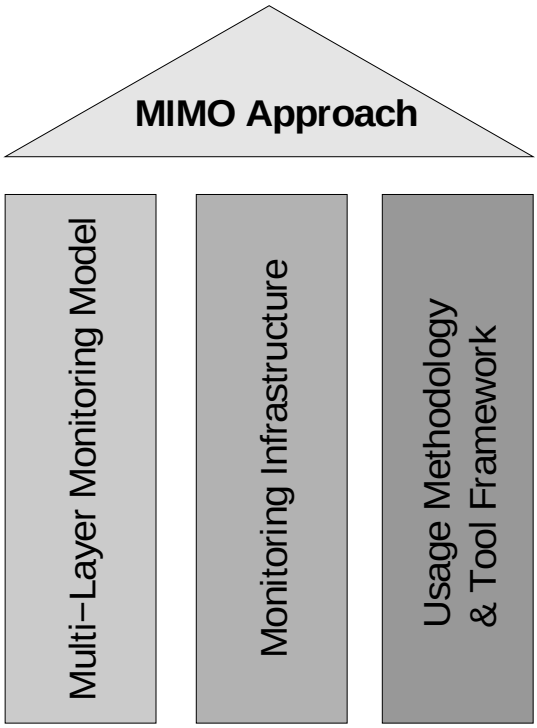

Figure 5.7 MIMO Fundamental Concepts

First, a multi-layer monitoring model defines an abstract information model that is able to handle complex and heterogeneous environments; concrete middleware can easily be mapped to this model by pursuing a given standard procedure. Secondly, MIMO proposes a general monitoring infrastructure affecting the layout of monitor components and their interaction patterns. And thirdly, a usage methodology describing the deployment of the overall infrastructure has been established; this methodology represents an essential part of the approach, as for complex mon-

itoring environments an efficient tool and instrumentation development and collaboration is indispensable. Hence, taking all aspects of the MIMO approach together, its primary aim is to provide a generic monitoring approach that enables a flexible and efficient application in heterogeneous environments.

Chapter 6

MIMO Architecture and Implementation

After having presented the conceptual aspects of the MIMO approach in the previous chapter, we are now proceeding with the architecture and implementation of the MIMO monitoring system [RLRS2000].

The MIMO approach introduced before represents a basic concept for the construction of a generic monitoring system. As we have seen, it is based on the multi-layer monitoring model, a generic monitoring infrastructure, and an appropriate usage methodology. In this chapter, we will present a monitoring architecture describing the components within the resulting monitoring system that fulfils the required properties. Subsequently, we will proceed with a description of the access and usage patterns for these components and a technical view on implementation aspects of the MIMO prototype.

The monitoring architecture can be seen as a kind of coarse design of the MIMO monitor and its components, which is justified by the preceding general discussion of monitoring aspects. The access and usage patterns specify the interactions between components more precisely, and therefore serve as a starting point for tool and instrumentation developers for the integration of further tools or middleware. The description of the implementation finally gives further details about technical aspects; these aspects are not necessarily relevant for using the MIMO system, but interesting from the programmer's point of view.

6.1 Tool Development and Usage Process

First, before going into detail with the MIMO architecture and implementation, we will start with a short survey of the overall tool development and usage process. This is important as several – potentially different – persons are involved in the tool development and usage process, and we have to distinguish between these actuators carefully in the following. Figure 6.1 shows an overview of MIMO's tool development and deployment use cases[1]. The three actuators occurring in our scenario are the tool user, the tool developer, and the application developer.

Based on an existing application or an application under development, the use cases depicted in the figure are necessary to produce a running tool. Starting with

[1] From now on, we will illustrate models and processes using the Unified Modeling Language (UML) [RJB1998], [Alh1998], [Wah1998].

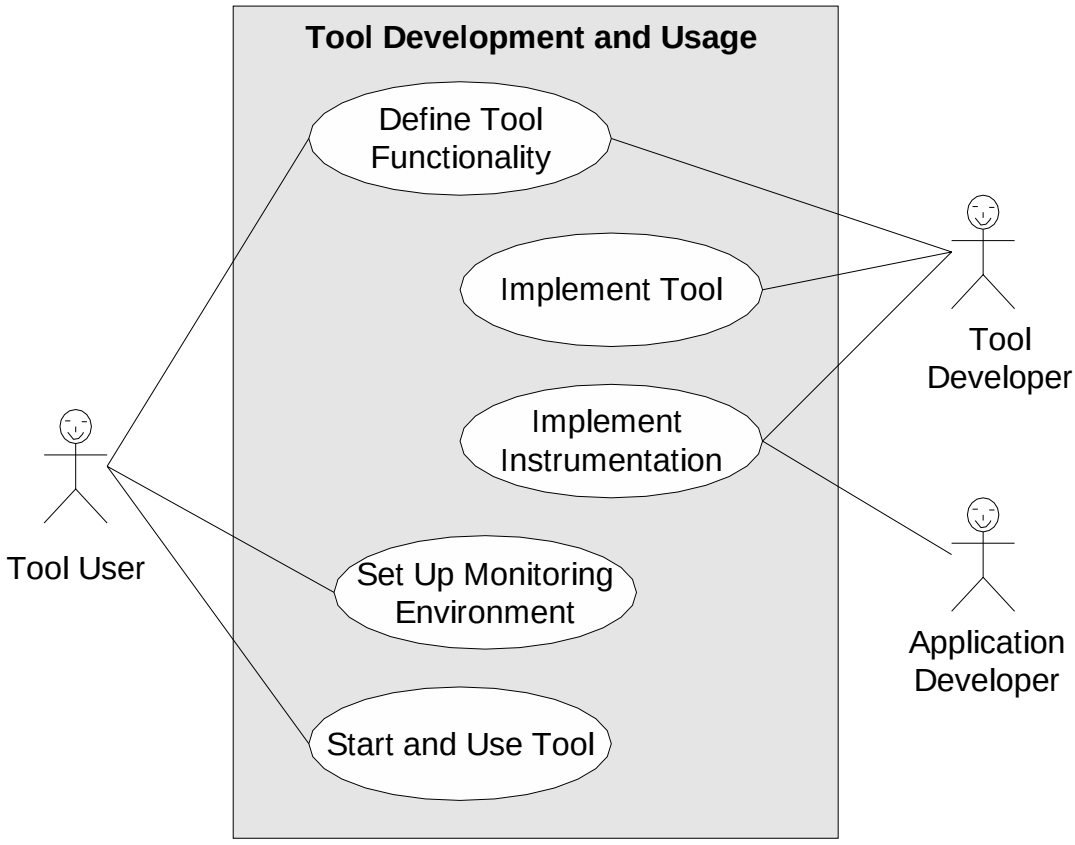

Figure 6.1 MIMO Tool Development and Deployment Use Case

a specification of tool functionality that is made by the tool user and possibly influenced by the tool developer in order to take into account the underlying monitoring system, the subsequent implementation of tool and instrumentation has to be carried out. While the tool itself is of course implemented by the tool developer, the instrumentation can be implemented by both tool or application developers, depending on the concrete scenario. After the implementation of the tool, the tool user is responsible for setting up and configuring the monitoring environment, as well as for finally deploying the tool.

A relevant consideration concerns the mapping of the three roles of application developer, tool developer, and tool user to real persons. Depending on the scenario, these persons need not necessarily be different, whereby two factors mainly influence this mapping:

- *Project size:*
  Of course, for small projects application developer, tool developer and user can merge into a single person. With a rising project size, tasks will be split among several persons, resulting in the different mappings of roles to persons.

- *Tool type:*
  Another important aspect that is orthogonal to the project size is the type

of tool that is being developed. On the one hand, for example, for standard development tools like debuggers, the application developer and tool user will often be identical, whereas the tool developer is working independently. On the other hand, other tools covering e.g. management functions might be developed by the application developer, but used by independent tool users deploying the delivered software.

As a consequence, no general assumption about the persons involved in the tool development and usage process can be made, if our monitoring system should be kept open for all scenarios. Nevertheless, the separation between the different roles has to be kept in mind for the following considerations, and we will come back to these roles especially during MIMO's tool development methodology.

6.2 Monitoring Architecture

The architecture of the MIMO monitoring system relies on the three-tier monitoring approach presented in section 3.1.2. This basic architecture is appropriate to fulfil the requirements stated before and to implement a system according to the criteria postulated in chapter 5.

6.2.1 Basic System Structure

Figure 6.2 shows an overview of the resulting structure of the monitoring environment. The main participants are tools, the MIMO core components, instrumentation components, and application components.

An application may consist of several components, which are represented by processes executed on a given node. These application components are monitored directly by instrumentation components that are responsible for gathering information and controlling them. As instrumentation can be implemented in a great variety of ways, we cannot make any assumption about its concrete implementation here and it is thus denoted as a generic component. In practice, the instrumentation could e.g. either reside in the application component's process or be integrated into the operating system. The relationship between instrumentation and application component is a one-to-one relationship, expressing that every instance of an instrumentation component is responsible for a single application component.

In order to communicate with MIMO, every instrumentation component is assigned to a single instance of MIMO; conversely, MIMO instances can handle several instrumentation components simultaneously, so that any number of instrumentation components may be assigned to a single MIMO instance, depending on the concrete monitoring environment. For tools, the same policy as for instrumentation components holds: Any number of tools can be assigned to a single MIMO instance, but every tool is only attached to one single instance. Both MIMO instances and tools are denoted as processes because they are executed independently.

Finally, as there might be several MIMO instances within an observed environment at a given time, these instances cooperate with each other for ordering events properly and for getting synchronised.

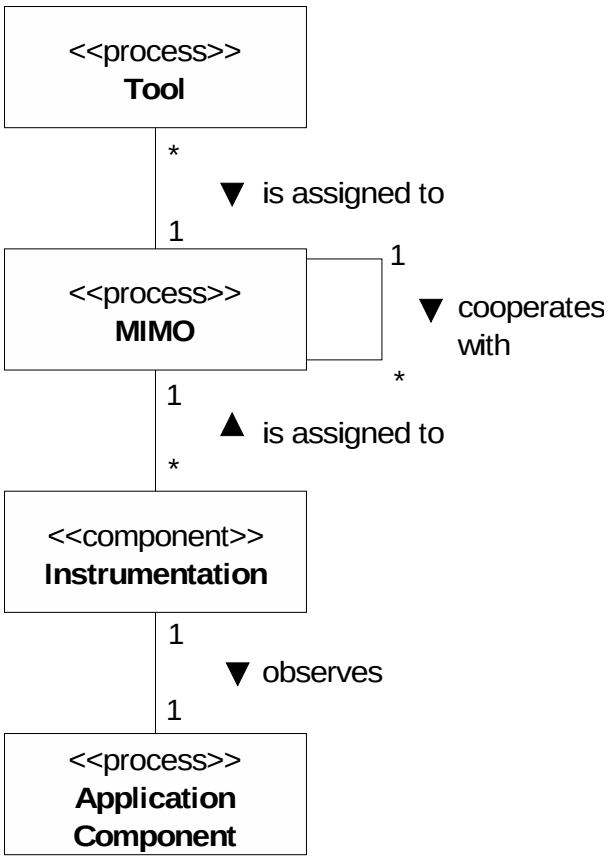

Figure 6.2 MIMO Structure

In the following, we will discuss the individual components and their interactions in more detail; here, distribution aspects concerning the assignment of instrumentation components to MIMO instances are also of relevance, especially with respect to performance aspects.

6.2.2 Components

The basic system structure introduced above contains a set of logical components that have to be mapped to physical implementation components. As a modular layout of the system is already given by following the general three-tier model, the originating components can be defined straightforwardly. In addition to this definition of basic components, their interfaces need to be specified. The resulting component diagram is shown in figure 6.3, including the three components of the monitoring system and the application part.

The interfaces modelled and specified within MIMO are the tool-monitor interface, which serves as an entrance point for tools, and the instrumentation-monitor interface, which represents the contact point for instrumentations. The tool's user interface and the instrumentation-application interface cannot be specified generally: For the user interface, this is impossible as tool functionality can be very

diverse. For the instrumentation-application interface, no general specification is possible if we do not want to restrict the scope of usage of the monitoring system. From the MIMO point of view, the first standardised interface to applications has to be provided by the instrumentation, which can use any arbitrary approach to interact with the application components. Finally, the monitor-monitor interface represents the communication facility among different MIMO instances. This interface is an internal monitor issue and therefore not of interest for tool development.

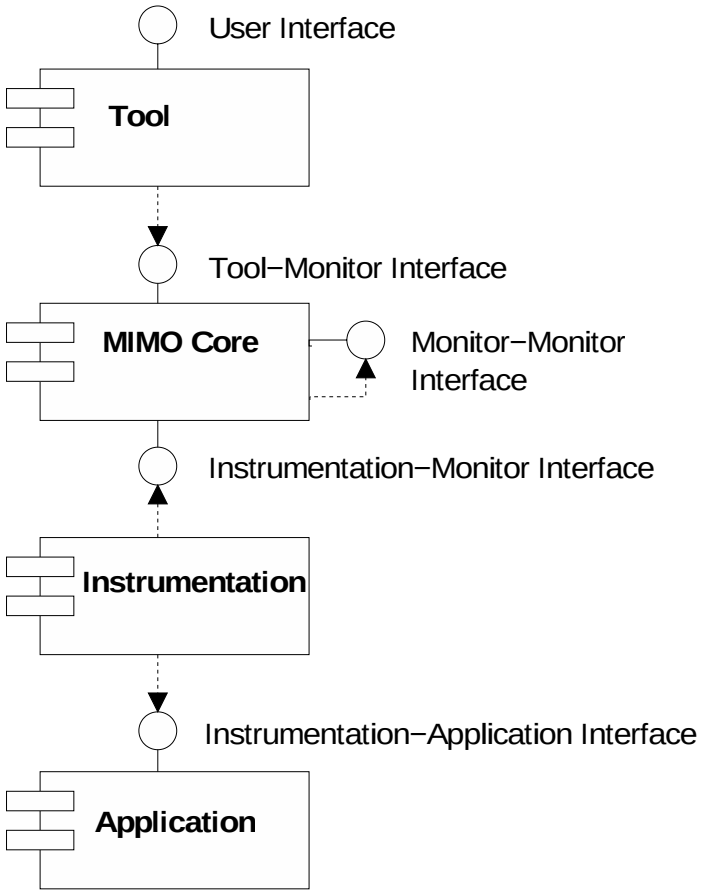

Figure 6.3 MIMO Components

The subsequent sections will illuminate the respective MIMO core, tool, and instrumentation components more precisely.

6.2.2.1 MIMO Core

The MIMO core represents the central component of the monitoring system. It serves as the only entrance point for tools and instrumentation components and provides the interfaces to these components. Besides the implementation of the multi-layer monitoring model, the MIMO core's main duty is to provide the request processing capabilities of MIMO and to care for the proper handling of events. In addition, several preconditions stated before have to be taken into account in order to comply with the requirements of the MIMO approach.

Request Processing and Event Handling. Requests issued by tools have to be processed by the respective MIMO core to which the tool is attached. Thereby, requests issued by tools may either be *synchronous*, i.e. the tool waits for a response and is blocked until the result is passed back from the monitor, or they may be *asynchronous*, where the tool is not blocked and gets the results delivered back by means of events. Depending on the request type, both forms of request processing need to be supported. Event handling, in contrast, is always carried out asynchronously due to the nature of the event notion. Hence, the MIMO core has to provide means for both synchronous and asynchronous interaction with tools and instrumentation components.

Providing a Logically Central Monitor. Physically, MIMO may consist of a set of concurrent MIMO core components that cooperate with each other in order to present a single monitor to the attached components. Thus, the distribution of MIMO components has to be transparent for tools and instrumentation components that only "see" the MIMO core instance they are assigned to. In order to maintain this illusion of a single monitor for tools and instrumented applications, several arrangements are required:

- Requests being issued by a tool to its MIMO instance need to be processed by all active MIMO instances, if they are concerned. To reach this, requests either have to be distributed to all other MIMO instances in order to guarantee the inclusion of all active application components, or it has to be assured that all relevant events are being perceived by the tool's MIMO instance. Furthermore, if necessary, results coming from other MIMO core instances have to be assembled and passed back to the tool in order to return a single result for a given synchronous request.

- Events originating in any application component have to be distributed to interested tools. As said above, this can be done by distributing either events or requests to all active MIMO core instances. However, in addition to this distribution, synchronisation and event ordering has to be carried out in order to guarantee the same time and order of events for all tools, because the applications are running in a heterogeneous environment where no global clock can be assumed [Lam1978].

Of course, the actions required to provide this logically central monitor have to be hidden from tools and instrumentation components, and thus communication required for this purpose is handled through the monitor-monitor interfaces. Details about the implementation of these features will be shown later.

Support of Multi-Layer Monitoring. Another important feature of the MIMO core is the support of the multi-layer monitoring model introduced in the previous chapter. As the MLM model is one of MIMO's fundamental aspects, the MIMO core is designed to provide its functionality based on this model. Requests, responses, and event handling therefore rely on data types derived from the entity model defined within the multi-layer monitoring approach. Hence, every tool and instrumentation component has to interact with MIMO using these types.

Provide Monitoring Infrastructure. One final important task of the MIMO core is to implement the monitoring infrastructure illustrated in section 5.2. Above all, this includes the implementation of generic interfaces needed to communicate with tools and instrumentations. Tools may issue requests to the MIMO core after an initial attachment to the monitor, and instrumentations may send events to it. The "application logic" within the MIMO core is responsible for processing the requests and triggering the right actions on the basis of the incoming events. Doing so, the implementation of these features has to be kept generic in order not to restrict the scope of usage of the monitor core. We will explain the concrete implementation of these interfaces and event handling procedures in section 6.2.3.

6.2.2.2 Tools

Tools represent the next essential component within our monitoring scenario. As mentioned before, every tool is assigned to a single MIMO instance and may issue requests to it. Results are passed back synchronously or asynchronously, depending on its type. Requests where the tool waits for the response immediately and is blocked for that time are called *synchronous requests*. This is e.g. useful for querying the current state of some monitored entity; here, the result can be returned with a very small delay caused by the MIMO core for processing the request, and no critical waiting time for the tool arises. Other requests, which are interested e.g. in the occurrence of a certain event, are normally issued asynchronously as the time until the event actually takes place is not predetermined.

In our general context, we do not make any further assumption about tools, except that they communicate with their MIMO core through the defined interfaces.

6.2.2.3 Instrumentation

The third important component of our monitoring architecture is the instrumentation. Instrumentation is required for gathering information from monitored application components and delivering it to the MIMO core, as well as for executing monitor commands that manipulate the running application.

There is a great variety of ways for instrumenting applications, which highly depend on the runtime environment and implementation issues of the application. Therefore, the MIMO monitoring architecture does not make any assumption about how instrumentation is actually implemented. The only important issue is to enable a standardised interaction between the monitor core and the instrumentation. Syntactically, the interaction between the instrumentation and the MIMO core is implemented using standardised interfaces caring for a proper communication. However, in order to allow MIMO to monitor a certain application component, additional information concerning its classification within the MLM model is required.

Hence, the semantics of certain, fundamental events needs to be predefined, allowing the MIMO core to interpret these events and store basic information about the state of the observed system. Beyond this basic information, the semantics of events cannot be predefined because the deployment scenarios of the monitor can be very diverse and instrumentation components may produce various event types.

This may e.g. comprise further attributes or properties of monitored entities, which can then be queried by tools with an event-based approach without affecting the MIMO core at all. To deal with this issue, the MIMO core does not interpret these events, but passes them directly to interested tools registered for these event types. The only thing that needs to be guaranteed in this case is that instrumentation and tool are adjusted to each other and operate on a common event semantics. This is relevant for events being passed from the instrumentation to the tool as well as for commands being sent from the tool to an instrumentation component. We will explain the required proceeding for this cooperation in detail in chapter 7.

6.2.3 Interaction Patterns

During the description of the MIMO components, the interactions between these components have already been touched slightly. In this section, we will introduce MIMO's communication concept in detail, before we proceed with more complex usage patterns in the following section. A key precondition of the communication concept is to implement a generic communication approach as defined in section 5.2.

6.2.3.1 Synchronous and Asynchronous Communication

As already mentioned, MIMO's communication relies on both synchronous and asynchronous interactions between tools, the MIMO core, and instrumentations. Synchronous communication is used whenever the component initiating the interaction waits for the result and is blocked until it is received. Hence, synchronous communication is in general only useful when the processing time of a request is low and the result gets passed back immediately. Asynchronous communication, in contrast, is useful for event-based interactions where the time until the occurrence of the event is not determined and the originator of the request does not want to be blocked until that time.

Within the MIMO architecture, synchronous and asynchronous communication are implemented using different approaches. For synchronous communication, *synchronous interfaces* offering a set of predefined operations are used, whereas for asynchronous communication *event channels* for the delivery of events are set up.

As the MIMO implementation itself is a distributed middleware application, it makes use of middleware techniques to enable the communication between its components. Independent of an actual MIMO implementation, nearly every middleware platform offers mechanisms for implementing synchronous interfaces and event channels; therefore, the following considerations are still independent of the concrete middleware taken for the MIMO prototype.

6.2.3.2 Synchronous Interfaces

Usually, interfaces are defined using an Interface Definition Language (IDL), which lists all available operations and their parameters. Although some middleware platforms allow the definition of interfaces with asynchronous operations, MIMO exclusively uses interfaces with synchronous operations.

Looking at the interfaces illustrated in the component diagram in figure 6.3, the interfaces marked there denote any kind of communication facility, no matter whether they are synchronous or asynchronous[2]. Within MIMO, synchronous interfaces are used to implement the following parts of the communication architecture:

- *Attach and detach operations:*
  For the tool-monitor interface and the instrumentation-monitor interface, the operations for attaching a new tool or instrumented application component, or for detaching such a component are carried out synchronously. This makes sense because these operations are only executed once per tool or instrumentation component, while they are not time critical.

- *Request operations:*
  Moreover, issuing new requests by a tool to the MIMO core is carried out synchronously; depending on the type of request, either the result is returned to the tool immediately, or a request identifier is given back if further results will arrive through the event mechanism.

The monitor-monitor interface in the component diagram is completely implemented without using synchronous interfaces, and for the instrumentation-application interface, no statement about its design can be made here as it entirely depends on details of the application.

6.2.3.3 Event Channels

Event channels are supported by most middleware platforms and provide a mechanism for decoupling the communication between a supplier and a consumer. With the event channel approach, suppliers obtain the possibility to *push* events into the event channel, which in turn delivers the events to consumers that are registered for these events. As multiple subscribers can be attached to a single event channel, this approach represents a mechanism for implementing a multi-cast system. Within MIMO, event channels are applied for three cases:

- *Monitor-tool event delivery:*
  As mentioned above, tools may receive events from the MIMO core when requests for certain events have issued.

- *Instrumentation-monitor interaction:*
  Except for the attachment and detachment operations, any further interaction between instrumentation and monitor core is carried out asynchronously. Events originating in the instrumented application are passed to the MIMO core through a monitor-instrumentation event channel, and commands or requests from the monitor core to the instrumentation are passed through an instrumentation-monitor event channel.

[2]The term “interface” in the UML component diagrams is more general as it covers both synchronous and asynchronous communication mechanisms; in our technical context, we speak of interfaces only in the synchronous case.

Asynchronous communication is in this case obligatory because the instrumentation is often integrated into the application process, which should not be blocked in any case. As the event channel buffers incoming events for the instrumentation until it pulls them from the channel, a minimal intrusiveness can be guaranteed for the application with this proceeding.

A final, relevant aspect of event channels is that they can be used in an *untyped* mode, in which the type of event needs not be declared at development time. Rather, events carry their own type description with them, enabling to dynamically generate new event types during runtime. This is of crucial importance as MIMO's event model should be kept generic, which can be guaranteed with our proceeding.

□ *Monitor-monitor interaction:*
A final usage scenario for event channels is the interaction between several MIMO core instances. Here, requests or events need to be distributed among the instances in order to carry out synchronisation or event ordering tasks. As the MIMO core instances act independent of each other, asynchronous communication is obligatory here, too.

Moreover, a benefit of event channels for this case is its multi-cast ability: Whenever data need to be shared among all instances, a simple push operation to the event channel enables the distribution of the data to all attached MIMO instances. This is extraordinarily useful in our monitoring scenario, where the number of active MIMO instances can vary in the cause of time. Without using event channels, every instance would have to keep track of the living instances, which would result in a rather high management effort.

To sum up, it can be seen that event channels enabling asynchronous communication are deployed within MIMO at several crucial positions where the decoupling of communication partners is mandatory.

Altogether, synchronous and asynchronous communication paradigms complement one another within MIMO, such that administrative tasks are managed through synchronous interfaces, while the dynamically evolving behaviour is handled by event channels. We will come back to implementation aspects of our communication architecture during the description of the MIMO implementation in section 6.4.

6.2.4 Distribution and Assignment of MIMO Instances

During the description of the basic monitoring structure, the assignment of tools and instrumentation components to MIMO instances has already been mentioned. The assignment of an entity to a given MIMO instance has the effect that events originating from this entity are always passed to its assigned MIMO instance, and that it receives commands or requests from its MIMO instance; the assigned MIMO instance therefore represents the entity's direct communication partner. Now we will present a comprehensive and formal analysis of this aspect, followed by several application scenarios with practical relevance.

6.2.4.1 Formal Description of Distribution Aspects

In general, in our monitoring environment a set of entities is monitored by a set of MIMO instances. Here, questions concerning the distribution of MIMO instances and the assignment of entities to these instances arise. Our approach to solve this problem is to define a simple mapping between entities and MIMO instances, which may be used flexibly according to the respective usage case.

Formal Model. For simplicity, the assignment of the entities to a certain MIMO instance is determined by properties of a predefined layer λ. Hence, let E be the set of entities in layer λ:

$$E = \{e_1, \ldots, e_k\},$$

and M be the set of active MIMO instances:

$$M = \{m_1, \ldots, m_l\}.$$

The assignment of entities to MIMO instances defines a mapping of E to M, such that every m_i in M monitors a set of entities μ_i with

$$\mu_i = \{e_{i,1}, \ldots, e_{i,k_i}\},$$

where $e_{i,j} \in E \; \forall j \in \{1, \ldots, k_i\}$. As explained before, every entity in E is monitored by a single MIMO instance, such that the following property holds:

$$\forall i, j \in \{1, \ldots, l\}, i \neq j : \mu_i \cap \mu_j = \emptyset,$$

and every entity in E is monitored by exactly one MIMO instance:

$$\sum_{i=1}^{l} k_i = k,$$

where k_i represents the number of entities assigned to m_i (for $i \in \{1, \ldots, l\}$).

In addition to this model, other assignment strategies taking into account more than one entity layer are imaginable, but no further profit arises from such an approach; we will therefore restrict our considerations to the simple case described above.

Heuristics. As a precondition of this distribution strategy, a certain layer λ has to be distinguished for the assignment of entities to MIMO instances. This selection can be influenced by many criteria that depend on the concrete monitoring environment.

However, in practice, a heuristics for our proceeding is to take the hardware layer (i.e. the computational nodes) of the multi-layer monitoring model as the distinguished layer and run one MIMO instance per node. This heuristics guarantees that entities are monitored by their local MIMO instance, which is in many cases advantageous for performance reasons. Nevertheless, other assignment strategies are possible and in some cases useful, too, as we will illustrate in the following section.

6.2.4.2 Example Scenarios

After having presented the formal background of the mapping of entities to MIMO core instances, we will now show two examples for the practical usage of the model.

Central MIMO Instance. The first application scenario is shown in the UML component diagram in figure 6.4. The example consists of two tools running on two different nodes, a single MIMO core instance running on a separate node, and two instrumented applications on two further nodes. The assignment of entities to MIMO instances is trivial in this case, as there exists only one MIMO instance that manages all monitoring activities.

Clearly, the MIMO instance in this case represents a potential bottleneck, especially when several instrumented applications are involved. For small monitoring scenarios, this approach can be useful as only little effort is required to set up the monitoring environment. For cases carrying out management tasks that generate only few events and where latency is not critical, this approach is applicable, too.

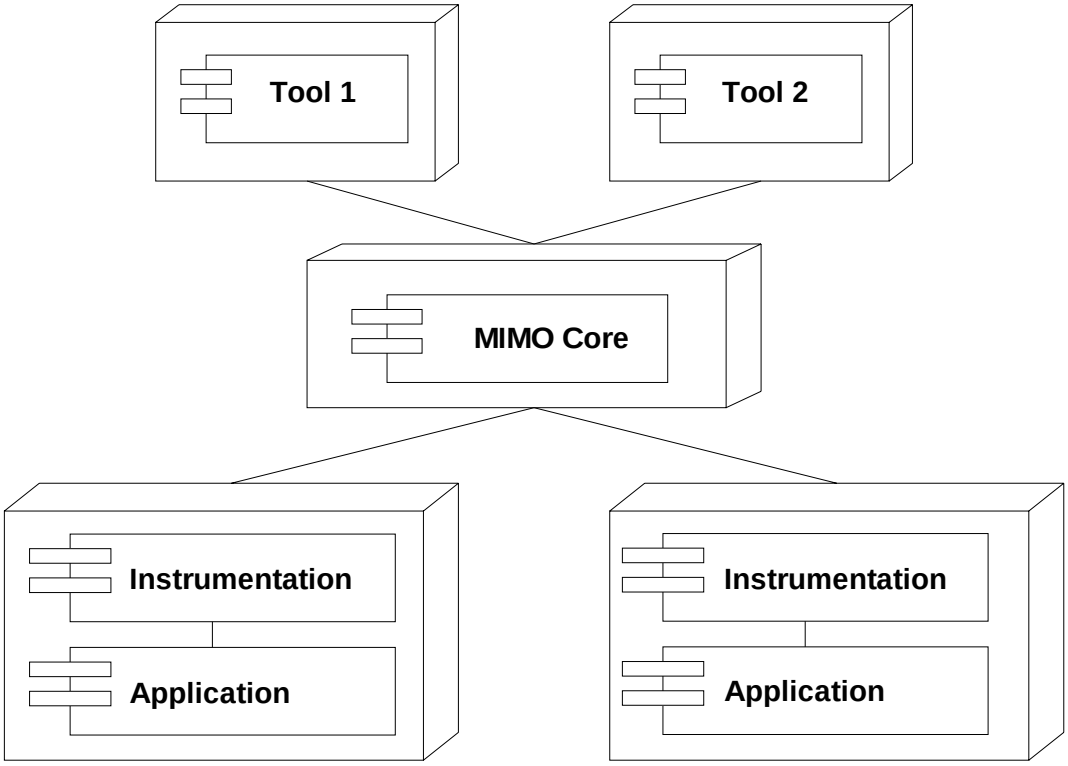

Figure 6.4 Deployment Scenario with Central MIMO

Distributed MIMO Instances. A second example is illustrated in figure 6.5. Here, each of the two nodes executing instrumented application components possesses its own MIMO instance, to which several tools are attached. This case represents an example for the heuristics mentioned in the previous section, where nodes are

used as the decisive entities for the assignment to MIMO instances and every node executes an own MIMO instance.

This monitoring scenario is well suited for large environments, as no MIMO instance becomes a potential bottleneck (except the case of a single node overloading its local MIMO instance, which should not happen in any case). Nevertheless, network traffic increases when the number of events that have to be distributed among the MIMO instances is high. With our approach it is therefore less likely that nodes get overloaded.

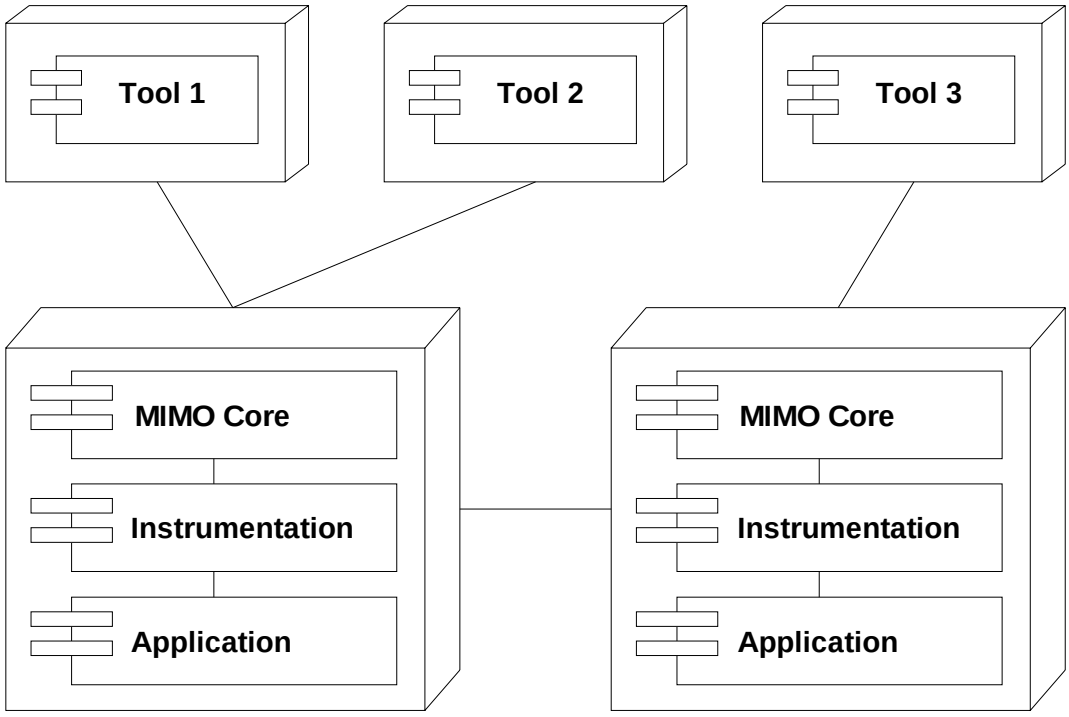

Figure 6.5 Deployment Scenario with Distributed MIMO

Summing up, our examples represent two extreme cases with either a complete centralisation or a complete distribution of the monitoring tasks. Pursuing these considerations, mixed scenarios combining both approaches are also imaginable and useful. The result of such combinations are monitoring environments where certain MIMO instances are responsible for a given set of nodes. The number of assigned components may e.g. depend on the expected effort needed to monitor these components. Of course, the strategy for the distribution and assignment of MIMO instances is a question arising during the set-up and deployment of the monitoring system. Here, to minimise the monitoring overhead a trade-off between computation and communication overhead on the respective nodes has to be found. In any case, the benefit of the MIMO approach is to enable its applicability to all kinds of distribution cases.

6.2.5 Summary

In this section, we have introduced the monitoring architecture of the MIMO system, i.e. the software components and their potential physical distribution. The basic architecture relies on a three-tier model with the components tools, MIMO core, and instrumentation.

The monitoring architecture fulfils the requirement for a generic monitoring infrastructure, as it defines a core framework, and because it is flexibly usable and configurable for a wide range of application scenarios. The interfaces within the system are based on synchronous interfaces and event channels, both of which are kept generic and not restricted for a certain usage case. Moreover, the MIMO core represents the central component for the implementation of the multi-layer monitoring model, on which the overall MIMO approach is based.

What remains to be solved is a coordinated development process for tools and instrumentation components. As we have shown, this requirement can be derived from the event semantics that has to be defined according to the respective tool scenario. The solution to this issue lies in a systematic tool development methodology, which we will present in the next chapter.

6.3 MIMO Access and Usage

After discussing the general monitoring architecture of the MIMO approach, we will now describe the access and usage of MIMO from the tool and instrumentation point of view.

Furthermore, from now on we will describe our concrete implementation of the MIMO prototype, which is based on the CORBA. Data types, interfaces, and event channels are therefore specified using the CORBA interface definition language (IDL). Of course, an implementation could also make use of a different middleware platform as long as it supports synchronous interfaces and event channel mechanisms. The transformation of the CORBA description to another middleware is straightforward.

6.3.1 Entity Definition

A foundation for the interaction between all participating components is the description of the entities defined in the multi-layer monitoring approach. Figure 6.6 shows the CORBA IDL specification of the entity notion. Every entity is determined by its type and its identifier; the type can be any layer from the MLM model and the identifier in its general shape is represented by a string. Within MIMO, it is assumed that identifiers are globally unique. This condition holds for most middleware platforms, where object references are introduced; if proprietary identifiers are used here, the instrumentation has to take care of their uniqueness.

For cases, where a set of entities is passed as a parameter, the type for an entity list can be used; the CORBA sequence allows to pass a list of arbitrary length here.

Another basic specification treats the exceptions that MIMO can throw while processing its requests. For this purpose, the type MIMOException is introduced

```
module MIMO {
  enum  EntityType { E_Appl, E_Interface, E_Object,
                     E_Impl, E_Runtime, E_Node };

  typedef string  EntityID;

  struct  Entity {
    EntityType etype;
    EntityID   eid;
  };

  typedef sequence<Entity>  EntityList;
};
```

Figure 6.6 Entity IDL Definition

and catches possible errors.

6.3.2 MIMO Core Start-Up

Before any tool or instrumented application can make use of the MIMO system, the monitoring environment needs to be set up and started. As we have seen in the previous section, the distribution and assignment of MIMO core instances is very flexible and can be selected according to the respective application scenario. We will discuss details concerning these aspects in chapter 7.

Independent of the distribution of MIMO instances, tools and instrumentations need to find a MIMO instance when being started. For this reason, the CORBA Naming Service [OMG2000b] is used as a central point for looking up active MIMO core instances. The first MIMO core that is started creates a naming context for the MIMO system, where all MIMO instances register themselves. Here, every registration contains the name of the host where the MIMO instance is being executed, allowing tools and instrumentations to select a MIMO instance on their local host (if it exists); this proceeding simplifies the heuristics described in section 6.2.4, whereas other selection strategies might cause more effort.

With this start-up procedure, it is ensured that every tool or instrumentation is able to find a MIMO instance to become attached to. Of course, further activities are necessary at the start-up of the MIMO environment, but we will present implementation details – e.g. the installation of the event channel for monitor-monitor communication – later in section 6.4; here we will only concentrate on aspects important from the tool and instrumentation point of view.

6.3.3 Tool View

Tools communicate with the MIMO core through the tool-monitor interface and through the monitor-tool event channel. Figure 6.7 shows IDL specification of the tool-monitor interface, which serves as the entrance point for any tool-monitor interaction. The operations offered by this interface cover attachment and detachment of tools. as well as request processing functionality.

```
module MIMO {
  interface ToolMonitor {
    ToolID  attach(in string tname,
                   out CosEventChannelAdmin::EventChannel evch)
            raises(MIMOException);

    void    detach(in ToolID tid)
            raises(MIMOException);

    long  request(in RequestName rname,
                  in EntityList elist,
                  out any result)
          raises(MIMOException);

    RequestID  start_request(in ToolID tid,
                             in RequestName rname,
                             in EntityList elist)
               raises(MIMOException);

    long  stop_request(in RequestID rid)
          raises(MIMOException);
  };
};
```

Figure 6.7 Tool-Monitor Interface

6.3.3.1 Attachment and Detachment

To start a monitoring session, every tool needs to be attached to a MIMO instance. For this reason, it has to query the naming service and get an object reference of a MIMO core according to its selection strategy. In order to simplify this process, there are helper packages for several programming languages, which we will explain in more detail in the implementation part.

Once the tool keeps a reference of its chosen MIMO instance, it calls the attach operation to indicate its existence. As a parameter, the tool's name is passed to MIMO; this name can be user-defined and has no further relevance, except that MIMO is able to display a list of attached tools. The result of the operation is a tool identifier that is unique for this MIMO instance and an event channel through which events are passed back to the tool.

The event channels within the MIMO prototype make use of the CORBA Event Service [OMG2000a]. Using the event channel, the tool is able to subscribe its events in various ways; most importantly, it can implement a push- or pull-style consumer, receiving events either immediately or on demand by polling the channel.

To indicate the end of a monitoring session, a tool has to call the detach operation. Upon that, the MIMO core destroys the event channel and cancels all active requests for that tool.

6.3.3.2 Requests

The most important operations of the tool-monitor interface are the two request operations for synchronous or asynchronous tasks.

The request operation is used for synchronous queries about the current system state. The input consists of a request name and a list of entities upon which the request should be applied. The result of this operation is of the CORBA type any, which is a generic type that can hold any kind of data. This type of data contained in the result is determined according to the respective request. For example, if the request queries all active entities in a given layer, the result is a list of these entities, packed up in the CORBA any. Requests supported by the MIMO prototype are only queries for the system state, while the rest of functions is handled through event-based requests.

The start_request operation works similarly to the synchronous operation, but receives a request identifier as a result instead of the CORBA any. If relevant events for the request occur, they are sent through the monitor-tool event channel with the respective request ID, which is needed by the tool in order to determine the appropriate request. The stop_request operation finally terminates an active asynchronous request.

6.3.3.3 Tool Lifecycle

Summing up, a tool's lifecycle can be illustrated with the UML sequence diagram in figure 6.8. After an initial attachment, an arbitrary number of requests can be issued, where results are given back either synchronously or through the event channel. The detachment operation finally terminates the tool lifecycle.

6.3.4 Instrumentation View

The communication between instrumentation components and the MIMO core is handled by the instrumentation-monitor interface and two event channels for further asynchronous interaction. Figure 6.9 shows the instrumentation-monitor interface, which only provides the two operations for attaching and detaching an instrumented application component.

6.3.4.1 Attachment and Detachment

The attachment works similar to the tool attachment described above. After fetching an object reference of the MIMO core, the attach operation is called. In this case, two event channels are set up: The instrumentation-monitor event channel for delivering events to the monitor, and the monitor-instrumentation event channel for sending commands or requests to the instrumented application. As explained before, events from the monitor core to the instrumentation need to be passed asynchronously in order to leave the decision about the processing modalities to the instrumentation, which tries to perturb the running application as little as possible.

Like before, the detachment of an instrumented application is announced through the detach operation, which takes the instrumentation's ID as a parameter.

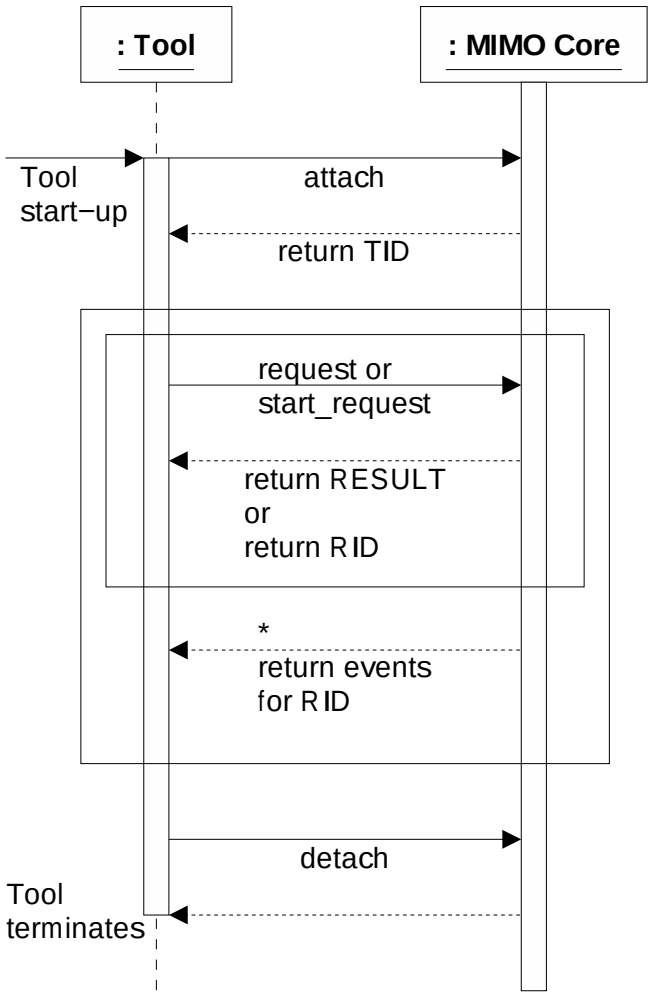

Figure 6.8 Tool Lifecycle

6.3.4.2 Event-Based Interaction

The event-based interaction through the two event channels between MIMO core and instrumentation is twofold:

- Firstly, general events that are important for the overall system state are passed any time from the instrumentation to the monitor core, without being explicitly requested. Such events concern mainly the creation of new entities, or the deletion of existing entities.

  In order to keep track of the current system state, the MIMO core needs to be informed about such actions in any case. With these data, the graph describing the multi-layer monitoring oriented view on the observed system can be constructed; this graph therefore serves as a basis for further requests to certain entities describing the system state. Of course, the syntax and semantics of these events needs to be predefined because the static MIMO core has to be able to analyse them.

```
module MIMO {
  interface InstrumentationMonitor {

    InstrumentationID attach(in string iname,
                    out CosEventChannelAdmin::EventChannel ievch,
                    out CosEventChannelAdmin::EventChannel mevch)
                  raises(MIMOException);

    void detach(in InstrumentationID iid)
          raises(MIMOException);
  };
};
```

Figure 6.9 Instrumentation-Monitor Interface

- Secondly, events containing requests about specific entities can be sent from the monitor core to the instrumentation. Such events may contain queries about further properties of entities, or issue commands to be executed by the instrumentation. If a response is necessary, the instrumentation sends back an event enclosing the results for a preceding query. The content of these events cannot be predefined because it depends on the application and the capabilities of the respective instrumentation.

To implement this approach, it is necessary to define a general event format that allows to pass any kind of data between instrumentation and monitor, without predefining the semantics of all possible events. Therefore, MIMO uses the generic event format shown in figure 6.10 to pass events between instrumentation and monitor core. Every event contains a request identifier that is used to refer to

```
module MIMO {
  struct InstrumentationEvent {
    RequestID rid;
    long long time;          // time of event
    string    etype;         // type of event
    any       description;   // description of the event
  };
};
```

Figure 6.10 Instrumentation Event Definition IDL

a certain request to which this event is a response; for general events (that have no corresponding request) this identifier is zero. Furthermore, every event contains its occurrence time, a string describing its type, and a generic CORBA `any` with its description.

For general events like the creation or deletion of entities, the type and content of the description are predefined in order to enable the MIMO core to analyse and update its system state. Other event types are not touched by MIMO, but passed on to tools interested in these events. Those tools have to be able to analyse the content of the description due to additional agreements that are made in the context of a coordinated tool and instrumentation development. Moreover, the capabilities

of the respective instrumentation highly depend on implementation details of the application, such that in some cases different components of the same application may exhibit different functionality; therefore, a valid response to a query may also contain the non-availability of a requested feature.

6.3.4.3 Instrumentation Lifecycle

Recapulating the lifecycle of an instrumentation yields the UML sequence diagram in figure 6.11. The initial attachment after application start-up is followed by any number of events originating in the instrumented application, or by requests or commands coming from the monitor core.

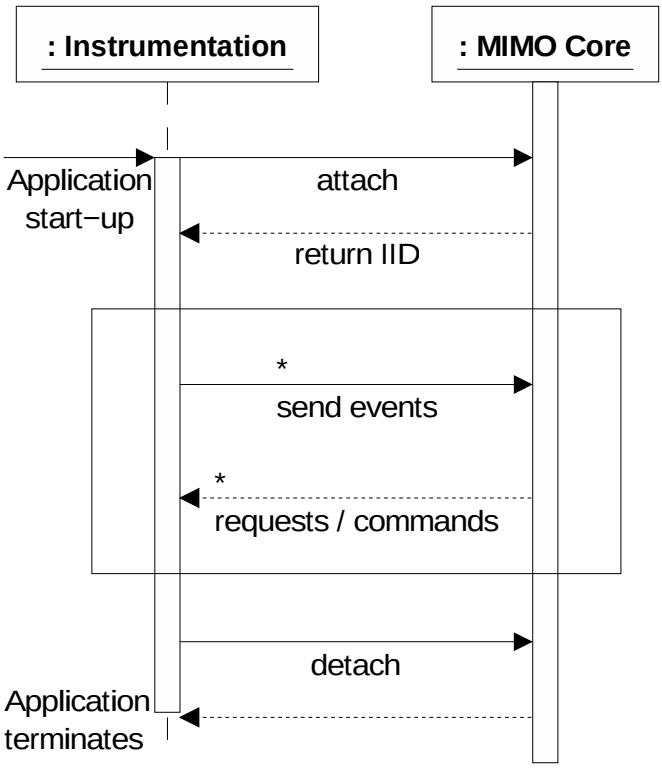

Figure 6.11 Instrumentation Lifecycle

6.3.5 Example

Figure 6.12 shows an example of the collaboration of tools, MIMO core, and instrumented applications. After starting up a MIMO core, two applications attach to MIMO and receive their identifiers. During application runtime, two application entities $O1$ and $O2$ are newly created and announced to MIMO by sending the appropriate events.

A subsequently started tool usually proceeds as follows: First, it queries MIMO about the current system state, i.e. the entities existing within a specific abstraction layer of the layer model. In our example case, the tool issues a `get_objects`

request that queries the entities on the object level of the multi-layer monitoring model. Accordingly, MIMO returns the two known entities $O1$ and $O2$ to the tool.

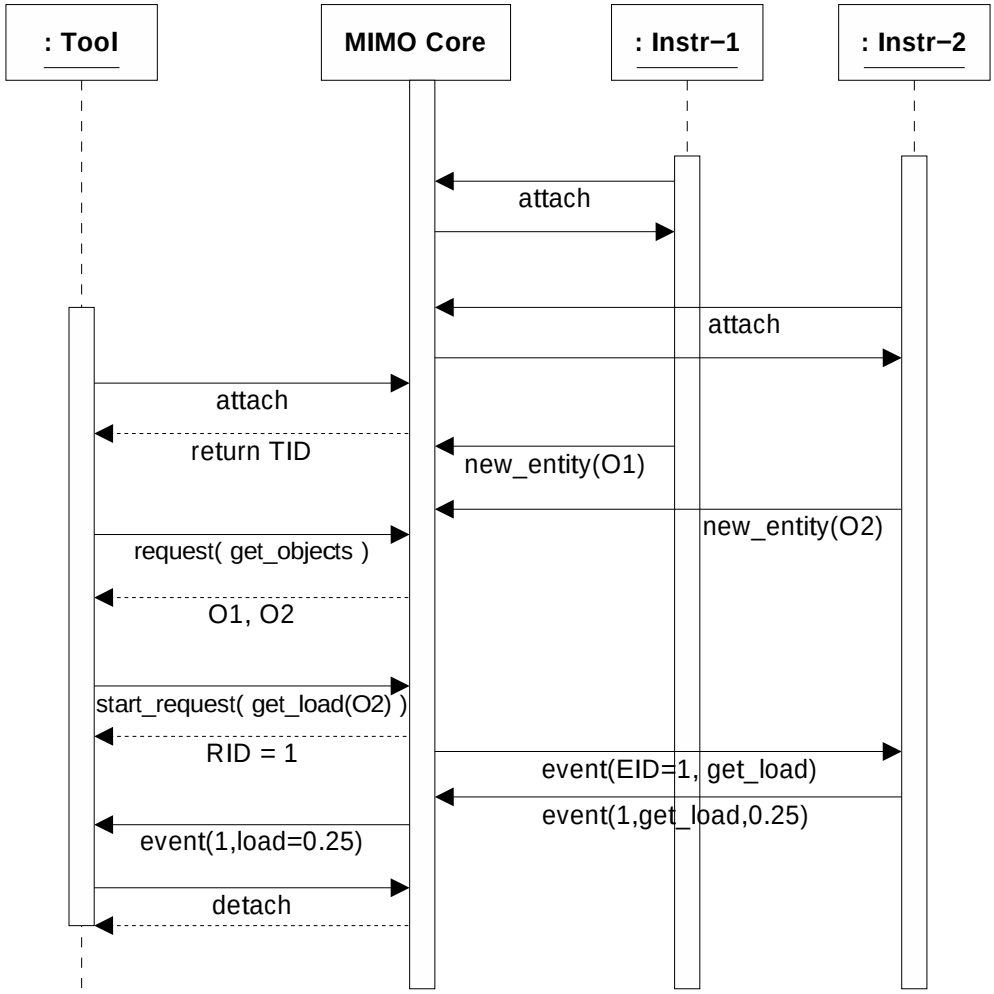

Figure 6.12 Usage Example

Now, the tool can issue further requests concerning the entities known to the system. Our example tool queries entity $O2$ for its load by sending the request `get_load` for $O2$ to MIMO. As this is no basic request concerning the system state, it is initiated asynchronously, such that MIMO generates and returns a request identifier to the tool and forwards the request to the respective instrumentation for $O2$, again with an identifier for its request to the instrumentation.

The instrumentation now measures the required data and sends an event with the appropriate event identifier back to MIMO, which in turn returns the event with the respective tool request identifier to the tool. Usually, as the request was started asynchronously it would have to be stopped again, but for our example we assume the semantics of our `get_load` command only to return it once and terminate automatically.

An important aspect that can be seen in this example is the coordination between tool and instrumentation. Except for basic event types, tool and instrumentation

need to agree on the syntax and semantics of events in order to cooperate with each other. This point will be intensified with our tool development methodology in chapter 7.

6.3.6 Summary

This section was devoted to the access and usage of the MIMO system. We have illustrated the interfaces for both tools and instrumented applications, as well as mechanisms for asynchronous event-based communication. Furthermore, we have shown the basic usage patterns that describe the common proceeding for using MIMO.

The benefit of MIMO lies in its applicability for very diverse scenarios because only few predefined requests and events for updating the system state are required, whereas any kind of user-defined request and event type can be dynamically introduced into the system. With respect to our requirements defined in chapter 4, this allows to build a flexible and extensible monitor that is able to systematically integrate different request and event types.

6.4 MIMO Implementation

In this section, we will describe further details of our prototypical MIMO implementation [RLRS2000]. As already indicated, this includes organisational aspects of the start-up process of the MIMO environment and internals of the MIMO core implementation, as well as implemented tools and instrumentation components.

The MIMO core description concentrates on internal data structures, request and event processing procedures, and distribution aspects. The tool description mainly covers our MIVIS tool framework, while for the instrumentation side the focus lies on the integration of the CORBA and DCOM middleware, as well as Java applications. Finally, we will conclude with performance considerations of our MIMO prototype.

6.4.1 MIMO Core

Here we are describing MIMO's start-up procedure, its internal data structures, its request and event processing policies, and synchronisation and event ordering mechanisms. The prototype of the MIMO core is completely implemented in Java, thus allowing it to be executed on nearly any operating platform.

6.4.1.1 Start-Up Procedure

Our monitoring environment is highly dynamic due to the flexible distribution of MIMO instances and the dynamic attachment and detachment of applications. Therefore, a method to look up and connecting to MIMO instances is needed for tools and instrumented applications.

Naming Service and MIMO Registration. As already mentioned, the CORBA Naming Service is used for looking up active MIMO instances. However, in our

dynamic context this is not sufficient for solving our initialisation problem: In order to connect to the CORBA Naming Service, tools and instrumentation components need to know its object reference (IOR). As this IOR may vary in the cause of time when the Naming Service is restarted or migrated to another host it cannot be used as a static entry point for components that wish to look up MIMO. The solution to this problem is the introduction of another abstraction level for initialising the system: When the Naming Service is started on a given machine, it makes use of a WWW server to register its IOR under a predefined and static URL (Uniform Resource Locator). Clients wishing to know the IOR of the Naming Service can then contact the WWW server and query the IOR with a fixed URL. With this IOR, they can connect to the Naming Service and receive the object references of MIMO instances.

Once the Naming Service is started and its IOR can be queried from the WWW server, every MIMO instance that is started registers its IOR under the MIMO naming context; as a key, the machine name where the MIMO instance is being executed is taken. Hence, tools or instrumented applications can query the MIMO naming context and get a list of all active MIMO instances with their respective host names. With this information, they can select a MIMO instance to attach to according to their policy; for example, as mentioned before, they might take a local MIMO instance if it exists.

Setting Up the Monitor-Monitor Event Channel. A monitor-monitor event channel is needed for the communication between the active MIMO instances. Therefore, the first MIMO instance that is started up creates this event channel and registers it under the MIMO naming context. Every subsequent MIMO instance can then query this event channel and announce its existence to the other instances.

6.4.1.2 Internal Data Structures

The relevant internal data structures existing within every MIMO instance are the system state, the tool list, the instrumentation list, and the request list.

System State. Every MIMO instance stores a graph of all existing entities within the observed environment with respect to the multi-layer monitoring model. As said before, events indicating the generation or destruction of entities are broadcast to all MIMO instances through the monitor-monitor event channel, so that every instance can incrementally update its system state. When a MIMO instance starts up, it can receive the current system state by requesting it through a special event that is sent to the monitor-monitor event channel. Any active instance can then answer this request to initialise the new MIMO instance; a simple heuristics is to take the first MIMO instance that set up the monitor-monitor event channel as the reference to answer such initialisation requests.

Tool and Instrumentation List. Obviously, every MIMO instance has to store lists of tools and instrumented applications attached to it. In addition to the respective identifiers, a list of the event channels used for interaction with these components is also stored.

Request List. As described above, there are two kinds of requests that may be issued by a tool. Synchronous requests concerning the current system state are processed by MIMO immediately and need not be stored. Asynchronous requests, however, for which events might be delivered later need to be stored in the request list of the concerned MIMO instances.

6.4.1.3 Request and Event Processing

The two kinds of requests that can be issued are synchronous (ad-hoc) requests and asynchronous requests. From the MIMO core's point of view, processing these requests is quite different.

Synchronous Requests. MIMO immediately answers synchronous requests and blocks the tool until it delivers the response, which is passed back as the result to the operation call. Such requests can only query the current system state and do not imply further operations on instrumentation components. Therefore, they need not be stored in the request list because no events for them may originate later.

The fundamental operation for querying the system state has to implement the algorithm for detecting related entities described in section 5.1.6.3. The basic problem here is to return the set of entities on the given layer that are related to a set of input entities on any other layer. As the system state is incrementally updated whenever entities are created or destroyed, the application of the algorithm is straightforward.

Asynchronous Requests. Asynchronous requests cause much more effort for the MIMO core. As a result of such requests, a request identifier is returned to the tool and the request is stored in MIMO's request list. The request itself is passed to the concerned instrumentation components, which in turn pass results back to MIMO by means of events. Whenever an event for an active request in the request list occurs, it is passed back to the respective tool.

Unfortunately, requests may not only concern entities that are controlled by the local MIMO instance, but remote entities may also be affected, which makes it necessary to distribute the requests among all active MIMO instances. The resulting procedure for processing an incoming asynchronous requests is as follows:

1. Every request is stored locally in the request list and the request is passed to corresponding local entities.

2. The local MIMO acts as the root MIMO for this request and sends the request to the monitor-monitor event channel to inform all other instances about it.

3. Every remote instance checks whether its local entities are concerned by the request when the request comes in through the event channel. If there are concerned entities, the request is passed to them and stored in the request list.

4. Whenever an event for the request at the root MIMO occurs, it is passed to the tool through the monitor-tool event channel.

5. Whenever an event for the request occurs at a remote MIMO instance, it passes the event to the monitor-monitor event channel.

6. Events coming in from the monitor-monitor event channel are only processed by the root MIMO that has an appropriate entry for this request in the request list; all other instances will discard it.

The benefit of this approach is that only one event channel is required for the interaction among the MIMO instances. The amount of events passed through the monitor-monitor event channel can be high if there are many requests, but this solution represents a trade-off between the number of messages and the amount of data that needs to be stored by each MIMO instance. Here, further optimisations concerning the network traffic and MIMO load may be integrated.

State Change Events. In addition to events generated as response to active requests, there are general events concerning the system state. Therefore, every instrumentation component automatically generates events for the creation or destruction of entities. These events are always passed through the monitor-monitor event channel in order to make it possible for every MIMO instance to update its system state.

Tools are not necessarily informed about these events because they generally query the system state synchronously. Nevertheless, if a tool wishes to be informed about changes in the system state, it can issue an asynchronous request for the creation and destruction events.

6.4.1.4 Synchronisation and Event Ordering

As our monitoring environment is distributed, the synchronisation of the clocks and the ordering of the events ([LS1994], ch. 3) has to be taken into account for the implementation of MIMO. According to [Lam1978], a system is considered distributed if the message transmission delay is not negligible compared to the time between events in a single process. In such a system, time stamps are in general not sufficient for correctly ordering the events because the clocks of different components may deviate from each other. To solve this problem, there are several approaches that can be implemented within MIMO.

Clock Synchronisation using the Network Time Protocol. A first and simple approach to order the events is to make use of the Network Time Protocol (NTP) [Mil1992]. With this protocol, the clocks of distributed computers can be synchronised across the Internet with an accuracy of one millisecond.

The advantage of NTP is that it exists for nearly all operating systems and that it is easy to install and use. No additional measures within MIMO need to be carried out because the clocks generating the time stamps are synchronised. However, the drawback of NTP is that it does not ensure a correct ordering of events:

- Firstly, for applications where deviations of one millisecond are of importance its accuracy is not sufficient.

- Secondly, synchronising the clocks is not sufficient to guarantee a total ordering of events.

In practice, though, synchronising clocks with the Network Time Protocol is often sufficient; thus, our current MIMO prototype relies on NTP synchronised clocks within the monitored environment and does not take additional measures to order the events.

Implementing an Event Ordering Algorithm. If a partial or total ordering of events within the monitored environment is indispensable, an additional algorithm has to be integrated into the monitoring system. For this purpose, several algorithms have been proposed ([Gos1991], ch. 6). The approaches are either centralised or distributed, and make e.g. use of time-based event ordering or token passing mechanisms. Elementary examples for distributed, time-based algorithms are those of Lamport [Lam1978] and Ricart and Agrawala [RA1981]. Both of them are able to reach a total ordering of events, but need to exchange $3(N-1)$ and $2(N-1)$ (where N is the number of participating nodes) messages for every event.

Another algorithm for implementing a virtually synchronous system is the CBCAST algorithm used within the ISIS system [BJ1987], with which a consistent time order is reached by introducing vector clocks. With the ISIS approach, the order in which events are perceived is equal for all participants, and causally connected events are ordered according to their dependencies. The algorithm for obtaining this ordering works as follows:

1. Every participant within the system is assigned an identification number and a state vector.

2. Every participant stores a state vector containing the number of the last message being received from every other participant.

3. If any participant wants to send a message, it increases its message number by one and sends its complete state vector with the message to all other participants.

4. If a new message arrives at any participant P, it has to check two conditions to receive it:

   - First, all preceding messages from this sender must already have been received.
   - Second, it has to be assured that all messages that the sender has already received from any other participant have also been received by P.

   These conditions can be checked by using the state vector sent with the message. If they are fulfilled, the message can be received, and if not, it has to be delayed until the conditions are fulfilled after the reception of other messages.

The advantage of the algorithm is that causal connections are always reflected in the event ordering, while the number of messages to be sent for every event

is $N - 1$. For example, this is important when two events for the sending and receiving of a remote method call are generated; if the monitoring system perceived the second event before the first event, additional effort would be needed to solve this inconsistency.

The disadvantage of the algorithm is that it requires the exchange of the state vectors with every event, which causes a considerable overhead. Another disadvantage of all such ordering algorithms is that every event needs to be distributed to every other participant. However, with our MIMO prototype, the second problem would not occur because events can easily be broadcast through the monitor-monitor event channel.

Finally, a third problem within MIMO would be caused by the fact that the number of participants varies over time because of the dynamic nature of our environment. Therefore, additional measures adapting the algorithm for a varying number of participants would have to be integrated. This would increase the overhead further as for every new participant a synchronisation message announcing its existence needs to be broadcast to all participants.

Using the CORBA Time Service. Another approach to synchronise clocks or to implement an event ordering mechanism is to use the CORBA TimeService [OMG2000c]. A useful scenario for implementing clock synchronisation is the Timer Event Service described in the TimeService specification. The components participating in this scenario are as follows:

- A logical Timer Event Service object generates the actual time values.
- These time values are sent to Timer Event Handler objects through push-style CORBA event channels.
- Actual Timer Events are attached to the Timer Event Handlers, which allow to set timers and trigger the events in case of occurrence.

The resulting structure of the TimeService is shown in figure 6.13. This architecture of the TimeService can be used to build a synchronisation system for the clocks. Unfortunately, currently only few CORBA implementations like e.g. MICO [PR2000] support the CORBA TimeService, prohibiting its deployment for most environments. Moreover, the initial Request-for-Proposal (RFP) for the CORBA TimeService also contained suggestions for event ordering mechanisms, but these were not included in the final specification. Thus, the ordering problem can only be solved by an implementation of the algorithm outlined in the previous paragraph.

To sum up, it can be seen that clock synchronisation can be achieved using several approaches, but no total event ordering can be reached without manually integrating a costly algorithm that introduces vector clocks. Therefore, the current MIMO prototype relies on NTP synchronised clocks, while the usage of the CORBA TimeService to synchronise the clocks could also be integrated if no NTP is available in an actual environment.

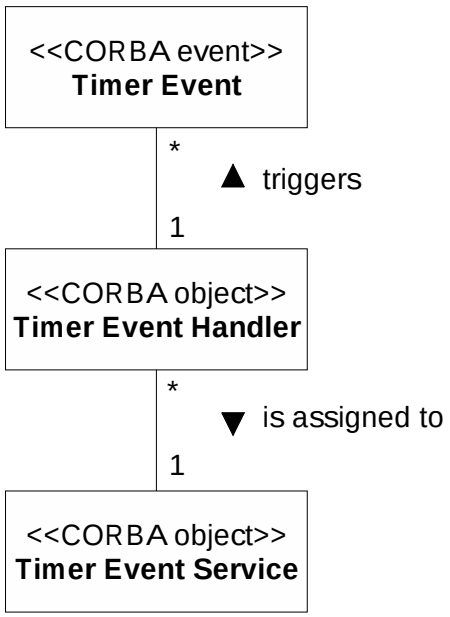

Figure 6.13 CORBA TimeService

6.4.2 Tools

Tools making use of the MIMO system can be very diverse due to the complex nature of monitoring scenarios. MIMO only requires tools to make use of the standardised interfaces to access its functionality, but beyond this, no restriction is given. As an example tool, we here illustrate the MIVIS visualisation tool, which has been developed together with the MIMO prototype.

6.4.2.1 MIVIS Visualisation Tool

The MIVIS visualiser [Rud1999] represents a basic tool for demonstrating the multi-layer monitoring approach of MIMO. Above all, its capabilities within complex heterogeneous environments – where scalability and a uniform view on the observed system are important – are shown.

On the one hand, MIVIS is based on the analysis of other visualisation approaches like VISTOP [Bra1994], DePauw [dPHKV1993], and CORBA-Assistant [Fra1997]. The main requirements resulting from this analysis are scalability, uncomplicated extensibility, platform independence, an ergonomic user interface, and the possibility of having several displays at a time. On the other hand, the visualisation tool should comply with the multi-layer monitoring approach in order to support its advantageous usage.

MIVIS Concept. The general problem of visualisation is scalability because huge amounts of data have to be presented in a way that allows the observer to keep track of the information offered. Thus, it has to be possible to reduce data by means of filtering mechanisms. MIVIS realises this reduction by using its selection mechanism that provides a kind of filtering based on the multi-layer monitoring model.

Figure 6.14 shows a screenshot of the MIVIS tool. All entities in the monitored application are shown inside the selection frame. Each layer of the multi-layer monitoring model is represented in one tab of a tabbed pane. The user can select entities within the different layers and thus control the granularity of the visualisation. To gain an overall insight, monitoring the system on the application layer or the hardware layer can be carried out without exposing any details. To get more insight into the internals of the application the user can pick out a few interesting entities and go up or down to adjacent layers to get more detailed information.

MIVIS Displays. Up to now, the three display types text display, scroll display, and call frequency display have been implemented:

- The text display prints out the events that are monitored in plain text, which can basically be used for logging purposes; details that might not be visible in a graphical display can be looked up here at a later time.
- The scroll display visualises communication between entities. The selected entities are displayed on the y-axis in a coordinate system. The x-axis shows the time. When an entity communicates with another entity, an arrow between the two is shown in the coordinate system.
- The call frequency display visualises communication in a different way: Only cumulative data containing the number of calls are of interest and thus shown as a vertical beam for each caller and for each called entity.

These displays only represent fundamental aspects of an application that might be of interest, while others can easily be added depending on the respective monitoring scenario.

6.4.2.2 A Tool Framework Based on MIVIS

When we look at a set of common tools for distributed systems, it is remarkable that a given amount of basic functionality is present within most tool sets. This includes for example the basic visualisation capabilities for gaining an overview of the observed system, or logging functions. Beyond this, tool functionality can be very diverse and highly depend on the type of tool and middleware. Hence, as we wish to build tools for heterogeneous middleware with our approach, a tool framework offering these basic functions on the one hand, and being dynamically extensible with any kind of tool plug-ins on the other hand, results to be profitable. Especially for GUI tools, basic visualisation capabilities are needed in almost any case and serve as a starting point for further functions.

Within MIMO, due to its design and implementation concept MIVIS can serve as such a kind of environment with respect to the construction of GUI-based tools.

A Tool Framework Using JavaBeans. As MIVIS is completely implemented in Java, platform independence and portability is given, which represents an important criteria for a universal tool framework. To fulfil the requirement of uncomplicated extensibility of the framework, it is split into a main program and several JavaBeans

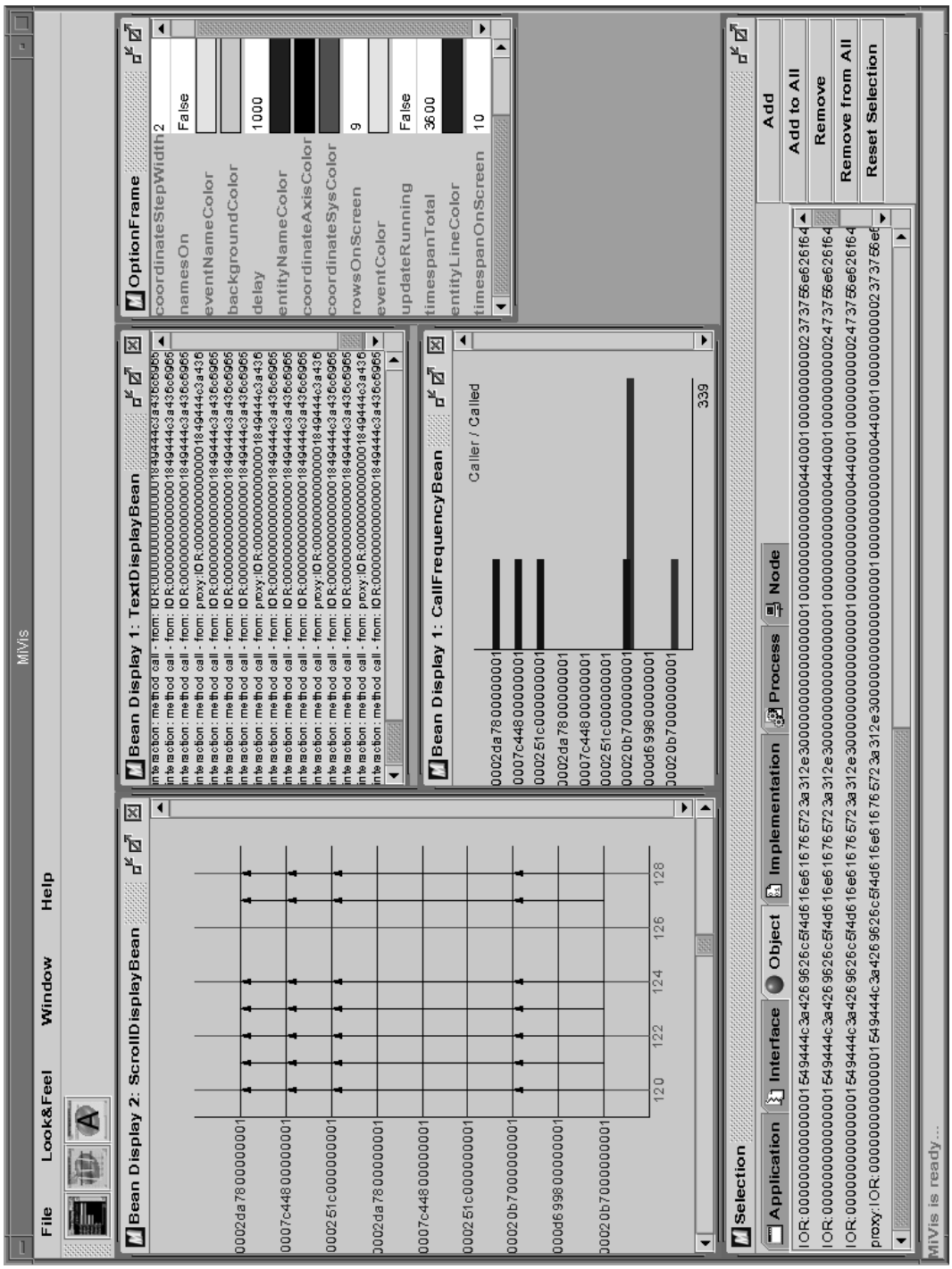

Figure 6.14 MIVIS Screen Shot

[Ham1997] software components. The main program takes care of the communication with MIMO and the processing of the data, and the JavaBeans take over the graphical presentation.

All JavaBeans are discovered by MIVIS at startup time and are dynamically

integrated into the GUI. If a different type of display is needed, a user can integrate that display type using Java and turn it into a JavaBean. This component is placed into a specific directory intended for MIVIS to find and use it. With this approach, the main program does not have to be changed or recompiled at all, the only requirement is that the JavaBean implements a minimal interface that enables the main program to communicate with the bean.

The bean-specific properties can be set by the user, while MIVIS knows about these properties by means of the introspection mechanism and provides editors to change the settings of these properties. Additional editors for properties of a special data type can be placed inside the JavaBean and used instead of the standard editors. All properties together with their editors are shown inside the Option Frame, where MIVIS allows the user to edit properties unknown to the framework at compile-time. This approach offers a very dynamic and flexible way to configure the behaviour of various display types; the concept of separating the display types from the main program makes it very easy to generate new display types for MIVIS without the need of changing the original code.

Consequently, adding new tool functions is possible by simply implementing a new JavaBean that offers a minimal interface for the MIVIS framework. Subsequently, it can make use of the complete MIVIS features, including intelligent selection mechanisms and standard properties; therefore, this “plug-in” approach enables an efficient usage of the MIVIS framework.

Summing up, MIVIS represents a basic visualisation tool for applications monitored by MIMO and additionally serves as a framework for integrating further tool functions depending on the actual deployment scenario. This concept provides a major contribution to the tool development methodology that will be presented in chapter 7.

6.4.3 Instrumentation

The instrumentation of applications can be implemented in a great variety of ways. A taxonomy of existing instrumentation strategies can be found in [MBBD1999]. In this section, we are presenting three examples of the integration of common middleware platforms.

6.4.3.1 Intruders and Adapters

For our MIMO implementation, two general approaches of instrumentation can be distinguished according to the transparency with respect to the instrumented application.

- *Intruders* are instrumentation components that are transparently integrated into the monitored application. Such instrumentation can be implemented on different transparency levels: Transparency can either mean that the application programmer does not need to integrate code into the application, but the source code gets preprocessed before it is compiled in order to add monitoring code, or, on a higher transparency level, the compiled application is

instrumented without recompilation by means of library or operating system instrumentation. In our case, we only define intruders as components that do not require manual instrumentation of the source code.

- *Adapters* in contrast are instrumentation components that are non-transparently integrated into the application. They can e.g. be implemented by inserting additional monitoring code into the application, which are then prepared for further monitoring from the beginning.

Subsequently, we will present two intruders for the CORBA and DCOM middleware, followed by a generic Java adapter that is suited for any kind of Java-based middleware.

6.4.3.2 CORBA Intruder

The CORBA intruder for MIMO applications [Ort1999] is able to transparently monitor CORBA applications written in C++. It relies on the instrumentation of CORBA implementation libraries [Lev1999] that are linked to the actual application components; in these libraries, CORBA method calls are wrapped by instrumentation methods, which interact with the MIMO core and then invoke the original method.

Implementation for ORBacus. Our prototypical intruder is implemented using the ORBacus C++ ORB [Obj2000]. It does not create a separate thread for the instrumentation, so that interaction with the MIMO core is only possible at positions where instrumented methods are called from within the application. Among the instrumented CORBA methods are functions indicating the creation and destruction of CORBA objects, as well as communication methods allowing to detect the interaction of different CORBA objects. As our instrumentation resides on the CORBA implementation level, both client- and server-side calls to CORBA functions can be detected, which is hardly possible with most existing CORBA monitors.

Micro-Benchmarks. To measure the overhead of the instrumentation caused by the CORBA intruder, several micro-benchmarks describing the behaviour of important CORBA method calls have been carried out. Major CORBA methods to be observed are those indicating the creation or destruction of objects and communication among objects. To evaluate the performance, several scenarios with a varying number and location of MIMO instances for the participating components have been used. Figure 6.15 summarises the results with average values from the different monitoring scenarios[3]. The measured values represent the time needed for calling object creation, object deletion, and interaction methods, either with or without the instrumentation ([Ort1999], ch. 6).

The average overhead shown in the diagram is about 80% for creation and deletion of CORBA objects and approximately 60% for interactions between objects.

[3]The measurements were carried out on single processor 300 MHz Sun Ultra-10 workstations with 192 MB, the Solaris 2.6 operating system, connected through a 100 MBit Ethernet network. The ORBacus version used for the experiments was ORBacus 3.2.

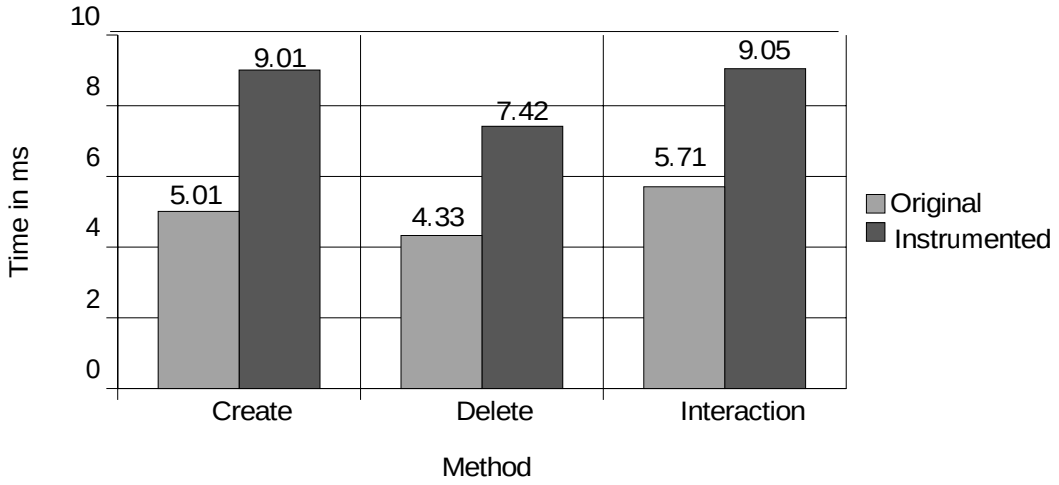

Figure 6.15 CORBA Intruder Overhead

These overhead values are relatively high because for most CORBA method calls, an additional call to the MIMO instance is necessary. As this call is implemented using the same middleware as the observed application, it is not negligible compared to the original invocation time. However, for most applications, the overhead caused for creation and deletion operation is not critical because such operation are not carried out frequently. For interactions, in contrast, this overhead may cause serious delays for the applications. Therefore, this proceeding is not applicable for situations with a high number of monitoring events.

Nevertheless, these measurements represent values collected with our non-optimised implementation; the overhead can therefore be decreased by setting up local MIMO instances for performance critical applications components, and by intelligently monitoring the application in order to minimise the number of events sent to MIMO.

6.4.3.3 DCOM Intruder

For monitoring DCOM applications, a DCOM intruder [Süs1999] has been developed. Generally, there are various possibilities for retrieving data from DCOM applications that can be integrated. For our prototype, we have implemented an approach that relies on wrapping DCOM objects with a universal object wrapper.

Implementation of the DCOM Intruder. The wrapping process used for our DCOM intruder makes use of an interception framework that is based on a *universal delegator* [Bro1999a], [Bro1999b]. With this delegator, calls to DCOM interfaces are passed to a delegator hook for monitoring purposes before delivering them to the original object. Our DCOM intruder uses this technique for passing calls to observed objects to a COMService that contains the local monitoring component. The COMService is implemented as a Windows NT service that runs with one instance on every monitored machine. It encloses a COMLog component for processing the events and a COM-CORBA bridge component that transforms the data into the

format required by MIMO.

Performance. The outlined implementation already suggests a high overhead of this instrumentation approach, as it comprises internal COM communication with an additional data transformation to CORBA. Figure 6.16 illustrates performance data showing the overhead of the DCOM intruder for local and remote DCOM method invocations. The test case consists of a local or remote caller invoking a method on a machine running a DCOM object that is observed by our COMService, where also a local MIMO instance is present[4]. The measured times are for 20 subsequent calls to the monitored object.

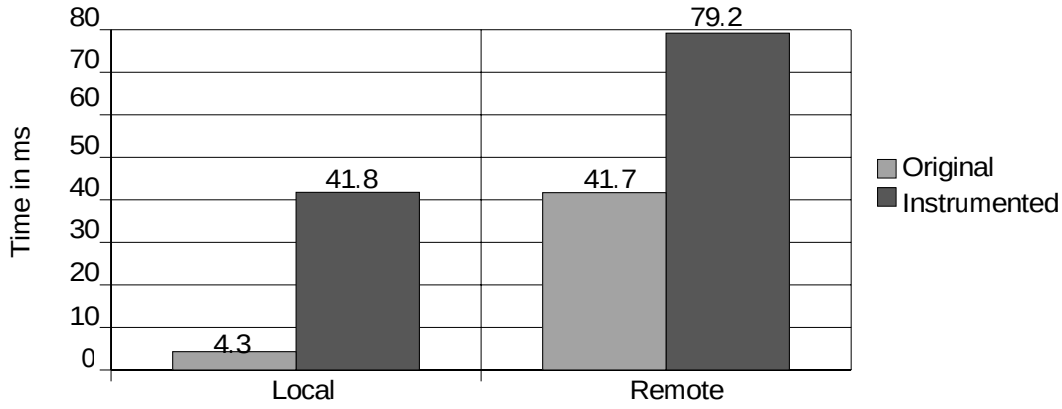

Figure 6.16 DCOM Intruder Overhead

The resulting overhead for remote calls is approximately 90% for remote calls and 870% for local calls. While for remote calls the overhead is still acceptable, our approach is not feasible for observing local COM calls. As already indicated, the main reasons for the poor performance are the frequent local inter-process COM calls and the additional overhead for transforming the data to CORBA calls for MIMO. In the local case, the percental overhead is unacceptably high because local COM calls are optimised by the COM runtime libraries, whereas our transformation to CORBA and the subsequent communication with MIMO are not optimised for local usage.

Nevertheless, several optimisations are possible for improving the performance of our DCOM intruder:

- First, the universal delegator used for implementing the object interception mechanism is designed for very general applicability and could be optimised by deploying it without separating the delegator and COMLog component by COM interfaces, which introduce an additional abstraction level.

- Second, our COM-CORBA bridge is relatively primitive and does not include any optimisations for data conversion. Integrating a sophisticated COM-CORBA bridge here yields considerable performance improvements.

[4]The machine used for the test case was a PC with a 266 MHz Pentium II processor, 128 memory, running Windows NT 4.0; the network connection was a 10 MBit Ethernet.

- Third, COM+ [Pla1999] – the successor of COM – provides an extended support for monitoring and managing COM applications. Therefore, applying the universal delegator for wrapping COM objects will lapse in any case.

Hence, if we take these criteria into account for further performance improvement, monitoring DCOM applications with the DCOM intruder represents a feasible approach with certain limitations. For management tasks or object supervision tasks where events occur rather infrequently, the DCOM intruder can be applied smoothly. For high-performance computing scenarios or performance critical tasks, additional effort is necessary for reducing the overhead, which can e.g. be done by building more intelligent intruders that minimise the event rate as far as possible.

6.4.3.4 Generic Java Adapter

Another instrumentation component implemented with the MIMO prototype is a generic Java adapter. As mentioned above, adapters can be used for integrating monitoring tasks into the source code of applications. Our generic Java adapter therefore represents a Java class library containing functions for easy access to the MIMO core. For example, this comprises functions for looking up and attaching to MIMO, as well as sending and receiving events to and from MIMO without taking care for the underlying CORBA communication. An example for the deployment of our Java adapter will be shown in section 7.2.2.

Moreover, it should be mentioned that the implementation of a Java intruder based on the Java adapter is straightforward. The technical implementation of instrumenting Java class files can be carried out by wrapping methods in the class files; as the Java class file format is well defined, this can be reached in a clean and portable way.

6.4.4 Conclusion

In this section, we have outlined the implementation of the MIMO prototype. Beginning with the start-up procedures and the description of the MIMO core, we have subsequently shown the MIVIS visualisation tool and its suitability as a general tool framework, as well as examples for possible instrumentation mechanisms for general purpose middleware.

Regarding the prototype implementation, performance improvements could be reached by integrating the various optimisations mentioned above. However, new technologies like COM-CORBA bridges or the release of COM+ will foil some insights because critical parts will change with these techniques. Using all these techniques still does not yet solve some problems that limit the performance of the MIMO implementation; the performance of CORBA for passing events is inherently limited with existing CORBA implementations, especially when complex data types have to be marshalled and unmarshalled.

Therefore, it can be seen that monitoring middleware applications with a monitoring system that is based on middleware techniques of the same granularity yields a non-negligible overhead with regard to performance aspects. This overhead is the price for the flexible and extensible usage of the monitoring system and has to

be taken into account when instrumentation components are designed and implemented.

With regard to our overall monitoring scenario, we found out that *organisational aspects* are extremely important for the cooperation of monitoring components. The set-up of the monitoring environment including the distribution and assignment of MIMO instances, their registration and look-up procedures, needs to be managed before any monitoring action can actually occur. Hence, a main result of our implementation is that an advantageous organisation of the monitoring environment is of crucial importance within dynamic and heterogeneous environments. For deploying tools efficiently, organisational aspects therefore play an immensely important role, especially for performance reasons.

6.5 Summary

This chapter has illustrated the details of the MIMO architecture, its usage, and implementation. The architecture is layed out very generally in order to fulfil the requirements stated in chapter 4. After a short introduction of the overall tool development and usage process, we have presented MIMO's architecture, which consists of the three main components MIMO core, tools, and instrumentation components. The usage of the MIMO core – which implements the multi-layer monitoring approach – has been illuminated from both tool and instrumentation side.

Concerning the implementation of the MIMO prototype, the basic internal communication principles used for request and event distribution have been shown; here, the problem of clock synchronisation and event ordering is an important aspect, for which several solution approaches within our environment have been explained. From the tool point of view, the MIVIS tool framework represents a general purpose framework for building GUI tools efficiently. For the instrumentation side, intruders for the CORBA and DCOM middleware have been developed as well as a generic adapter for Java based middleware.

In the next chapter we will continue with a methodology for efficiently developing tools based on the MIMO implementation and several real-world application scenarios.

Chapter 7

Tool Development and Application Scenarios

In this chapter, we will present a methodology for efficiently developing tools using the MIMO monitoring system, and we subsequently demonstrate the applicability of our methodology by means of several real-world application scenarios.

So far, we have explained the basic idea of the MIMO approach, followed by the implementation of the monitoring system. As the resulting system is complex and leaves a lot of freedom to the tool developer, a methodology for efficiently using it in practice is required. Therefore, we now derive a tool development methodology that can easily be followed by tool developers in order to build tools systematically and rapidly. Our methodology consists of a predefined sequence of steps that lead to the integration of new middleware platforms to the MIMO system. Depending on the respective middleware environment and tool needs, these steps can be carried out iteratively in order to augment the middleware integration or tool functionality, so that an incremental tool development process is accomplished.

Furthermore, we will present several application scenarios that demonstrate the applicability of our approach to real-world scenarios. The spectrum of our examples includes the Globus grid computing environment, the LSD metacomputing system, the SEEDS airport simulator, and a medical image processing application. In addition to our general purpose middleware (CORBA, DCOM) instrumentation shown in the previous chapter, these examples exhibit the flexibility of the MIMO system under varying deployment conditions.

Hence, the quintessence of this chapter is to prove the advantageous usage possibilities of the MIMO system, both on the methodical level and from the practical point of view represented by real-world applications.

7.1 Tool Development Methodology

A methodology that describes the proceeding for the development of new tools or the integration of new middleware is indispensable for the MIMO system. This section first motivates the proposal of a tool development methodology, then gives an outline of the methodology itself, and finally considers some implementation aspects.

7.1.1 Motivation

Most monitoring systems do not include a methodology for their efficient usage. However, when we consider the MIMO system, there are several reasons that recommend the definition of a tool development methodology:

- Firstly, the MIMO monitoring environment is a rather complex system, which leaves many degrees of freedom to the tool developer. For example, the requests and events going beyond simple system state queries are completely customisable, so that tools and instrumentation need to be adapted to each other. Therefore, a tool development methodology can serve as a guidance for tool developers that describes a fixed procedure for developing tools or integrating middleware. With this approach, a systematic usage of the MIMO infrastructure is made possible without confusing tool developers about how to deploy MIMO's facilities correctly.

- A second reason for MIMO's tool development methodology is to enable *rapid tool development*. In analogy to the general software development process (see [BH1996], [McD1991]), where it is considered important to build prototypes of running programmes rapidly, MIMO takes up this concept and allows to build tools and instrumentation components very quickly. Moreover, the tool development methodology is layed out to support an iterative development or the incremental extension of tools and instrumentation components. Hence, with MIMO's tool development methodology it is possible to build tool prototypes rapidly and to extend them incrementally during their deployment.

- Another important point considering the tool development within heterogeneous systems is to maintain the comparability of the data retrieved from various middleware platforms. To reach such a comparability, it is necessary to define the mapping of the middleware platform to the multi-layer monitoring model. The tool development methodology takes over this task and standardises the integration of heterogeneous platforms.

- A further aspect concerning the heterogeneity of the observed middleware is to enable reusage of developed instrumentation components or tools. For example, the integration of some Java-based middleware might be reused for future Java-based environments. Such a reusage mechanism can only be implemented if the design and implementation follows some basic predefined rules, which are defined within the tool development methodology. The result of this reusage mechanism is to provide a kind of self-extending monitoring infrastructure containing instrumentation components or tools that can be reused for future monitoring scenarios.

- A last significant aspect of our tool development methodology concerns the implementation of tools and especially instrumentation components. As performance can be critical in some observed applications, it has to be assured that implementations do not produce too much overhead. Our methodology

supports this requirement by its incremental instrumentation process, which reduces instrumentation to the essential parts without losing its extensibility. This means that only the required information should be collected from the observed system, while a simple and rapid introduction of further functionality is preserved.

Considering these criteria, the integration of our tool development methodology into the MIMO system is justified from several points of view. Especially for modern operating environments where middleware and applications change frequently, the possibility to rapidly adjust to new circumstances is highly desirable.

7.1.2 Rapid Tool Development Methodology

After justifying the need for our tool development methodology for MIMO, the concrete model for developing tools with MIMO can now be established.

7.1.2.1 The Tool Development Lifecycle

Figure 7.1 illustrates the major steps of MIMO's tool development methodology [RL2001] that make up the tool development lifecycle within MIMO.

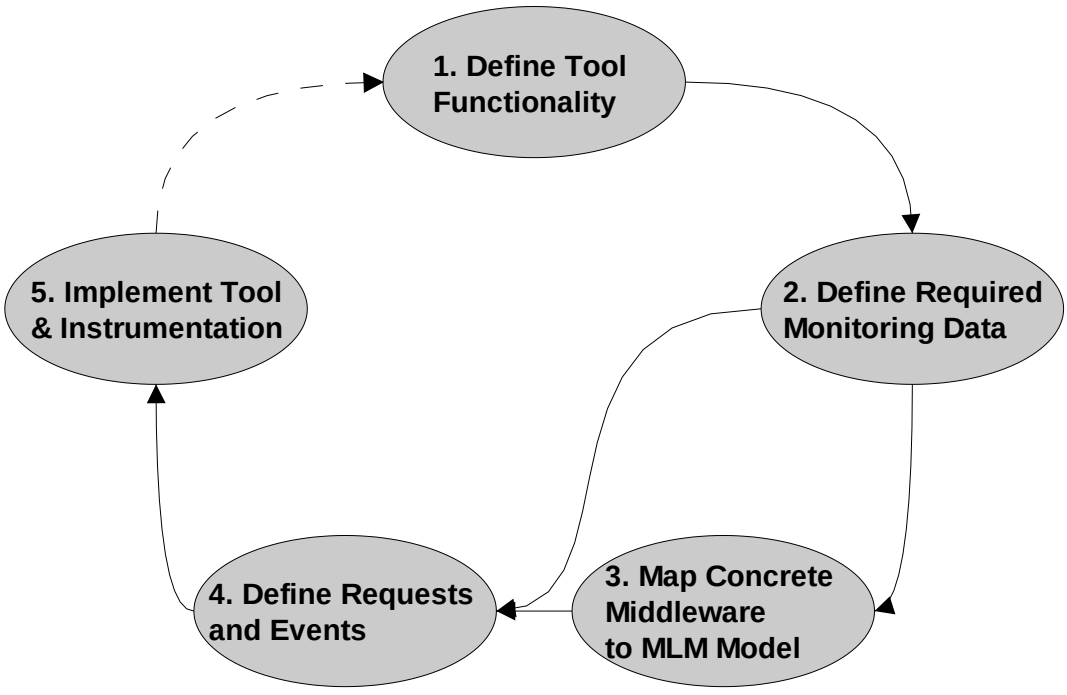

Figure 7.1 Steps of the Tool Development Methodology

Basically, the methodology consists of steps for the definition of tool functionality, the resulting data that need to be exchanged, and the requests, commands, and events derived from this definition, as well as a mapping of the respective middleware to the multi-layer monitoring model, and the implementation of tool and instrumentation.

7.1.2.2 Roles of Developers and Users

With reference to section 6.1, where the use case for the tool development and deployment process has been shown we can now come back to the roles defined there. The definition of tool functionality is mostly carried out by the tool user who wishes to observe certain aspects of the system; also, it might be supported by the application developer, who may cooperate with the tool user during application development. As mentioned before, the two roles can also be mapped to a single person in some application scenarios.

The definition of the required monitoring data, the mapping to the multi-layer monitoring model, and the definition of the requests, commands, and events is carried out by the tool developer. He or she has to consider both the tool user's perspective as well as MIMO's classification scheme. Like before, the tool developer can be supported by or even be identical with the application developer for these tasks. The tool implementation is finally carried out by the tool developer, whereas the implementation of the instrumentation can both be taken over by both the tool or application developer.

Considering the fusion of different roles to single persons, the following guidelines can be given:

- For tools like debuggers, which are developed independently from certain application scenarios, the tool developer is separate from the tool user and application developer, whereas the tool user often merges with the application developer in this case.

- For tools being delivered with software products, for example for management tools that support the software deployment, the application developer is frequently identical with the tool developer.

- For very specialised tools that are neither provided with applications and that are no standard tools, the tool developer and tool user may merge. Consequently, in this case, the tool user develops his or her own tool because there is no other possibility for getting the desired tool functionality.

Moreover, as already mentioned before, the three roles can also be mapped to three disjoint persons, which is imaginable especially for large projects.

7.1.2.3 Steps of the Tool Development Methodology

The detailed tasks of the single steps of our tool development lifecycle are as follows:

Definition of Tool Functionality. The first step of the methodology defines the tool functionality in a relatively abstract way. For example, this task includes the definition of the tool type, whether it is interactive or automatic, or whether it only observes an application or also manipulates it. Here, the tool user's view on the system serves as a starting point for all considerations, while the definition of functionality is kept coarse.

Definition of Monitoring Data In the second step, the coarse description of the functionality is refined to refer to more technical aspects of the observed environment. Data that need to be retrieved from the application have to be determined as well as commands that need to be issued by the tool in order to accomplish the tool functions.

Middleware Mapping to the Multi-Layer Monitoring Model A very important aspect of tool development is the integration of the middleware platform to be monitored into MIMO's multi-layer monitoring model. In order to perform this task, the tool developer needs to consider the layering given in the new middleware platform and map it to the best matching layers of MIMO's model according to the tool requirements.

Most middleware platforms exhibit a clear layer model, although it is not always made explicit. The decisive point for the mapping is to reflect MIMO's philosophy of layer definitions for determining the corresponding layers within the new middleware that has to be integrated. As a starting point for the mapping, it is often useful to consider the *distributed objects* layer because it possesses a central position for middleware-based applications. From this layer, the tool developer can move upwards and downwards to the adjacent layers. The interface and implementation layer also depend highly on the respective middleware platform with their interaction and implementation policies for distributed objects, whereas for many common systems, the application, runtime, and hardware layer are determined by the notion of the monitored application, the operating system processes, and the computing node.

Moreover, the mapping of middleware layers to MIMO's layer model influences the comparability of data coming from different middleware platforms, because entities within the same layer should correspond to each other even if they originate from different middleware platforms. As a consequence, it is essential to keep MIMO's philosophy of layers in mind when integrating a new middleware platform.

Definition of Requests, Commands, and Events Once the definitions of the monitoring data and the mapping to the multi-layer monitoring model have been carried out, the concrete requests, commands, and events to be exchanged between tool and instrumentation need to be defined. This comprises the syntax as well as the semantics of the transferred data, including their parameters and attributes.

The decisions taken during this phase are also important from reusage aspects of tools and instrumentation: When standardised names for requests, commands, and events are chosen, tools can possibly be reused for new middleware platforms that are instrumented using the same syntax and semantics, so that the tool functionality can easily be transferred to these platforms. Conversely, newly developed tools can rely on existing instrumentation components if they all support the same requests and commands, which enables a simple reusage of already implemented instrumentation code. Hence, as using well-defined syntax and semantics for requests, commands, and events is essential for reusage purposes, we will elaborate this aspect in more detail in the next section.

Implementation of Tool and Instrumentation The final step of the development methodology is the implementation of tool and instrumentation. Here, the decision of implementing the tool as a stand-alone program or as part of the MIVIS framework has to be taken. For GUI tools, being part of MIVIS is in most cases simpler, whereas for automatic and non-interactive tools like e.g. load balancers a stand-alone program being executed in the background is preferable.

For the instrumentation, the developer needs to decide whether to implement an intruder or an adapter. Depending on the respective middleware and tool functionality, both approaches may be reasonable as we have shown in section 6.4.3. Furthermore, the implementation of instrumentation code is influenced by performance considerations that have to be taken into account in order to limit the overhead of the instrumentation.

7.1.2.4 Iterative Tool Development and Incremental Tool Extension

As we have seen, tool development with MIMO's methodology is a straightforward process that leads from the definition of tool functionality to the implementation of the tool. However, as tool functionality is often complex, the implementation of the tool itself and the instrumentation are costly. Therefore, our methodology also allows to implement basic functionality and to iterate over the tool development lifecycle in order to incrementally add new functions to tool and instrumentation. With this approach, a rapid prototype of a tool can be implemented quickly in order to prove the feasibility of the implementation, while a following extension is made possible without having to discard previously implemented functionality.

The step that needs to be carried out only during the first iteration is the mapping of the middleware to the MIMO model. This mapping should be kept constant as it represents a basis for the subsequent steps, which would have to be changed entirely when the mapping changes.

Furthermore, when new requirements arise during the deployment of a tool the lifecycle can be traversed again, starting with a redefinition of tool functions based on the new circumstances.

7.1.3 Implementation Aspects

After specifying the general tool development methodology, some details concerning the implementation of the illustrated concepts remain.

For the definition of requests and events, two classes of queries have to be distinguished: Firstly, system state queries that are processed by the MIMO core are standardised by MIMO and hence identical for all tools and instrumentation components. Secondly, all other requests not concerning the system state are user-defined and only propagated intelligently by the MIMO core. For this second class of requests and events, we have found that it is beneficial for reusage purposes to standardise the syntax and semantics for a set of often occurring monitoring tasks. Therefore, we will sketch the shape of such a standardised request and event set for middleware environments.

7.1.3.1 System State Queries.

The layout of system state queries has already been outlined in section 6.3.5. Such requests allow to query the existing entities within one of the MIMO layers, constrained by some additional parameters; the request names for the respective layers are as shown in table 7.1.

Layer	*Request Name*
Application Layer	`get_application`
Interface Layer	`get_interface`
Distributed Objects Layer	`get_object`
Implementation Layer	`get_implementation`
Runtime Layer	`get_runtime`
Hardware Layer	`get_node`

Table 7.1 System State Query Requests

As a parameter to these requests, a list of entities can be provided. The result consists of all entities within the required layer that are related to the given input entities according to the *related-entities* algorithm described in section 5.1.6. The system state queries may be issued synchronously or asynchronously, where in the synchronous case the current systems state is considered, and in the asynchronous case the query is applied to all newly created entities. The format of the results being passed back synchronously or by means of events is also well-defined.

7.1.3.2 User-Defined Requests and Events.

Queries not accessing the system state are not touched by the MIMO core, but only propagated to involved entities. Tool and instrumentation need to be adjusted in order to agree on the syntax and semantics of exchanged data. Nevertheless, most tools make use of common monitoring tasks, so that a definition of a standard set of requests is advantageous. Common tasks may e.g. include the following functions:

- For observing the communication between entities, a request called `get_interactions` is useful; as a parameter, it takes a list of entities for which communication activities should be reported. In case of communication, an event describing the communicating entities and their exchanged data is triggered.

- For retrieving information about certain attributes of monitored entities, an operation named `get_attribute` can be used, which takes the name of an attribute as a parameter. Such attributes may e.g. concern the load of an entity or other internal aspects.

The definition of further requests and events highly depends on the application environment of the monitoring system. For example, manipulating commands that initiate a migration of objects can only be applied if the underlying middleware

is capable of this feature. Hence, we cannot give an extensive list of all possible requests and events, but supporting a standard set of operations should be kept in mind when designing and implementing a tool environment.

7.1.3.3 Usage of Standardised Descriptions.

Another relevant aspect during the implementation of a tool environment is to rely on standardised descriptions for certain components. Approaches like the *Management Information Bases (MIBs)* defined with SNMP can be used for the description of attributes that have to be queried with MIMO. As SNMP only provides a set- and get-operation, reading or writing such parameters can easily be emulated within MIMO. For components where MIBs are available, like e.g. CORBA environments based on Orbix [Ion2000], these can be used for monitoring with MIMO, too.

7.1.4 Conclusion

In this section, we have presented MIMO's methodology for developing tools and integrating new middleware. As our monitoring infrastructure is complex and leaves many degrees of freedom to the user, the methodology enables a rapid tool development by giving a systematic standard procedure to the tool developer. Tool development gets "demystified" because our methodology serves as an orientation for developers to construct new tools efficiently.

What remains is the question whether an even more customisable infrastructure could be useful. For example, the multi-layer monitoring model itself could be made customisable and provide a varying number of layers. However, with such extensions, the comparability of data coming from various middleware platforms would be lost, and hence one of the major advantages of MIMO would disappear. Furthermore, an even more customisable infrastructure would be more complicated to use and prevent developers from making use of all features. Therefore, MIMO's approach represents a well suited degree of generality for middleware-based environments, while the tool development methodology allows to build tools rapidly and efficiently.

7.2 Application Scenarios

Our proposed tool development and middleware integration methodology can be used for adapting very diverse kinds of middleware to the MIMO environment. In this section, we will show several examples for the applicability of the MIMO approach to integrate heterogeneous middleware into the observed environment.

After having presented how to monitor general purpose middleware like CORBA or DCOM in section 6.4.3, we now proceed with more specialised middleware platforms. Our application scenarios include the two metacomputing platforms Globus and LSD, the SEEDS environment for distributed, interactive real-time simulation, and a medical image processing application. For all these platforms, we will show their mapping to the multi-layer monitoring model and implementation issues concerning the instrumentation of components within the respec-

tive systems. A quintessence of this section is the insight that the integration of different platforms can be extremely diverse, and moreover that – due to different monitoring goals – instrumentation possibilities are manifold even within a single middleware platform.

7.2.1 Globus

This section describes the design and implementation of a Globus adapter for MIMO [Rac2000], i.e. a component for collecting information from Globus applications and delivering it to MIMO in an appropriate way.

7.2.1.1 Monitoring the Globus Job Submission Procedure

The Globus metacomputing system has already been described in section 2.2.3. An important aspect of Globus is the distribution of jobs within the set of available resources. For this purpose, on every host participating in the Globus environment a gatekeeper controls the authentication and authorisation processes for users requesting to start jobs on this host. Once a user is granted resources on a machine, the Globus job manager is responsible for launching the specified job on this machine. Hence, monitoring the gatekeepers and job managers allows to observe the submission and distribution of jobs within Globus. Besides other relevant aspects, monitoring the job submission procedure is therefore a major instrument for gaining an overview of a running Globus environment. Our proceeding follows the tool development methodology by defining the tool functionality and the derived monitoring data, mapping Globus to the multi-layer monitoring model, and implementing the resulting request and event types.

Consequently, the first steps are to define the tool functionality and the monitored data needed for this purpose. As we are concentrating on the integrative aspects of the MIMO environment here, we wish to visualise the job submission procedure of Globus. The processes that have to be monitored for this purpose are the authentication and authorisation at the Globus gatekeeper, and the job submissions at the Globus job managers. Whenever the gatekeeper or job manager is contacted from a potential client, events for MIMO need to be generated, if requested. The data delivered with the events may contain additional details about the user wishing to submit a job, or further details of the job specifications.

7.2.1.2 Multi-Layer Monitoring Mapping for Globus

Secondly, the Globus middleware needs to be mapped to the multi-layer monitoring model. The result is the assignment of MIMO's abstract entities to concrete Globus entities shown in Table 7.2.

As we wish to observe the job submission procedure of Globus, our mapping mainly concentrates on running Globus jobs that are started on different hosts by using the gatekeeper's interface to authenticate and authorise a user. Other Globus information sources like the heartbeat monitor or MDS could also be integrated into the model, but have not been taken into account here.

MIMO MLM Layer	*Globus Entities*
Application	*Globus applications*
Interface	*Globus gatekeeper*
Distributed Objects	*Globus job identification*
Implementation	*Local executable*
Runtime	*Local PID*
Hardware	*Node*

Table 7.2 Mapping of Globus Entities to the MIMO Model

7.2.1.3 Globus Adapter Design and Implementation

The Globus adapter delivering data to MIMO has to use MIMO's CORBA interfaces for attaching and detaching, and CORBA event channels for asynchronous communication. Therefore, an ORB handling the communication with MIMO needs to be integrated into the adapter. As the Globus adapter needs to be integrated and linked with Globus components, it is conveniently implemented in C like the Globus components used for the prototype implementation. Hence, the following criteria were important for selecting a C ORB to implement the Globus adapter:

- *Light-weight:*
  The ORB should be light in size and performance in order to influence the observed system as little as possible.
- *Free availability:*
  The chosen ORB should be freely available without any constraints, because the Globus software is also freely available.
- *Functionality:*
  The ORB has to provide a C mapping and basic naming and event services.
- *CORBA compliance*:
  Many ORBs still show a lack of CORBA compliance. This results in a lacking interoperability with other ORBs provided by different vendors. As MIMO is implemented using the ORBacus ORB [Obj2000], CORBA compliance and interoperability are crucial criteria for the selected ORB.

The number of ORBs fulfilling the above requirements is rather small because C mappings are not provided at all with most ORB implementations. After comparing existing ORBs fulfilling our requirements the ORBit ORB [Red1999] was chosen. ORBit is being developed as a GNOME project [GNO2000], provides a C-mapping, and is a very light-weight and high-performing implementation. Also, CORBA compliance and interoperability with ORBacus are given.

7.2.1.4 Globus Adapter Implementation

The Globus adapter is implemented as a C library providing basic functions for attaching to and detaching from MIMO, and for providing event-based asynchronous

data in case of state changes. The instrumentation of the C code itself can be done in various ways, e.g. source code instrumentation or transparent instrumentation of other libraries.

Figure 7.2 shows the functions that are available with the Globus adapter for MIMO. The attach and detach operations are used for setting up and terminating

```
/* attach to and detach from MIMO: */

int  MIMO_attach(int argc, char *argv[]);
int  MIMO_detach(void);

/* generic events: */
int  MIMO_event(char *evtype, CORBA_any descr);

/* object lifecycle and interaction events: */
int  MIMO_newEvent(char *entities[6] );
int  MIMO_delEvent(int layer,char *entname);
int  MIMO_interactionEvent(int layer1, char *entname1,
                           int layer2, char* entname2,
                           char *method);
```

Figure 7.2 Globus Adapter Functions

connections to MIMO. Whenever no MIMO is running within the environment, any function call to the adapter will fail without further influence to the observed program. The parameters to be passed with the above attach and detach operations are defined in MIMO's instrumentation-monitor interface.

The generic MIMO_event operation can be used to generate any kind of event for MIMO. Its format contains a string giving the type of the event and a CORBA any keeping the description. The time of the event is automatically added by the Globus adapter. For often occurring events like lifecycle events (construction and destruction of objects) and object interaction, additional operations have been added to the Globus adapter. The operations are MIMO_newEvent, MIMO_delEvent, and MIMO_interactionEvent. The parameters passed to them comply with the IDL description of possible MIMO instrumentation events, which has been shown in section 6.3.4.

7.2.1.5 Instrumentation of Globus Components

An important concept of MIMO is not to depend on any special kind of instrumentation. Therefore various instrumentation techniques can be applied, as long as communication with MIMO is handled through its IDL interfaces and event channels.

Source Code Instrumentation. Within Globus, the gatekeeper and job manager components have been instrumented by inserting code into the Globus sources. The following things need to be carried out in order to apply this kind of instrumentation to the Globus components:

- Include the "globus-adap.h" header file.
- Add MIMO_attach and MIMO_detach operations at the beginning and at the end.
- Add MIMO event operations wherever events need to be generated.
- Link the Globus adapter library with the executable.

Obviously, the precondition for this approach is the ability to modify the sources and rebuild the application. The advantage is that events can be generated with a very fine granularity, so that the programmer may concentrate on specific aspects that have to be monitored. Moreover, the basic changes to the sources are kept very small as only three lines of code have to be added.

Gatekeeper and Job Manager Instrumentation. In order to apply the techniques described above, the Globus gatekeeper and job manager have been instrumented. By observing the gatekeeper, authentication and authorisation behaviour within Globus can be monitored, because the gatekeeper serves as the "entrance point" for users to a machine; this is the reason why it is classified in the interface layer of the multi-layer monitoring model. Instrumenting the job manager then allows to monitor running Globus jobs and to observe their queueing behaviour. Thus, instrumentation of both components with the approach described above enables the generation of events for MIMO whenever new job submissions arrive at the observed nodes.

Of course, the instrumentation technique used here could be improved in order not to require source code manipulation, e.g. by means of instrumenting libraries used by the gatekeeper and job manager. In this case, no recompilation and no relinking would be necessary, but the event analysis would get more complicated.

7.2.1.6 Summary

In this section we have shown how to connect job submission components of the Globus metacomputing infrastructure to the MIMO infrastructure. For this purpose, an adapter for the CORBA C language mapping has been developed and successfully deployed with the Globus gatekeeper and job manager components. This proof of concept demonstrates the applicability and flexibility of MIMO's multi-layer monitoring approach. Moreover, the developed C adapter for MIMO can be used for other C-based middleware platforms.

To extend the Globus adapter, other Globus information sources can be included to provide information for MIMO, for example the Globus heartbeat monitor [SFK+1998] and Globus MDS [FFK+1997]. Consequently, MIMO then serves as a central point for collecting different information coming from different sources, which leads towards a more general approach for discovering information or event sources within distributed computing environments.

7.2.2 LSD — Latency Sensitive Distribution

In this section we present the integration of the LSD (Latency Sensitive Distribution) metacomputing middleware into the MIMO environment [DR2000].

7.2.2.1 LSD Overview

LSD is a client/server based middleware that relies on the Java thread model. A LSD server is implemented by a master daemon, which represents a task that accepts all kinds of Java threads implementing a specific worker interface. Its interface defines the interaction between the runtime system of LSD and the application's worker tasks. Additionally, the LSD server stores a dynamically generated and periodically updated latency table that contains the communication bandwidth and latency information from every client node in the system. When requested by the master daemon, all registered clients are expected to update their communication tables and send the results to the master daemon. Thus, when receiving several tasks from an application that have to be distributed, the master daemon reads the job communication behaviour from a given structure defined in the worker interface. After the job is distributed to several nodes according to the latency table, a light-weight name service is used for the job-to-job communication.

The default algorithm for thread distribution is straightforward: Low-latency nodes are mapped to threads with the highest communication requirements. In order to improve the distribution of worker tasks in addition to this default distribution algorithm, it is possible to dynamically load and execute new distribution strategies from the connecting application. With this approach the applications are able to install their own distribution mechanism, which makes use of the master daemon's methods for gaining information about the clients.

On the client side, a node daemon handles the jobs sent to this node and cares for all administrative actions to be carried out. For returning the results and for the job-to-job communication, the name server provides the location of the remaining threads of the application. The latency table is also sent to the master daemon by the node daemon.

7.2.2.2 Monitoring LSD with MIMO

In this section, we will show how to monitor the LSD metacomputing platform with MIMO. Doing so, we will follow MIMO's tool development and middleware integration methodology: The tool functions and the derived monitoring data are defined before LSD is mapped to the multi-layer monitoring model. Subsequently, the defined request and event types are implemented on the instrumentation and tool side.

Monitoring Goals. Like with Globus, our goal for monitoring the LSD system is to observe the distribution of jobs within the system. To make this possible, the interaction between users and the LSD master daemon on the one hand, as well as the interaction between master and node daemons on the other hand needs to be observed. Such activities therefore have to be propagated to MIMO by means of appropriate events.

Mapping of LSD to MIMO. As we have chosen to take an application-oriented point of view to monitor LSD applications, our main entities are *jobs* and *threads* distributed by LSD to solve a given problem coming from an LSD-based application. Hence, the mapping of LSD-entities to the MIMO MLM model is as illustrated in Table 7.3.

MIMO MLM Layer	*LSD Entities*
Application	LSD Application
Interface	LSD Worker Interface
Distributed Object	LSD Jobs
Implementation	LSD Thread
Runtime	LSD Process
Hardware	Node

Table 7.3 Mapping of LSD to the MIMO Model

Here, applications making use of the LSD meta-computer can be observed by monitoring the job submissions distributed by the LSD master through the worker interface, and the resulting jobs and their corresponding threads that are executed in processes on specific hosts.

Implementation. The next step is to define appropriate events of interest. For our job distribution scenario, the following events are of interest:

- Creation and deletion of new jobs and threads.
- Interaction between jobs and threads.

This allows to monitor the dynamic behaviour of LSD applications, covering the distribution of jobs, their interactions, and the collection of results.

The instrumentation of LSD applications is implemented using the generic Java adapter described in section 6.4.3. This adapter contains methods for conveniently handling the communication with MIMO on a Java basis. It includes standard functions for attaching to and detaching from MIMO, and allows to send the above events with their parameters to MIMO. As we are only observing applications without manipulating them, this instrumentation can rapidly be inserted into the LSD implementation.

As a result, the LSD runtime behaviour can be visualised with the MIVIS tool without further effort. If additional functionality – for example for steering a running application – is needed, additional displays carrying out these functions can be inserted into the MIVIS framework. Moreover, it is possible to simultaneously monitor Globus and LSD applications whenever running within the same MIMO environment. These cross-platform observation and management capabilities are a major benefit of the MIMO approach.

7.2.3 SEEDS

Having demonstrated the integration of two metacomputing systems into the MIMO system, this section describes a very different distributed environment to be monitored with MIMO. The SEEDS simulation environment is particularly interesting because it is based on two different middleware platforms that are deployed simultaneously within the simulator. One of the main benefits of MIMO is the capability to monitor different platforms at the same time, which results in an improved comparability of data coming from different middleware platforms, but belonging to the same application.

7.2.3.1 SEEDS Overview

SEEDS (Simulation Environment for the Evaluation of Distributed traffic control Systems) represents a distributed interactive real-time simulation environment [Ale1999] [BdSLR1997] composed of powerful workstations connected by a local area network. It is targeted at the evaluation of Advanced Surface Movement Guidance and Control Systems (A-SMGCS), e.g. used for airport ground-traffic management systems. The simulation environment allows the definition and evaluation of technologies and performance requirements needed to implement new functions and procedures of A-SMGCS, to mould new roles within airports, and to introduce new automatic tools and interfaces to support A-SMGCS operators.

The simulation environment is a distributed interactive real-time system, in which several actuators take part in the simulation by practising various roles. These roles include airport surface traffic controllers, traffic planners, pilots and vehicle drivers, and other external actuators like airport authorities. In general, actuators can either be human operators interactively participating at the simulation (e.g. for training purposes), or automated actuators, being simulated by software processes acting according to predefined rules. Human operators (controllers, pilots) are equipped with a 3D visualisation of their current scenario and a 2D visualisation generated from different kinds of radar systems. They act in real-time by issuing commands via specifically designed human computer interfaces. The simulation environment is responsible for managing and distributing the state of the airport (consisting of its static and mobile objects, and other airport features such as control lights or stop bars), as well as for enabling the interaction between different actuators.

7.2.3.2 Monitoring SEEDS with MIMO

The main actuators within SEEDS are interactive controllers supplied with 2D and 3D visualisations of the simulated world, where they can manoeuvre their vehicles. The interesting aspect of building tools for observing or steering the SEEDS system is that it is based on two middleware platforms used simultaneously: CORBA communication is used for the transmission of simulation control data and for 2D visualisation, and the DIS [Gol1995] (Distributed Interactive Simulation) protocol is used for efficiently distributing 3D visualisation data [RdSH$^+$1999a], [RdSH$^+$1999b], [RdSH$^+$2000].

Mapping of SEEDS to MIMO. Therefore, for monitoring a running SEEDS simulation, entities of both platforms have to be considered. Table 7.4 illustrates the resulting mapping of SEEDS to the MLM model, when 2D and 3D visualisation aspects are of particular interest; it can be seen that for the interfaces and distributed objects, both CORBA and DIS entities have to be considered.

MIMO MLM layer	*SEEDS layer*
Application	*SEEDS simulation run*
Interface	*2D CORBA and 3D DIS visualisation interfaces*
Distributed Objects	*2D CORBA and 3D DIS clients and servers*
Implementation	*Programming language objects*
Runtime	*OS Processes*
Hardware	*Node*

Table 7.4 Mapping of SEEDS components to the MIMO model

Implementation Aspects. The implementation of the instrumentation code for the SEEDS components is similar to the instrumentation of the metacomputing platforms. As the SEEDS simulator is implemented in C++ and Java, the C and Java adapters can easily be reused here. Events need to be generated for the various aspects of the 2D and 3D visualisation, which is a performance critical part of the system that has to be optimised carefully [RL1997].

7.2.4 Managing a Load-Balanced Medical Application

Another interesting application scenario for MIMO is a medical application in the field of distributed image processing. Here we present an example for visualising and steering a realignment application used for the reconstruction of images obtained through functional magnetic resonance imaging (fMRI) techniques [May2000].

7.2.4.1 The Load-Balanced Realignment Application

Data obtained from fRMI need to be processed to reconstruct the images from the measured projection data. An important task within this procedure is the realignment of images gained from the fRMI, with which a set of images is aligned to a predefined reference image.

As the reconstruction effort for such images is very high, distributed reconstruction using clusters of workstations is carried out. The architecture of our exemplary realignment application is based on the client/server paradigm. Clients submit realignment jobs to the load-balanced realignment server, which in turn distributes the jobs to appropriate servant objects according to its load distribution strategy. The servants carry out the realignment and pass back the result to the respective client. The advantage of this proceeding is that depending on the amount of available resources a variable number of servants can contribute to the reconstruction process.

The distributed realignment application is based on CORBA as a communication middleware and implemented both using Java and C++.

Furthermore, in order to utilise the dynamically changing resources in an advantageous way, a load balancing mechanism is applied to the realignment application [Lin2000]. The entities of interest for the load balancer are the servants carrying out the computation tasks. As these servants are implemented using CORBA objects, *object migration* and *object replication* represent the load balancing mechanisms applied for the realignment application. The implementation of the load balancing mechanism makes use of a modified ORB that has been extended for load balancing purposes.

7.2.4.2 Monitoring and Managing the Realignment Application with MIMO

Figure 7.3 shows an overview of the load-balanced realignment application and the monitoring scenario. Jobs coming from the clients are distributed to the servants, which retrieve original images and pass back the realignment data. The load balancer controls the servant objects and carries out object migrations or replications according to its load distribution strategy.

In our monitoring scenario, both the realignment application and the load balancer are instrumented and controlled by MIMO. The realignment application uses the Java adapter to provide data about the existing client and servant objects for MIMO, so that a visualisation of the overall realignment process is made possible. Additionally, the load balancer is instrumented and provides information about load balancing decisions to MIMO, such that load balancing decisions can also be traced. The resulting visualisation is implemented with a specialised MIVIS display that shows the available computing nodes, the servants running on these nodes, and additional load information.

Beyond these visualisation capabilities, our scenario also allows to manipulate the running realignment application by steering the load balancer from the MIVIS display. Object migrations and replications can be initiated with a drag-and-drop approach that allows to move objects between the available computing nodes in the MIVIS display.

Interoperability between MIVIS and Load Balancer. The realignment scenario implements a kind of interoperability between the load balancer tool and MIVIS. However, the load balancer directly operates on the realignment application without making use of MIMO, which complicates the interaction between both tools. For keeping the load balancer under the control of MIMO, it is therefore instrumented to provide information about its load balancing operations as well as to receive explicit commands from MIMO. Hence, MIVIS represents the leading tool in our case: It visualises information from the realignment application and the load balancer, while it can manipulate the running application by issuing commands to the load balancer. This kind of interoperability is an advantageous approach for our scenario, where the automatic load balancer independently performs its tasks while it can be observed and controlled by the dominating MIVIS tool in order to carry out manual interventions.

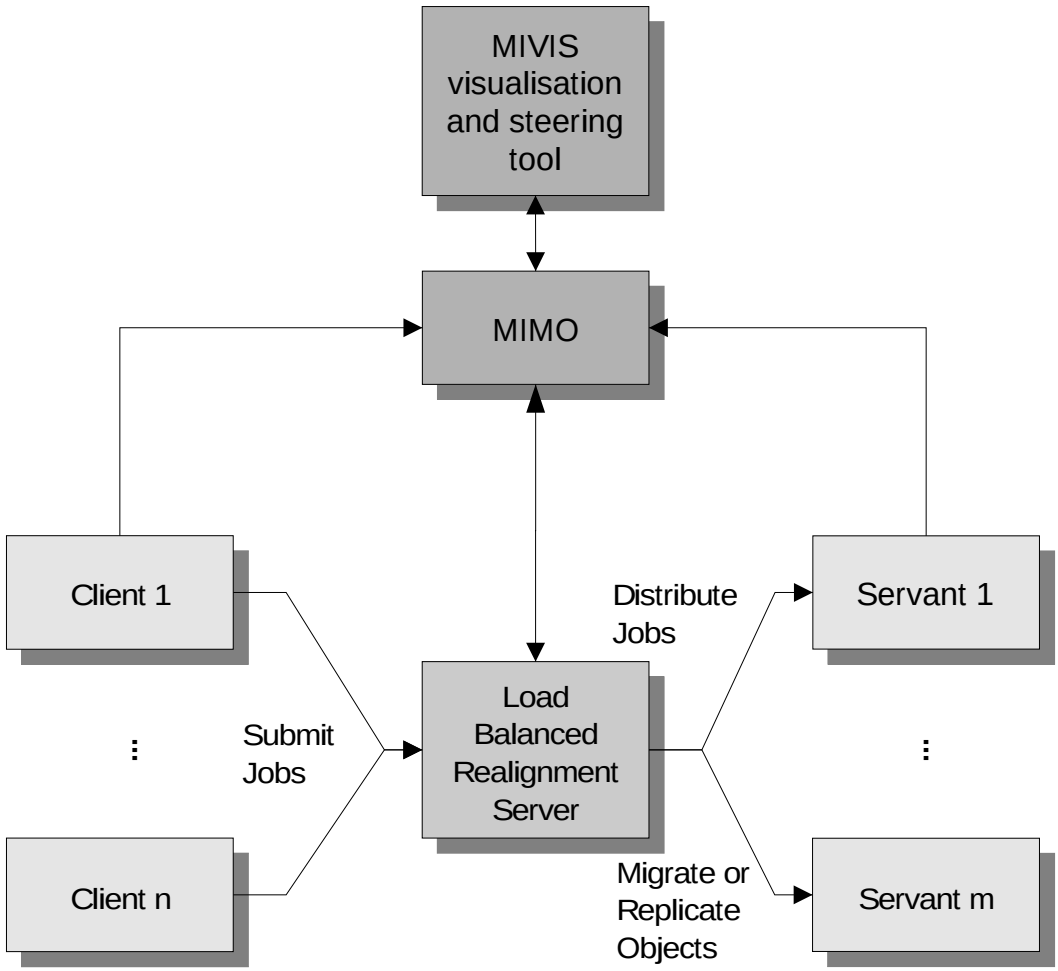

Figure 7.3 Overview of the Realignment Scenario

Application Steering and Management. Our realignment scenario represents an example for steering monitored applications by means of a GUI-based MIVIS display. The migration or replication of objects can be initiated for manually controlling running applications, which can be profitable for several purposes:

- Application steering can be carried out to tune the performance of the realignment application.
- Application or system management can be carried out by moving objects away from computing nodes, e.g. when maintenance tasks need to be executed on those machines. This manipulation is implemented without interfering with the running realignment process.

Depending on the respective usage scenario, other steering or management tasks are imaginable in addition to our object migration and replication examples. Nevertheless, the ability to manipulate running applications and the possibility for tools to interoperate with each other represent an essential foundation for further tasks.

7.2.5 Summary

Summing up, we have presented a set of rather diverse application scenarios for the MIMO infrastructure. The Globus and LSD metacomputing platforms have been integrated as well as the SEEDS simulation environment. Instrumentation mechanisms for these platforms can be very diverse and concentrate on certain aspects that have to be observed. For example, with Globus we focused on the job submission procedure, but due to the complexity of the Globus environment many other processes could also be monitored, if necessary. In any case, MIMO provides the basis for supervising further aspects or for adding further information sources.

Moreover, our realignment application from the medical image processing field has shown how to integrate visualisation capabilities with steering or management tasks. The interoperability between the load balancer and the MIVIS tool is a powerful example that allows to manipulate running applications for computational steering or management purposes.

Concerning the tool implementation, we have seen that basic visualisation capabilities are available after the integration of the middleware by using the MIVIS tool. Further functionality can easily be introduced by implementing further displays that can be added to the tool framework.

7.3 Conclusion

This chapter has presented a methodology for advantageously using the MIMO infrastructure, followed by several real-world application scenarios demonstrating the feasibility of the described approaches. It can therefore be seen as a proof of concept for the MIMO idea and its implementation.

The theoretical methodology derived in the first section has shown to be applicable to the set of real-world applications demonstrated in the second section. Moreover, possibilities for further implementation techniques or extensions have been illustrated.

These considerations allow us to conclude the description of the MIMO system. In the following chapter, we will proceed with a critical evaluation of the overall approach and its consequences before we point out the major results achieved during the development of MIMO.

Chapter 8

Evaluation and Results

Critically reconsidering and validating the MIMO approach and its implementation is an important aspect of this thesis. This chapter will start with an evaluation of important MIMO topics, followed by an elaboration of the results of the MIMO project.

The evaluation of the work described in this thesis concentrates on three topics. First of all, the general approach consisting of the multi-layer monitoring model and its implications will be discussed. Secondly, our implementation of the MIMO approach that resulted in a MIMO prototype will be reconsidered with regard to general implementation issues as well as performance criteria. And thirdly, we will review MIMO's tool development methodology both on a theoretical basis and by means of the described application scenarios.

Concerning the results of this thesis, we will point out the insights gained during the conception and construction of MIMO. As an outcome, we will present significant characteristics of monitoring and management within heterogeneous middleware environments. Besides being able to master the heterogeneity within such systems, the capability to deal with dynamic environments is a major goal that needs to be accomplished. These insights substantially influence design and implementation decisions of future monitoring systems.

8.1 Evaluation

The evaluation of our work considers the three topics general approach, implementation and performance, and usage of the monitoring and management system. Important criteria for our considerations are the monitoring requirements stated in chapter 4 as well as further aspects that came up during the development.

8.1.1 MIMO Approach

The MIMO approach described in chapter 5 rests upon three fundamental blocks: The multi-layer monitoring model, a generic monitoring infrastructure, and a usage methodology and tool framework. Consequently, we will critically reconsider each of these blocks in order to get an evaluation of the overall MIMO approach.

8.1.1.1 Multi-Layer Monitoring Model

Structuring the information being monitored in complex distributed environments is an essential aspect of a monitoring system. For this purpose, MIMO uses the multi-layer monitoring model as an information model for the representation of the observed middleware environment.

The advantage of the multi-layer monitoring model is that it allows a classification of monitored entities into several hierarchy levels. As a result, information on all abstraction layers can be gathered and stored within the system state graph, while relationships between adjacent layers imply further semantics. Especially for large, distributed environments such a classification makes it easier to gain an overview of the observed systems and applications, so that the requirement for supporting complex systems is fulfilled. Another benefit of the multi-layer monitoring model is that it enables the comparability of information coming from different information sources or middleware platforms. Entities classified on the same layer of the model are comparable in a defined way that is determined by the mapping of the respective middleware to the multi-layer monitoring model. The impact of this comparability aspect can mainly be perceived when heterogeneous systems are being monitored with MIMO. Entities residing on the same abstraction level can advantageously be compared to each other, no matter from which middleware platform they are originating. Therefore, the multi-layer monitoring approach optimally fulfils the requirement for supporting heterogeneous computing environments.

Considering the drawbacks of the multi-layer monitoring model, a few points have to be discussed. As mentioned before, the six layers defined by our model are not appropriate for every existing middleware platform. The need to treat empty layers differently is an indication for this fact and allows to create a smart workaround for affected middleware platforms. However, there are only two possibilities to avoid completely the case of platforms not fitting into the multi-layer monitoring model:

- The first possibility is to abandon the idea of using an information model at all. The consequence would be a large set of entities that are not structured with respect to any criteria. With this approach all the advantages of MIMO would get lost, so that this approach results to be non-feasible.

- The second possibility is to make the multi-layer monitoring model customisable with respect to the layers existing within the information model. The result would be a more complicated information model because each new middleware could enforce to create new layers within the model, making the usage of MIMO essentially more complicated. Moreover, one of the main benefits of the model would also get lost because the comparability between entities coming from different platforms would no longer be given.

Hence, when we consider all these arguments, the multi-layer monitoring model is justified and represents a main benefit of the MIMO system. It can be seen as a kind of invariance that is used to merge all the heterogeneous middleware platforms. It tries to be as general as possible, while it is still tied to a notion of middleware

that cannot comprise all imaginable environments perfectly. In this sense, the multi-layer monitoring model represents a trade-off between generality on the one hand, and conformance with common middleware systems on the other hand, such that a maximum level of integration is reached.

8.1.1.2 Generic Monitoring Infrastructure

The generic monitoring infrastructure proposed with the MIMO approach fulfils all requirements stated before. It provides an infrastructure that takes over all main tasks of a monitor that are needed by all kinds of tools. This comprises start-up procedures for the attachment to the monitor within dynamic environments as well as basic functions for obtaining an overview of the observed system, its main actuators and their interactions. Furthermore, the interfaces for tools and instrumented components are kept generic in order to allow the implementation of arbitrary tool functionality.

Hence, the monitoring infrastructure fulfils the requirement for supporting all on-line phases of the software lifecycle, as tools for all these phases can be implemented on the basis of the MIMO infrastructure. Furthermore, complex and heterogeneous systems are supported as the infrastructure relies on the multi-layer monitoring model. Flexibility and extensibility are given due to the generic interfaces of the monitoring infrastructure. The development of new tools as well as the integration of new middleware can be carried out by using MIMO's standard interfaces and the customisable event exchange mechanisms.

A point of criticism of the infrastructure is that it is complex, which results in a relatively high effort for developing tools or integrating middleware. However, there are two arguments that object to this statement:

- Firstly, the usage methodology and tool framework delivered with the infrastructure simplify the process of tool development and middleware integration as far as possible.

- Secondly, a monitoring approach leaving many degrees of freedom to the user is in principle more complex than a system with a fixed functionality. But, providing a monolithic system with a fixed functionality that is as powerful as our infrastructure is impossible because not all application scenarios can be foreseen at the time of the development of the monitor.

As a consequence, developing tools with the MIMO infrastructure causes less effort because they can profit from the flexible infrastructure. With static and monolithic monitoring systems the monitor itself needs to be extended for the respective scenario in each case, which causes more effort than customising MIMO for a new scenario.

Therefore, the idea behind MIMO's monitoring infrastructure can be seen as an approach for intelligently enabling the communication between tools and instrumented applications, where the monitor core only provides basic functionality and propagates more advanced information between the involved components. The result of this approach is a light-weight and powerful monitoring infrastructure.

8.1.1.3 Usage Methodology

From the conceptual point of view, MIMO's tool development methodology optimally fits into the monitoring infrastructure. As the infrastructure leaves a lot of freedom to the user, the development methodology serves as a guidance for users developing new tools or integrating new middleware. By giving a standard procedure to the user, the complexity of development tasks can therefore be reduced, while for special purposes it is still possible to deviate from the methodology in order to make use of the full capabilities of MIMO.

Another advantage of adding a usage methodology to the MIMO system is that it enables a rapid tool development process. In analogy to the rapid prototyping concept from the software engineering area, the incremental and iterative development of tools and instrumentation components allows to implement such components rapidly, while the possibility of further extension is given.

All in all, MIMO's usage methodology emphasises the requirement for being able to implement new tools or to adjust to new platforms rapidly and efficiently . This aspect is of great importance, especially for current and coming middleware systems where monitors will have to deal with even more dynamic and evolving applications.

8.1.1.4 Conclusion

The overall MIMO approach consists of a separation into the three basic blocks multi-layer monitoring model, monitoring infrastructure, and usage methodology. This design of the monitoring approach has shown to be clear and applicable from a theoretical as well as from a practical point of view. The separation of the multi-layer monitoring model from a concrete implementation of the infrastructure decouples the underlying information model from technical aspects, and the definition of a usage methodology simplifies the application of the complex monitoring system. Thus, our monitoring approach represents a systematic and structured concept for observing heterogeneous middleware environments.

8.1.2 Implementation and Performance

To evaluate the implementation and performance of our MIMO prototype, we consider several techniques with their advantages and disadvantages. For different implementation techniques, the design and implementation effort can be deducted with respect to different programming languages, communication platforms, and software development tools. Subsequently, further criteria concerning the runtime performance, reliability, and portability of the implemented system can be evaluated.

For our evaluation, we will start with the MIMO core, which represents the heart of the monitoring system. Next, we will consider the instrumentation and tool components that make use of the MIMO core.

8.1.2.1 MIMO Core

As extensively described in section 6.4, the MIMO core is implemented using Java as a programming language and CORBA as a standard for communication among the distributed components. When we evaluate the MIMO core prototype according to the criteria stated above, the design and implementation efforts of Java and CORBA result to be low, yielding an efficient design and implementation of the MIMO core.

Designing the interfaces with CORBA can be carried out in a clear and structured manner using CORBA's interface definition language. The implementation of the core with Java is a straightforward process due to the advanced integration of Java and CORBA. As Java resides on a relatively high abstraction level that is supported by elaborate class libraries, the implementation of the MIMO core can be carried out with a low effort compared to other lower-level programming languages.

The resulting implementation can then be evaluated with respect to runtime performance, reliability, and portability. As we have seen, the performance of our prototype is sufficient for most middleware applications. Both latency and throughput of the communication between instrumentation, tools and the MIMO core do not delay application tasks significantly. The performance critical part is the instrumentation code that can increase the overhead of the observed application substantially in some cases. Moreover, the memory footprint of the Java virtual machine executing the MIMO core is relatively high; thus, when memory limitations are given on observed machines, they should possibly not execute an instance of MIMO core directly, but communicate with a MIMO instance on another machine.

A further advantage of our implementation using Java and CORBA is its reliability. Due to the advanced exception mechanisms for dealing with failures it is possible to detect and recover from error cases with little effort. Finally, the portability of an implementation based on Java and CORBA is outstanding. Both Java and CORBA implementations are available for nearly any hardware and operating system, allowing to transfer the MIMO core to new platforms easily.

As an alternative to our implementation approach, languages and communication paradigms with a lower memory usage and a higher communication performance could be taken into account. For example, the core could be implemented using C as a programming language and make use of PVM as a communication platform. This proceeding would reduce the communication latency and the memory consumption to some extent. However, considering the remaining evaluation criteria still makes our approach preferable to this solution: The design and implementation effort substantially increases with C and PVM, reaching a comparable reliability causes much more programming effort, and the portability is problematic in principle.

Thus, an overall consideration of the MIMO core yields the insight that our MIMO core using Java and CORBA is a well-suited approach for implementing the monitoring concepts. While the performance could be increased slightly with other implementation techniques, Java and CORBA are preferable with respect to all remaining evaluation aspects, making it the favourable solution for our middleware monitoring purposes.

8.1.2.2 Instrumentation

Regarding the instrumentation of applications, we have seen that the possibilities are manifold due to MIMO's concept of separating the monitor core from instrumentation components. As an example, we have presented several scenarios for integrating different middleware platforms. For all these intruders or adapters, the design effort is low as the interface to the monitor core is defined exactly, and the development methodology describes the major steps for implementing new instrumentation components. Of course, depending on the respective platform that needs to be instrumented, the effort of gathering relevant information or manipulating the application can be relatively high, but the overall procedure for interacting with the MIMO core is well defined.

For the remaining criteria, an evaluation has to distinguish carefully between the particular middleware that is instrumented. While a Java adapter causes little overhead due to the fact that it makes use of the virtual machine already executing the instrumented Java application, the instrumentation of the DCOM applications causes a high overhead if no sophisticated COM to CORBA bridge can be used. Therefore, the performance of the instrumentation components needs to be regarded in relation to the observed environment. In any case, if the overhead of the instrumentation is too high, i.e. if the application is influenced too much by integrating the CORBA-based instrumentation code, it can be decoupled from the monitor by using more efficient communication mechanism. For example, for performance critical applications the instrumentation component can act as a kind of proxy object for the monitor core, which communicates with the application using shared-memory techniques and converts the data to CORBA calls for interacting with MIMO. As shared-memory communication can be implemented very efficiently, the overhead of the instrumentation can be decreased substantially with this approach.

Finally, reliability and portability aspects have to be analysed. Of course, reliability is given from the point where CORBA communication is used for further interaction. Concerning portability, the presented instrumentation components can be reused for similar scenarios where the same middleware needs to be observed. Portability regarding the transfer of code to different hardware or operating systems is not an issue in this context because instrumentation components are tied to specific middleware, which is mostly bound to predefined environments.

8.1.2.3 Tools

The final aspect for evaluating our MIMO implementation is the MIVIS framework for developing tools. MIVIS is implemented in Java, relying on the Java Swing [ELW1998] package for building graphical user interfaces and the Java Beans concept [Ham1997] for adding new components. The design and implementation of GUI tools is made very simple as the framework and its existing displays can be deployed. The main benefit of this proceeding is that it reaches a very high degree of portability, which is very important for tools as they should be able to run on any machine within a heterogeneous environment. Moreover, the extensibility that is accomplished by using the Java Beans concept is an important advantage.

However, MIVIS is only applicable for building GUI-based tools. For tools that require no graphical user interface, e.g. automatic tools like load balancers, the framework is hardly suited as such tools should operate in the background. Nevertheless, as building graphical user interfaces causes a high effort within the tool development process, our contribution to simplify this task can reduce the overall tool development effort in many cases.

8.1.2.4 Conclusion

In this section, we have evaluated our prototypical MIMO implementation with respect to several criteria. We have shown that our approach making use of Java and CORBA as an implementation platform for the monitor core is the preferable solution regarding the design and implementation effort, the reliability, and the portability. The performance of the system is sufficient for most monitoring scenarios, while it can be reduced substantially by selecting intelligent instrumentation mechanisms, if necessary. The MIVIS tool framework has shown to be a very efficient way for constructing GUI-based tools.

Altogether, our implementation emphasises the integrative aspect of the monitoring concept, which is open to both the tool- and instrumentation-side. Also, for future systems, Java and CORBA represent a very useful choice for our purposes, as e.g. mobile devices will be able to communicate using *minimumCORBA* [OMG1998a], allowing MIMO to be used within these highly dynamic domains.

8.1.3 Tool development and Middleware Integration

The third important concept for our evaluation is MIMO's tool development and middleware integration methodology. As we have already seen in the evaluation of the MIMO approach, the tool development methodology provides a systematic way of using several degrees of freedom of the monitoring infrastructure. From a theoretical point of view, the MIMO core serves as an instrument to enable the communication between tool and instrumented applications. Consequently, tools and instrumentation components can be optimally adapted to each other by making use of the methodology. From the practical point of view, we have demonstrated the applicability of our methodology by means of several applications scenarios. The tool development and middleware integration process has shown to be a straightforward and efficient process in all of the considered example scenarios. Above all, the capability to incrementally augment instrumentation and tool functionality is a major benefit that supports the software development lifecycle.

Of course, when we critically analyse our methodology it becomes clear that not all imaginable application scenarios can be covered by the five steps making up the development lifecycle. Nevertheless, it still serves as a basis for dealing with most standard situations that users can adjust to their special needs, if required. Furthermore, the steps of the methodology are kept very general, which makes the customisation for particular scenarios easy.

When we consider the set of available monitoring systems in the field of middleware, MIMO is the only approach that explicitly provides a usage methodology.

The reason for this lies in the complex and powerful architecture of the monitoring approach. While other monolithic monitors do not need a tool development or middleware integration methodology because of their static and limited scope of usage, MIMO is laid out much more generally. The price for this generic approach that allows MIMO to be used in a large scope of scenarios is a more complex usage, which has to be supported by a sophisticated methodology. Hence, our methodology reflects these considerations and optimally assists users of the MIMO system.

8.2 Results

After having evaluated the key features of the MIMO system, we can now summarise the results acquired during the conception, development, and application of MIMO.

8.2.1 Problem Review

The initial motivation for our work was to find a way for efficiently constructing on-line tools for heterogeneous middleware environments. Based on an analysis of existing systems for current middleware platforms, we have postulated a list of requirements in order to eliminate the drawbacks of tools and monitors. The stated requirements comprise the following aspects:

- The need for a systematic monitoring concept for complex environments.
- The support of all on-line phases of the software lifecycle.
- The coverage of the system heterogeneity.
- The demand for a flexible and extensible tool environment.

The MIMO approach reflects these requirements and defines a monitoring concept that rests upon three basic blocks. Furthermore, during the design, implementation, and evaluation of the MIMO prototype, several additional criteria of great importance for monitoring heterogeneous middleware have been worked out. Therefore, we will here recapitulate the conceptual results derived from our requirements and theoretical considerations, as well as additional lessons learned during the implementation of the MIMO ideas.

8.2.2 Conceptional Results

A principle result derived from the requirements postulated for the construction of a tool environment in heterogeneous middleware platforms is to partition the monitoring system into three basic blocks. An information model is used to represent all kinds of entities within observed middleware applications. Based on this information model, a generic monitoring infrastructure enabling the interaction between tools and observed applications is defined and implemented. As this infrastructure is kept open and leaves a high degree of freedom to users, a methodology for developing new tools and integrating new middleware into the monitoring environment is established.

8.2.2.1 Abstract Information Model

The multi-layer monitoring model represents a hierarchical information model that is described by an abstract entity-relationship model. The abstraction layers of the model are defined in order to be able to cover almost all existing middleware platforms. A formal description of the multi-layer monitoring model allows to derive algorithms for efficiently accessing and manipulating the resulting data structures. Furthermore, the application to concrete middleware platforms is made possible by a well-defined procedure for mapping given middleware platforms to the abstract layers of the multi-layer monitoring model. Considering other monitoring approaches for middleware-based systems, no such system provides an information model with this degree of universality.

8.2.2.2 Generic Monitoring Infrastructure

Another essential result of the MIMO approach is the generic monitoring infrastructure. The insight that no static or monolithic monitoring system is able to fulfil the requirements of the middleware systems under consideration leads to the definition of the generic monitoring infrastructure.

However, our infrastructure goes further than the traditional separation of tools, the monitor, and the observed application presented in section 3.1.2, where the monitor is not monolithic anymore but still static with respect to its functionality. Instead, MIMO's infrastructure is generic with regard to both the monitoring architecture and its functionality. This property is achieved by keeping all interfaces to the infrastructure generic, i.e. putting as little semantics as possible into the monitor core. In order to be open to all kinds of middleware and tool requirements, no predefined application scenario is assumed.

The syntax of all interfaces and event descriptions is defined exactly, so that tools and instrumentation components can be integrated in a straightforward manner. Depending on the semantics of the information exchanged between tools and instrumentation, any kind of functionality can be implemented without being limited by a static set of operations within the monitor core. Moreover, due to its generic design, the complete infrastructure can be kept very light-weight.

8.2.2.3 Tool Development and Middleware Integration Methodology

The third basic block of the MIMO approach is the tool development and middleware integration methodology. As MIMO's monitoring infrastructure is layed out in a generic way, the need for a usage methodology arises. Fulfilling this need results in a methodology that describes a standard procedure for building new tools with MIMO or for integrating new middleware platforms. Therefore, it can be seen as a guidance for users in order to make use of the MIMO system in an efficient and rapid way. Furthermore, the comparability of data coming from different sources and the reusage of instrumentation components is improved by following our methodology. In analogy to the rapid prototyping approach within the software development process, MIMO's tool development methodology establishes a rapid tool development process. Also, while tools can be constructed rapidly, their incremental extension is

assured due to the iterative procedure.

8.2.3 Lessons Learned

In addition to the conceptual results, we have gained several important insights concerning our monitoring issue. These results have not intuitively been clear from the beginning of our developments, but represent outcomes of the incremental development process and its consequent review.

8.2.3.1 Monitor as an Event-Propagation Infrastructure

The first aspect concerns the usage possibilities of the generic infrastructure. Due to the openness of MIMO's communication facilities, it can be regarded as a generic event-propagation system that is not necessarily tied to pure monitoring purposes. As said, the infrastructure is light-weight and assumes only little semantics about the information that is exchanged. Standard services like looking up components existing within the system and interacting with these components are useful for any kind of interaction in distributed applications.

Hence, MIMO itself can be seen as a kind of middleware for event-based distributed systems. Through its generic design, not only the on-line tool domain, but also other application domains may profit of the MIMO infrastructure. Relevant domains are those requiring similar features as the monitoring system, including support for heterogeneity or dynamic integration of new components. Especially in the field of mobile devices, these requirements will increasingly be given in future systems.

8.2.3.2 Dynamic Behaviour of Modern Middleware Systems

Another important insight is to realise the dynamic behaviour of current middleware systems. Dynamic processes within middleware environments occur in several ways:

- First, middleware applications change frequently because components are often added, deleted, or modified. The middleware platforms taken for the implementation of distributed applications are constantly under development as technology moves rapidly in this field. Moreover, new applications and middleware platforms need to be dynamically integrated into existing operating environments.

- Secondly, the behaviour of the observed applications is highly dynamic. Complete applications or single application components are only temporarily active and may change their location.

Therefore, this dynamic behaviour of the overall application lifecycle, as well as the dynamic behaviour of running applications has a great impact on the capabilities of the monitoring system. *Organisational aspects* play a very important role for monitoring purposes, including the dynamic look-up procedures for detecting and attaching to the monitoring system and the location transparent communication

between the involved components. MIMO's approach that allows to detect MIMO instances through location transparent name service is an example for such an organisational aspect. Furthermore, a well-defined methodology for dealing with the dynamic application lifecycle is a useful feature in this context.

Altogether, we can conclude that current middleware systems represent a "moving target" that rapidly changes in various ways. For future systems, the dynamics will increase further because a higher number of mobile devices will participate in distributed middleware applications, enforcing the need for dealing with dynamic processes.

8.2.3.3 Integrative Aspects of Monitoring and Management

A last aspect that needs to be emphasised is the integrative aspect of monitoring and management in heterogeneous middleware environments. As the number of environments dedicated to single applications is decreasing in the field of middleware applications, the need to observe multiple applications simultaneously arises. Especially for management purposes, obtaining an overview about the overall operating environment is an important task. Therefore, a monitoring system integrating various heterogeneous middleware platforms and applications as well as tools supporting all aspects of the software lifecycle are a significant contribution with respect to the integrative aspect of monitoring and managing heterogeneous middleware.

8.3 Summary

In this chapter, we have evaluated our work with respect to the general approach, implementation and performance aspects, and the tool development methodology. Our overall approach has shown to be feasible from a theoretical point of view and with respect to the practical application to real-world scenarios.

After a review of the initial problem, we have presented the conceptional results and the lessons learned during the implementation and refinement of the MIMO system. The conceptional results mainly concern the architecture of the monitoring system and its application. The lessons learned emphasise the generality of the monitor implementation, which itself represents a kind of middleware platform. Essential criteria within such environments are the ability to deal with dynamic environments and the integrative aspect of the monitoring approach. These insights will gain even more importance for the construction of future monitoring and management tools.

Chapter 9

Conclusion

This thesis has presented a new approach for monitoring and managing heterogeneous middleware. In this chapter, we will conclude the thesis by summarising the work that has been carried out and the results that we have obtained. Additionally, we will provide an outlook to open questions and future research topics.

9.1 Summary

The motivation behind our work was the insufficient on-line tool support for developing and deploying distributed middleware applications. Based on the analysis of current middleware systems and existing tool environments, we have elaborated the shortcomings and disadvantages regarding the on-line tool support. The shortcomings of existing tool approaches lie in their proprietary solutions that are only applicable to simple and homogeneous scenarios. The architecture of existing monitoring systems is inflexible and makes the integration of new middleware platforms or new tools complicated.

Modern middleware environments in contrast tend to be very diverse and complex. Therefore, an approach for optimally constructing on-line tools for such environments needs to be generic and to fulfil a set of properties that we have extracted with our investigations. A generic monitoring approach for modern middleware environments needs to provide a systematic concept for dealing with complex environments. This comprises mechanisms for integrating heterogeneous components, the support of tools for all phases of the on-line software lifecycle, and a flexible and extensible architecture. On the basis of these requirements, we have derived the MIMO approach for monitoring and managing heterogeneous middleware. MIMO rests upon three fundamental concepts that address our requirements. As extensively illustrated, the *multi-layer monitoring model*, our *generic monitoring infrastructure*, and the *usage methodology* make up the basic blocks of our monitoring approach. For our prototypical implementation of the MIMO approach, we have described the monitoring architecture, the usage of the monitor by tools and instrumentation, and technical aspects. Subsequently, we have demonstrated the applicability of the MIMO system to various application domains. In addition to the observation of general purpose middleware, we have shown the usage for metacomputing systems, a simulation environment, and a medical image processing application.

The evaluation of our work has shown that the MIMO approach is well suited for our monitoring and management purposes. Splitting the overall concept into the

three parts information model, monitoring infrastructure, and usage methodology is advantageous for coping with the manifold requirements given within heterogeneous middleware environments. The multi-layer monitoring model is an appropriate approach that provides a systematic formalism for handling complex and heterogeneous middleware. The implementation of our generic monitoring infrastructure using Java and CORBA has shown to be applicable for our scenarios. Further performance improvements could be achieved by refining the instrumentation policies and implementations. Nevertheless, taking into account the design and implementation effort, no other language or communication paradigm results to be preferable to our implementation based on Java and CORBA. The usage methodology finally represents a beneficial component of the MIMO approach. Due to the generic and open design of the monitoring infrastructure, it serves as a guideline for implementing tools or integrating new middleware and thus enables a rapid tool development process.

When we reconsider our initial motivation and the solutions to the problems we have addressed, the major scientific difficulty is to handle the high degree of heterogeneity within the platforms under consideration. This heterogeneity not only concerns the middleware platforms themselves, but also the different types of tools that users wish to implement, ranging from early development tasks like visualisation to advanced management functions like the administration of application environments. In order to find a solution to this problem, a trade-off between universality on the one hand and adaptation to specific requirements on the other hand had to be found. A high degree of universality makes the system powerful and applicable to a wide range of scenarios, but more complicated to use. In contrast, a system that is tailored to specific requirements simplifies the handling, but restricts the scope of usage. Our trade-off to solve this contradiction has been to design a system that allows to integrate nearly any middleware that possesses a kind of object model, and to leave it open for implementing a wide range of tools. The multi-layer monitoring model and the generic monitoring infrastructure reflect this trade-off.

A flexible integration of middleware platforms is enabled by the mapping strategies of the multi-layer monitoring model, while the monitoring infrastructure provides a light-weight monitor core that assumes little semantics regarding the exchanged data. Nevertheless, in order to achieve a simple and efficient usage of the monitoring system, the monitoring infrastructure supports basic services that are needed by all tool environments. Moreover, our usage methodology provides additional assistance aiming at the efficient usage of the generic system. Hence, our overall approach for dealing with heterogeneity can be summarised by the idea of providing a universal system that is supported by sophisticated services and a usage methodology.

What we learned from our investigations is that the resulting monitoring system represents a kind of intelligent event-propagation infrastructure. It allows tools and instrumentation components to interact with each other in an asynchronous and location transparent way. This kind of communication is not necessarily tied to monitoring purposes, but could easily be transferred to other similar scenarios. In this sense, our monitoring infrastructure itself can be seen as a middleware platform,

which implements the publisher-subscriber communication paradigm, enhanced by additional services that simplify organisational tasks.

Furthermore, we learned that dynamic processes play a substantial role within current middleware applications. On the one hand, this concerns rapid changes regarding the platforms used for distributed applications and the existing applications themselves. As technology moves fast, a monitoring system continuously has to deal with changing platforms and applications. We attack this issue with our advanced usage methodology, which allows an incremental adaption to changing situations and supports a rapid tool development process. On the other hand, dynamic processes occur within existing applications. The behaviour of modern middleware applications is highly dynamic with respect to the existence, activity, and location of application components. This trend towards highly dynamic systems will further increase with the spread of mobile devices. Hence, modern monitoring systems need to be able to handle such dynamic processes. Our approach to solve this issue is to implement a monitor core that takes into account the organisational aspect needed for dynamically connecting tools, monitor, and instrumentation components with each other.

Finally, we have found that integrative aspects are of substantial importance when tool environments for heterogeneous middleware applications are constructed. Especially for management purposes, where obtaining an overview of the complete computing environment is a relevant feature, tools being able to combine information coming from different applications are advantageous. Therefore, the integrative aspect of our monitoring approach is a significant and non-negligible contribution for building on-line tools for heterogeneous middleware environments.

9.2 Outlook

In addition to the work described within this thesis, there are further related questions connected to our monitoring research. First of all, we did not focus on the aspect of interoperability between concurrently applied on-line tools. The only scenario considering the cooperation between tools is the medicine scenario with the combination of the MIVIS visualisation and management tool and a load balancer. Here, more sophisticated concepts for tool interoperability obtained in [Tri1999] could be taken into account.

Another interesting aspect concerns the support of standards for our monitor implementation. For example, gathering information about the observed computing environment could make use of the SNMP standard (see section 3.2.4.3). As implementations of SNMP are widely spread and available for most platforms, the monitoring system could use SNMP as a data source for gaining system related data. These data can be useful for supporting application-specific instrumentations that are not able to access system related data. With the MIMO approach, we have now provided the basis for integrating data sources like SNMP easily.

Moreover, the format for transferring events between tool, monitor, and instrumentations currently relies on the generic data types provided by CORBA. A more general approach could make use of upcoming standards like XML [LLF1998] to

describe the data being delivered with an event. XML allows to pass a description of the event semantics with the event itself in an advantageous way. By developing an appropriate XML document type description, a systematic approach for defining the event semantics could be provided for both tool- and instrumentation-side. With this proceeding, the tool development and middleware integration process could be organised in a more transparent manner.

Finally, a highly critical and sensitive aspect that we did not consider within this thesis is security. Especially for practical usage within enterprise computing environments, strategies prohibiting the access of unauthorised tools to the monitoring system have to be found. In this, mechanisms for controlling the access to the monitoring system on several levels need to be defined. For example, users or tools might be restricted to observe certain applications or aspects of applications, while system administrators are authorised to access all applications. Also, restrictions could be defined on different abstraction levels, such that ordinary users cannot access system critical data, or that the manipulation of running applications is reserved for distinguished users. Integrating security into monitoring systems therefore represents an important research topic for which up to now no sophisticated approaches have been proposed.

When we extrapolate current developments regarding the future of distributed applications and technologies, basic trends influencing the requirements for the next generation of on-line tools can be observed. At first, coming applications and access structures will be even more dynamic than today. This trend is driven by the increasing spread and capabilities of mobile devices on the one hand, and the enhanced communication networks on the other hand. For example, scenarios need to be handled in which a user connects to a computing environment using a personal digital assistant (PDA) with an Internet access provided by a mobile phone. Tools need to be able to track such users, even when they are only temporarily active, change their location, or migrate their application components. Problems that arise with this scenario are the need to recognise and follow instances of applications when network boundaries are crossed and the instances migrate to different execution platforms. Also, users may get temporarily disconnected or work off-line and reconnect to the computing environment through a different communication medium. Questions emerging from such scenarios need to be considered in future monitoring approaches, as the range of hard- and software components involved in distributed computing environments steadily gets broader. Currently, components reaching from host-based systems to mobile digital assistants need to be taken into account for such prognoses, but no limits are foreseeable yet.

A final consideration regarding the general utilisation of on-line tools is the extension of their scope of usage. As the presence of computers is getting pervasive and ubiquitous, daily life will more and more be influenced by automatically controlled procedures. Architectures for integrating any kind of static or mobile devices like Jini [Sun2000] are an indicator for this trend. In these highly distributed and dynamic environments, on-line tools for observing or steering attached devices will be indispensable. Our contributions for dealing with heterogeneous middleware are transferable to such scenarios in a straightforward way. Thus, a realistic and feasible

perspective for our monitoring approach is to leave the pure software development and deployment scenarios by moving to the construction of on-line tools for any kind of distributed devices. Steering the coffee maker in the kitchen from a laptop in the office is only a small but useful example for this vision.

Bibliography

[ACD+1996] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[AHU1975a] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1975.

[AHU1975b] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1975.

[Ale1999] Alenia Marconi Systems. SEEDS Official Webpage, 1999. `http://www.lti.alenia.it/ep/seeds.html`.

[Alh1998] Sinan Si Alhir. *UML in a Nutshell*. O"Reilly, 1998.

[Alt2000] AltaVista Company. Altavista. WWW, Feb 2000. `http://www.altavista.com/`.

[BBB+1991] T. Bemmerl, A. Bode, P. Braun, O. Hansen, T. Treml, and R. Wismüller. The Design and Implementation of TOPSYS. SFB-Bericht 342/16/91 A, Technische Universität München, July 1991. `http://wwwbode.in.tum.de/~wismuell/pub/design.ps.gz`.

[BBH+1998] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, and M. F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1), 1998.

[BdSLR1997] Sebastiano Bottalico, Filippo de Stefani, Thomas Ludwig, and Günther Rackl. SEEDS - Simulation Environment for the Evaluation of Distributed Traffic Control Systems. In Christian Lengauer, Martin Griebel, and Sergei Gorlatch, editors, *Euro-Par'97 — Parallel Processing*, number 1300 in Lecture Notes in Computer Science, pages 1357–1362, Berlin, 1997. Springer.

[BEA2000] BEA Systems Inc. BEA Manager — White Paper. WWW, March 2000. `http://www.beasys.com/products/manager/index.html`.

[BH1996] Ali Behforooz and Frederick J. Hudson. *Software Engineering Fundamentals*. Oxford University Press, 1996.

[BJ1987] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating system principles*, pages 123–138, Austin, Texas, USA, November 1987.

[BKLW2000] Arndt Bode, Wolfgang Karl, Thomas Ludwig, and Roland Wismüller. Monitoring Technologies for Parallel On-Line Tools. In *SFB 342 Final Colloquium: Methods and Tools for the Efficient Use of Parallel Systems*, pages 1–29. Technische Universität München, Munich, August 2000. `http://wwwbode.in.tum.de/~wismuell/pub/sfb00.ps.gz`.

[BMC2000] BMC Software. Application Service Management: Reducing Complexity in Today's Distributed Environmemt — White Paper. WWW, March 2000. `http://www.bmc.com/products/whitepaper.html`.

[BMR+1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley & Sons, 1996.

[BN1984] Andrew D. Birrell and Bruce J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[Bod1995] Arndt Bode. *Werkzeuge zur Entwicklung paralleler Programme*, chapter 13, pages 517–553. In Waldschmidt [Wal1995], 1995.

[Bra1994] Peter Braun. *Visualisierung des Ablaufverhaltens paralleler Programme*. PhD thesis, Technische Universität München, 1994.

[Bri2000] Britannica.com Inc. Encyclopaedia Britannica. WWW, Feb 2000. `http://www.britannica.com/`.

[Bro1999a] Keith Brown. Building a Lightweight COM Interception Framework, Part 1: The Universal Delegator. *Microsoft Systems Journal*, Jan 1999.

[Bro1999b] Keith Brown. Building a Lightweight COM Interception Framework Part 2: The Guts of the UD. *Microsoft Systems Journal*, Feb 1999.

[CFSD1990] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). IETF RFC 1157, May 1990. `http://www.ietf.org/rfc/rfc1157.txt`.

[CHY+1998] P.E. Chung, Y. Huang, S. Yajnik, D. Liang, J. Shih, C.-Y. Wang, and Y.-M. Wang. DCOM and CORBA – Side by Side, Step by Step, Layer by Layer. *C++ Report*, January 1998.

[Dat1995] C. J. Date. *An Introduction to Database Systems — Sixth Edition*. Addison-Wesley, 1995.

[dPHKV1993] Wim de Pauw, Richard Helm, Doug Kimmelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the eight annual conference on Object-oriented programming systems, languages, and applications*, pages 326–337, Washington D.C., USA, Sep 1993.

[DR2000] Philipp Drum and Günther Rackl. Applying and Monitoring Latency-Based Metacomputing Infrastructures. In P. Sadayappan, editor, *Proceedings of the 2000 ICPP Workshop on Metacomputing Systems and Applications*, pages 181–188, Toronto, Canada, August 2000. IEEE Computer Society.

[EE1998] Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[ELW1998] Robert Eckstein, Marc Loy, and Dave Wood. *Java Swin*. O'Reilly, 1998.

[Eur2000a] The European Commission. Information Society Technologies. Brochure, European Communities, 2000. `http://europa.eu.int/comm/research/ist/leaflets/pdf/ist_en.pdf`.

[Eur2000b] The European Commission. Information Society Technologies Programme (IST). WWW, October 2000. `http://www.cordis.lu/ist/home.html`.

[Ewa2000] Timothy Ewald. Use AppCenter Server or COM and MTS for Load Balancing Your Component Servers. *Microsoft Systems Journal*, January 2000.

[FFK+1997] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-performance Distributed Computations. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, 1997. `ftp://ftp.globus.org/pub/globus/papers/hpdc97-mds.ps.gz`.

[FK1998] I. Foster and C. Kesselman. The Globus project: A status report. In *Proceedings of the Heterogeneous Computing Workshop*, pages 4–18. IEEE Computer Society Press, 1998.

[FK1999] Ian Foster and Carl Kesselman, editors. *The Grid – Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1 edition, 1999.

[Fra1997] Fraunhofer-IITB. The CORBA-Assistant – Monitoring of CORBA-based Applications — White Paper. WWW, June 1997. `http://tes.iitb.fhg.de/corba-assistant/cawp697.pdf`.

[GBD+1994] Al Geist, Adam Beguelin, Jack Dongarra, et al. *PVM: Parallel Virtual Machine*. MIT Press, London, 1994.

[GHJV1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Glo2000] The globus project. WWW, February 2000. `http://www.globus.org/`.

[GNO2000] The GNOME Project. WWW, Sep 2000. `http://www.gnome.org/`.

[Gol1995] Brian Goldiez. DIS Technology. In *Proceedings of the 13th Workshop on the Standards for the Interoperability of Distributed Simulations*. Institute for Simulation and Training (IST), 1995.

[Gos1991] Andrzej Goscinski. *Distributed Operating Systems*. Addison-Wesley, 1991.

[GR1993] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, 1993.

[Gro1998] The Open Group. Systems Management: Application Response Measurement (ARM) API. Open Group Technical Standard C807, The Open Group, 1998. `http://www.opengroup.org/onlinepubs/009619299/`.

[Gro2000] The Open Group. The Open Group Portal to DCE. WWW, Feb 2000. `http://www.opengroup.org/`.

[Ham1997] Graham Hamilton. JavaBeans Specification. Technical Report Version 1.0.1, Sun Microsystems, July 1997. `http://java.sun.com/beans/`.

[HAN1999] Heinz-Gerd Hegering, Sebastian Abeck, and Bernhard Neumair. *Integrated Management of Networked Systems*. Morgan Kaufmann, 1999.

[Hei1927] Werner Heisenberg. Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik. *Zeitschrift für Physik*, 3/4(43):172–198, 1927.

[Hei1999] Mareile Heiting. Klassifikation und Evaluierung von Middleware für Client/Server Umgebungen. Diploma Thesis, Technische Universität München, April 1999. In German.

[Hen1998] Michi Henning. Binding, Migration, and Scalability in CORBA. *Communications of the ACM*, 41(10), October 1998.

[IBM2000] IBM. RMI over IIOP. WWW, March 2000. `http://www.ibm.com/java/jdk/rmi-iiop/index.html`.

[Inf1993] *Duden Informatik*. Bibliographisches Institut, Mannheim, Germany, 1993.

[Inp2000] Inprise Corporation. Managing E-business Applications — White Paper. WWW, March 2000. `http://www.inprise.com/appcenter/papers/managing.pdf`.

[Ion2000] Iona Technologies. OrbixManager — White Paper. WWW, February 2000. `http://www.iona.com/`.

[ISO1989] ISO International Organization for Standardization. Information processing systems – Open Systems Interconnection – Basic Reference Model – Part 4: Management framework. ISO/IEC 7498-4:1989, 1989.

[ISO1994] ISO International Organization for Standardization. Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. ISO/IEC 7498-1:1994, 1994.

[ITU1997] ITU International Telecommunication Union. TMN management services and telecommunications managed areas: overview, April 1997. Recommendation M.3200 (04/97).

[JLSU1987] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring Distributed Systems. *ACM Transactions on Computer Systems*, 5(2):121–150, May 1987.

[Joh1997a] Mark W. Johnson. The Application Response Measurement (ARM) API, Version 2. Technical report, Tivloi Systems, December 1997.

[Joh1997b] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, Oct 1997.

[Kac1999] Martin Kacalek. Leistungsuntersuchungen zur Einführung von online Tarifierungsmodulen in verteilten Client/Server Umgebungen. Diploma thesis, Technische Universität München, 1999. In German.

[KDW+1999] Reinhold Kröger, Markus Debusmann, Christoph Weyer, Erik Brossler, Paul Davern, and Aiden McDonald. Automated CORBA-Based Application Management. In Lea Kutvonen, Hartmut König, and Martti Tienari, editors, *Distributed Applications and Interoperable Systems II — IFIP TC 6 WG 6.1 Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, pages 229–241, Helsinki, June 1999. Kluwer Academic Publishers.

[Kor1997] Paul Korzeniowski. *Middleware: Achieving Open Systems for the Enterprise*. Computer Technology Research Corp., Charleston, South Carolina U.S.A., 1997.

[Lam1978] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[LD1998] Chris Loosley and Frank Douglas. *High-Performance Client/Server*. John Wiley & Sons, 1998.

[Lev1999] John R. Levine. *Linkers and Loaders*. Morgan-Kauffman, October 1999.

[Lin2000] Markus Lindermeier. Load Management for Distributed Object-Oriented Environments. In *International Symposium on Distributed Objects and Applications (DOA'00)*, pages 59–68, Antwerp, Belgium, 2000. IEEE Computer Society.

[LK1999] William Leinberger and Vipin Kunar. Information Power Grid: The new frontier in parallel computing. *IEEE Concurrency*, 7(4):75–87, October–December 1999.

[LLF1998] Michael Leventhal, David Lewis, and Matthew Fuchs. *Designing XML Internet Applications*. Prentice-Hall, 1998.

[LS1994] Horst Langendörfer and Bettina Schnor. *Verteilte Systeme*. Hanser, München, Germany, 1994.

[Lud1998] Thomas Ludwig. *Methoden zur effizienten Entwicklung paralleler Werkzeuge*. Habilitationsschrift, Technische Universität München, April 1998. In German.

[LWSB1997] Thomas Ludwig, Roland Wismüller, Vaidy Sunderam, and Arndt Bode. *OMIS — On-Line Monitoring Interface Specification (Version 2.0)*, volume 9 of *Research Report Series, Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM), Technische Universität München*. Shaker, Aachen, 1997.

[May2000] Marcel May. Vergleich von PVM und CORBA bei der verteilten Berechnung medizinischer Bilddaten. Diploma thesis, Technische Universität München, 2000. In German.

[MBBD1999] N. Melab, H. Basson, M. Bouneffa, and L. Deruelle. Performance of Object-Oriented Software Code: Profiling and Instrumentation. In *Proceedings of the IEEE International Conference on Software Maintenance*, Oxford, England, September 1999.

[McD1991] John A. McDermid, editor. *Software Engineer's Reference Book*. Butterworth-Heinemann, Oxford, 1991.

[Mer2000] Mercury Interactive. LoadRunner – The Enterprise Load Testing Tool — White Paper. WWW, March 2000. `http://www.merc-int.com/products/loadrunguide.html`.

[Mic1998] Microsoft Corporation. DCOM Architecture. Technical report, 1998.

[Mil1992] David L. Mills. Network Time Protocol (Version 3): Specification, Implementation and Analysis. IETF RFC 1305, March 1992. `http://www.ietf.org/rfc/rfc1305.txt`.

[MR1991] K. McCloghrie and M. Rose. Management Information Base for Network Management of TCP/IP-based internets: MIB-II. IETF RFC 1213, March 1991. `http://www.ietf.org/rfc/rfc1213.txt`.

[MS1995] Masoud Mansouri-Samani. *Monitoring of Distributed Systems*. PhD thesis, University of London, Imperial College of Science, Technology and Medicine, 1995.

[NBF1996] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly, 1996.

[Obe1998] Michael Oberhuber. *Methoden zur kontrollierten Ausführung nichtdeterministischer kommunizierender Programme*. PhD thesis, Technische Universität München, 1998.

[Obj2000] Object Oriented Concepts Inc. ORBacus, Sep 2000. `http://www.ooc.com/ob/`.

[OMG1995] OMG (Object Management Group). Common Facilities Architecture. Technical report, November 1995.

[OMG1997a] OMG (Object Management Group). A Discussion of the Object Management Architecture. Technical report, January 1997.

[OMG1997b] OMG (Object Management Group). CORBAservices: Common Object Services Specification. Technical report, November 1997.

[OMG1998a] OMG (Object Management Group). minimumCORBA. Technical Report OMG TC document orbos/98-08-04, 1998.

[OMG1998b] OMG (Object Management Group). Objects by Value — Joint Revised Submission. Technical report, January 1998.

[OMG1998c] OMG (Object Management Group). The Common Object Request Broker: Architecture and Specification — Revision 2.2. Technical report, February 1998.

[OMG1999] OMG (Object Management Group). Java Language to IDL Mapping. Technical report, March 1999.

[OMG2000a] OMG (Object Management Group). Event Service Specification. Technical report, June 2000. Version 1.0.

[OMG2000b] OMG (Object Management Group). Naming Service Specification. Technical report, April 2000. Version 1.0.

[OMG2000c] OMG (Object Management Group). Time Service Specification. Technical report, May 2000. Version 1.0.

[Ort1999] Lars Orta. Mechanismen zur Beobachtung des dynamischen Verhaltens verteilter objekt-orientierter Umgebungen . Diploma thesis, Technische Universität München, 1999. In German.

[Pla1999] David S. Platt. *Understanding COM+*. Microsoft Press, 1999.

[PR2000] Arno Puder and Kay Römer. *MICO. An Open Source CORBA Implementation.* dpunkt Verlag, 2000. `http://www.mico.org/`.

[PS1998] František Plášil and Michael Stal. An Architectural Overview of Distributed Objects and Components in CORBA, Java RMI, and COM/DCOM. *Software Concepts & Tools*, 19(1), 1998.

[RA1981] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[Rac1997] Günther Rackl. *Load Distribution for CORBA Environments*. Diploma thesis, Technische Universität München, 1997.

[Rac1999] Günther Rackl. Multi-Layer Monitoring in Distributed Object-Environments. In Lea Kutvonen, Hartmut König, and Martti Tienari, editors, *Distributed Applications and Interoperable Systems II — IFIP TC 6 WG 6.1 Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, pages 265–270, Helsinki, June 1999. Kluwer Academic Publishers.

[Rac2000] Günther Rackl. Monitoring Globus Components with MIMO. SFB-Bericht 342/06/00 A, Technische Universität München, 2000. `ftp://ftp.leo.org/pub/comp/doc/techreports/tum/informatik/report/2000/%TUM-I0006.ps.gz`.

[Rat2000] Sabine Rathmayer. *Online-Visualisierung und interaktive Steuerung paralleler und verteilter technisch-wissenschaftlicher Anwendungen in heterogenen Umgebungen*. PhD thesis, Technische Universität München, 2000. In German.

[RdSH+1999a] Günther Rackl, Filippo de Stefani, Francois Héran, Antonello Pasquarelli, and Thomas Ludwig. Airport Simulation using CORBA and DIS. In Peter Sloot, Marian Bubak, Alfons Hoekstra, and Bob Hertzberger, editors, *High-Performance Computing and Networking — 7th International Conference, HPCN Europe 1999, Amsterdam, The Netherlands*, number 1593 in Lecture Notes in Computer Science, pages 70–79, Berlin, April 1999. Srpinger.

[RdSH+1999b] Günther Rackl, Filippo de Stefani, Francois Héran, Antonello Pasquarelli, and Thomas Ludwig. Distributed Airport Simulation using CORBA and DIS. In Peter Croll and Hesham El-Rewini, editors, *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 166–173, Los Alamitos, CA, May 1999. IEEE Computer Society.

[RdSH+2000] Günther Rackl, Filippo de Stefani, Francois Héran, Antonello Pasquarelli, and Thomas Ludwig. Airport simulation using CORBA and DIS. *Future Generation Computer Systems*, 16(5):465–472, May 2000. `http://www.elsevier.nl/gej-ng/10/19/19/41/28/29/article.pdf`.

[RE1997] Rosemary Rock-Evans. *Middleware and the Internet*. Ovum Ltd, London, UK, 1997.

[Red1999] Red Hat Inc. ORBit, Nov 1999. `http://www.labs.redhat.com/orbit/`.

[RJB1998] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[RL1997] Günther Rackl and Thomas Ludwig. SEEDS SE Architecture Description. Technical Report ESPRIT Project EP 22691, SEEDS Consortium, 1997.

[RL2001] Günther Rackl and Thomas Ludwig. A Methodology for Efficiently Developing On-Line Tools for Heterogeneous Middleware. In *Proceedings of the HICSS-34 Conference*, January 2001. Accepted for HICSS-34. To appear.

[RLRS2000] Günther Rackl, Markus Lindermeier, Michael Rudorfer, and Bernd Süss. MIMO — An Infrastructure for Monitoring and Managing Distributed Middleware Environments. In Joseph Sventek and Geoffrey Coulson, editors, *Middleware 2000 — IFIP/ACM International Conference on Distributed Systems Platforms*, volume 1795 of *Lecture Notes in Computer Science*, pages 71–87. Springer, April 2000.

[RM1990] M. Rose and K. McCloghrie. Structure and Identification of Management Information for TCP/IP-based Internets. IETF RFC 1155, May 1990. `http://www.ietf.org/rfc/rfc1155.txt`.

[Röd1998] Christian Röder. *Load Management Techniques in Distributed Heterogeneous Systems*. PhD thesis, Technische Universität München, 1998.

[RP1999] Peter Rechenberg and Gustav Pomberger, editors. *Informatik-Handbuch*. Carl Hanser, München, Germany, 1999.

[RS1997] Günther Rackl and Thomas Schnekenburger. Dynamic Load Distribution for Orbix Applications: Integration and Evaluation. In *2nd Component User's Conference*, Munich, Germany, July 1997.

[Rud1999] Michael Rudorfer. Visualisierung des dynamischen Verhaltens verteilter objekt-orientierter Anwendungen. Diploma thesis, Technische Universität München, 1999. In German.

[RWMB1998] Diane T. Rover, Abdul Waheed, Matt W. Mutka, and Aleksandar M. Bakic. Software Tools for Complex Distributed Systems: Toward Integrated Tool Environments. *IEEE Concurrency*, pages 40–54, April-June 1998.

[RZB1999] Günther Rackl, Ivan Zoraja, and Arndt Bode. Distributed Object Computing: Principles and Trends. In *International Conference on Software in Telecommunications and Computer Networks – SoftCOM '99*, pages 121–132, Oct 1999.

[Sch1997] A. Schade. An Event Framework for CORBA-Based Monitoring and Management Systems. In *International Conference on*

Open Distributed Processing (ICOPD) and Distributed Platforms (ICDP), May 1997.

[Sed1988] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1988.

[Seg2000] Segue Software Inc. Silk for CORBA — White Paper. WWW, March 2000. `http://www.segue.com/`.

[SFK+1998] P. Stelling, I. Foster, C. Kesselman, C.Lee, and G. von Laszewski. A Fault Detection Service for Wide Area Distributed Computations. In *Proc. of the 7th IEEE Symp. on High Performance Distributed Computing*, pages 268–278, 1998. `ftp://ftp.globus.org/pub/globus/papers/hbm.ps.gz`.

[SMSP1996] A. Schill, Ch. Mittasch, O. Spaniol, and C. Popien, editors. *Distributed Platforms*. Chapman & Hall, London, 1996.

[SOHL+1999] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, 2nd edition, 1999.

[SR1997] Thomas Schnekenburger and Günther Rackl. Implementing dynamic load distribution strategies with orbix. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pages 996–1005, Las Vegas, August 1997.

[Sta1998] William Stallings. *Operating Systems – Internals and Design Principles*. Prentice-Hall, 1998.

[STK1996] A. Schade, P. Trommler, and M. Kaiserswerth. *Object Instrumentation for Distributed Application Management*, pages 173–185. In Schill et al. [SMSP1996], 1996.

[Sun1990] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[Sun1995a] Sun Microsystems. RPC: Remote Procedure Call Protocol Specification Version 2. IETF RFC 1831, August 1995. `http://www.ietf.org/rfc/rfc1831.txt`.

[Sun1995b] Sun Microsystems. XDR: External Data Representation Standard. IETF RFC 1832, August 1995. `http://www.ietf.org/rfc/rfc1832.txt`.

[Sun1997] Sun Microsystem Inc. Java Remote Method Invocation Specification — Revision 1.2. Technical report, October 1997.

[Sun2000] Sun Microsystems Inc. Jini Architecture Specification. Technical report, October 2000. `http://www.sun.com/jini/`.

[Süs1999] Bernd Süss. Konzepte und Mechanismen zum on-line Monitoring von DCOM-Anwendungen. Diploma thesis, Technische Universität München, 1999. In German.

[TK1998] Owen Tallman and J. Bradford Kain. COM versus CORBA: A Decision Framework. *Distributed Computing*, Sep-Dec 1998.

[Tre1996] Thomas Treml. *Monitoring paralleler Programme*. PhD thesis, Technische Universität München, 1996.

[Tri1999] Jörg Trinitis. *Interoperability of Distributed Checkpointing and Debugging Tools*. PhD thesis, Technische Universität München, 1999.

[Vin1998] Steve Vinoski. New Features for CORBA 3.0. *Communications of the ACM*, 41(10):44–52, October 1998.

[Wah1998] Günther Wahl. UML kompakt. *Objektspektrum*, February 1998.

[Wal1995] Klaus Waldschmidt, editor. *Parallelrechner: Architekturen - Systeme - Werkzeuge*. Teubner, 1995.

[Weg2000] Wego.com Inc. Gnutella. WWW, October 2000. `http://gnutella.wego.com/`.

[WLB+1997] Roland Wismüller, Thomas Ludwig, Arndt Bode, Rolf Borgeest, Stefan Lamberts, Michael Oberhuber, Christian Röder, and Georg Stellner. THE TOOL-SET Project: Towards an Integrated Tool Environment for Parallel Programming. In Xu De, K.-E. Großpietsch, and C. Steigner, editors, *Proceedings of Second Sino-German Workshop on Advanced Parallel Processing Technologies, APPT'97*, pages 9–16, Koblenz, Germany, September 1997. Verlag Dietmar Fölbach. `http://wwwbode.informatik.tu-muenchen.de/~wismuell/pub/appt97.ps.gz`.

[WTL1998] Roland Wismüller, Jörg Trinitis, and Thomas Ludwig. OCM — A Monitoring System for Interoperable Tools. In *Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT'98*. ACM Press, 1998.

[X/O1991] X/Open Ltd. *X/Open Common Application Environment — Distributed Transaction Processing: Reference Model*. Reading, Berkshire, England, 1991.

[ZL1995] Krzysztof Zielinski and Aleksander Laurentowski. A Tool for Monitoring Heterogeneous Distributed Object Applications. In *Proceedings of the 15th Int. Conference on Distributed Computing Systems*, Vancouver, Canada, May 1995.

[Zor2000] Ivan Zoraja. *Online Monitoring in Software DSM Systems*. PhD thesis, Technische Universität München, 2000.

[ZRL1999] Ivan Zoraja, Günther Rackl, and Thomas Ludwig. Towards Monitoring in Parallel and Distributed Environments. In *International Conference on Software in Telecommunications and Computer Networks – SoftCOM '99*, pages 133–141, Oct 1999.

[ZRS+2000] Ivan Zoraja, Günther Rackl, Martin Schulz, Arndt Bode, and Vaidy Sunderam. Modern Software DSM Systems: Models and Techniques. SFB-Bericht, Technische Universität München, 2000. To appear.

Index

F

G

H

I

J

L

M

N

U

W

X