MOnitoring Distributed Object and Component Communication

Nikolay K. Diakov

Centre for
Telematics and
Information
Technology

Telematica
Instituut

Enschede, The Netherlands, 2004

CTIT PhD.-thesis series number 04-63
Telematica Instituut Fundamental Research Series, No. 012 (TI/FRS/012)

Cover Design: Studio Oude Vrielink, Losser and Jos Hendrix, Groningen
Cover Image: Nikolay Diakov
Book Design: Lidwien van de Wijngaert and Henri ter Hofte
Printing: Universal Press, Veenendaal, The Netherlands

Graduation committee:

Chairman, secretary:	prof. dr. W.H.M. Zijm (Universiteit Twente)
Promotor:	prof. dr. ir. C.A. Vissers (Universiteit Twente)
External expert:	dr. Xavier Logean (Cap Gemini Ernst & Young)
Members:	dr. ir. M. van Sinderen (Universiteit Twente)
	prof. dr. ir. L.J.M. Nieuwenhuis (Universiteit Twente)
	prof. dr. ir. E. Backer (Delft University of Technology)
	prof. dr. Thomas Plagemann (University of Oslo)

CTIT Ph.D.-thesis series, No. 04-63
ISSN 1381-3617; No. 04-63
Centre for Telematics and Information Technology, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

Telematica Instituut Fundamental Research Series, No. 012
ISSN 1388-1795; No. 012
Telematica Instituut
P.O. Box 589, 7500 AN Enschede, The Netherlands

Telematica Instituut Fundamental Research Series (see http://www.telin.nl/publicaties/frs.htm)

ISBN 90-75176-38-4

Copyright © 2004, N.K. Diakov, The Netherlands

All rights reserved. Subject to exceptions provided for by law, no part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the copyright owner. No part of this publication may be adapted in whole or in part without the prior written permission of the author.

MONITORING DISTRIBUTED OBJECT AND COMPONENT COMMUNICATION

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof.dr. F.A. van Vught,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op woensdag 16 juni 2004 om 15.00 uur

door

Nikolay Kirilov Diakov
geboren op 20 November 1973
te Yambol, Bulgarije

Dit proefschrift is goedgekeurd door:
prof. dr. ir. C.A. Vissers (promotor)

Abstract

This thesis presents our work in the area of monitoring distributed software applications (DSAs). We produce three main results: (1) a design approach for building monitoring systems, (2) a design of a system for MOnitoring Distributed Object and Component Communication (MODOCC) behavior in middleware-based applications, and (3) a proof-of-concept implementation of this system.

Monitoring execution aspects of DSAs plays an essential role in improving their quality in terms of user expectations, performance, and reliability. For example, monitoring communication between DSA parts produces information used for discovery of errors and their sources, fault and performance analysis, and also for balancing the work done by system components.

Designers and programmers often build utility monitoring systems to support the testing, and operation and maintenance phases of the lifecycle of a DSA product. For this, a monitoring system needs to employ models and mechanisms for maintaining a consistent view on DSA execution, and when necessary to deliver information about application execution during runtime.

This thesis focuses on monitoring of DSAs built with object and component technologies, and in particular on the aspects of object and component execution, such as inter-object and inter-component interaction.

The manuscript has the following structure:

Chapter 1 introduces the area of research, describes in further detail our motives for this work, and establishes our goals.

Chapters 2 and 3 introduce terminology and concepts needed throughout the manuscript.

Chapter 2 presents the basic terminology and fundamental concepts in the area of monitoring distributed software applications.

Chapter 3 presents an overview of object and component middleware technologies.

Chapter 4 presents and evaluates most relevant existing monitoring systems, focusing on systems supporting object and component middleware. As a result of this evaluation, we define a set of requirements for our monitoring system.

Chapter 5 describes a design approach for monitoring systems. The design approach consists of four stages: Generic Monitoring System (GMS) design, GMS specialization, instrumentation design, and monitor design.

Chapters 6, 7, and 8 follow our design approach in order to produce a system for monitoring middleware-based applications.

Chapter 6 proposes an architecture of a GMS. The GMS addresses only generic requirements for monitoring. This chapter also reports on the prototype of a GMS that we have built.

Chapter 7 presents the design of a MODOCC system. The design includes a monitoring model for monitoring object and component communication, and a design of a middleware instrumentation. This chapter also presents the prototype implementation of the instrumentation.

Chapter 8 presents the design and implementation of a basic monitor for visualizing object and component communication. This chapter also presents the use of the MODOCC system and the monitor in three different monitoring applications.

Chapter 9 presents a summary of our contributions and discusses possible directions for further research.

Acknowledgements

Many people have contributed to this work in one way or another. Below I present my words of gratitude and acknowledgement.

I thank Chris Vissers for giving me the opportunity to work on my PhD research. I deeply respect his commitment to his students. Marten van Sinderen I thank for his support from the very beginning. He helped me to focus my PhD research. I admire him for his inextinguishable energy and passion in pursuing the strategic, research, and financing matters of the Architecture group at Twente University. Dick Quartel has greatly contributed to my PhD research, especially in the writing phase. He has read and reviewed more draft manuscript versions than anyone else. I thank him for his patience and dedication.

Many thanks to present and past colleagues at the Architecture group for their productive discussions and friendship: Kris, Giancarlo, Joao-Paulo, Maarten, Patricia, Renata, Helen, Remco van de Meent, Remco Dijkman, Clever, Ciro, Alex, Valerie, Robert, Szabi, Diego, Jose, Marcel, Luis, Aart, Marlous, Aiko, Wilma, and Bert. I thank Maarten for reviewing the Dutch translation of the Abstract. I would also like to thank the colleagues at the Telematica Institute and Lucent Technology with whom I had productive collaboration within the FRIENDS project: Hans Zandbelt, Marten Wibbels, and Harold Batteram.

I would like to extend my gratitude to all my friends who made pleasant my stay in Enschede: Goran, Tony, Stanislav, Ina, Vania, Nikolay Kavaldjiev, Nikolay Dokovski, George, Lilith, Gloria, Pedro, Dessi, Samuil, Zlatko, Ivayla, Ulrich, Val, Lora, Boriana, Ivan, Dano, Marcos, Kelen, Christiaen Slot, Carla, Nynke, Manon, Jasper, Herman, Koen, and Wietze. I hope I did not forget anyone.

I would like to thank my family and relatives for their support from afar: Irina, Kiril, Nelly, Zdravko, Margarita, and Todor. My partner Galina has supported me through all the difficult times during my PhD research. I thank her for her patience.

I would like to thank Farhad Arbab and Jan Rutten for giving me the opportunity to continue my scientific career at the SEN3 group, CWI, and complete the final manuscript of my PhD thesis. I also thank Tomas from the SEN4 group, CWI, for reviewing the Dutch translation of the abstract.

Last but not least, I would like to thank David Bourland and the other members of the International Society of General Semantics, who have contributed to the development and popularization of E-Prime [Bour91]. I wrote this thesis not in English, but in E-Prime.

Nikolay Kirilov Diakov
Amsterdam, May 2004

Contents

Chapter 1

Introduction

This thesis addresses the area of monitoring the communication in distributed software built using object and component middleware technologies. This chapter presents the background and the motivation for this work, discusses the problems in the area, defines the scope and the objectives, and presents the approach that we follow. The chapter concludes with an overview of the thesis structure.

1.1 Background

Nowadays, we routinely use computer systems in our environment. Personal computers running various software applications assist us in repetitive, error-prone and time-consuming tasks, such as personal time management, financial bookkeeping and word processing.

Advances in communication technologies enable computers to interact over great distances and thus to common tasks. This led to the development of distributed systems consisting of distributed software, computing devices, communication devices and underlying communication networks.

A distributed software application (DSA) represents software that runs on a distributed system. DSAs enable the interaction among (potentially many) geographically distant users, they allow the utilization and sharing of physically remote resources, such as content and services, and (if designed properly) they can provide higher availability, performance, and reliability compared to centralized systems. Examples of DSAs include file sharing, instant messaging, multi-user online gaming, electronic banking, e-mail, and the World Wide Web.

As users become increasingly dependent on DSAs, the quality of these DSAs becomes an important issue. For example, failure in a DSA that automates some business process involving different organizations, may lead to unfulfilled contractual obligations, loss of money, and in some cases, threat to human health and life.

Industry and the academia have spent a great deal of effort to develop a general software manufacturing process, such as the Rational Unified Process [JBR99], aimed at optimizing the production of DSAs within strict budget and time constraints, and at the same time increasing the degree of customer satisfaction. As a result, most contemporary approaches for software development follow several general phases: requirements analysis, specification, design, implementation, testing and validation, and operation and maintenance [Royce87].

Monitoring the execution of DSAs can play an important role in two of these phases: testing and validation, and operation and maintenance.

During the testing and validation phase, testers execute a DSA in order to make conclusions about its runtime behavior [KraWis98]. In this phase, monitoring serves as the enabling mechanism for several categories of activities [Ward01]:

- *Event inspection* – Developers require some useful information about the DSA execution, at the moment of the occurrence of some event in the behavior of the DSA. For example, in the event of sending a message from one DSA part to another, the developer may require inspection of the individual parameters of the message;
- *Conformance testing* – Developers may require checking whether the design of a DSA allows certain selected scenarios of use of a DSA prototype. Furthermore, developers may need to discover whether certain patterns appear in the behavior of a DSA implementation. For example, the implementation may allow transmitting certain types of messages in a particular order that the design does not allow;
- *Computation replay* – Developers require the re-execution of a previously monitored and recorded execution of a DSA in order to reproduce and closely examine certain effects of its behavior, e.g., a difficult to reproduce fault or a race condition;
- *Distributed breakpoints* – A distributed breakpoint has similarity with a breakpoint from sequential debugging, however the condition for reaching a distributed breakpoint now may require monitoring the progress of several concurrent activities within the DSA behavior;
- *Differentiation* – Developers may need to compare executions of a DSA, for example, to determine how different environment conditions affect the DSA execution, or to determine whether fixing a bug has actually made any difference (successful fixing);
- *Visual presentation* – Visualizing the execution of the DSA deals with representing the DSA visually in a way that meets the requirements of the testers. Visualization often has to deal with the constraints of human

comprehension, such as, number of objects on the canvas, frequency of updates, etc.

The operation and maintenance phase includes various management activities on DSAs, which require monitoring of the state, the errors, and the performance [SloMo89]. The management of a DSA comprises supervising and controlling the system so that it fulfils the requirements of both its owners and its users [Slo95]. According to the ISO/OSI management standards, management activities include *fault management*, *configuration management*, *accounting management*, *performance management* and *security management* [ISO90][ISO92], each of which depends on information collected through monitoring.

1.2 Software monitoring

The Merriam-Webster Online Dictionary defines to *monitor* as "to watch, keep track of, or check, usually for a special purpose" [M-W]. Further in this text we use the terms "to observe" and "to monitor" interchangeably.

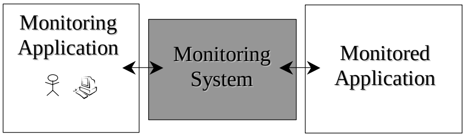


Figure 1-1 Parties involved in monitoring

We call *software monitoring* the process of observing various aspects of the execution of some *monitored application* (Figure 1-1). The communication between remote application parts constitutes an example of an execution aspect that one can monitor in a distributed environment.

Software designers may not have designed an application with monitoring in mind. In order to prepare such an application for monitoring, designers *instrument* the application, e.g., add or change something in it or in its execution environment.

We call the party interested in monitoring the *monitoring application*. A monitoring application involves computer software and may involve human operators. A monitoring application requires information about the execution of the monitored application for the purpose, for example, of testing or management.

A *monitoring system* supports the monitoring. In a typical scenario, a monitoring system performs *measurements* on the monitored application, packages the results into *monitoring data*, and presents the monitoring data to the monitoring application.

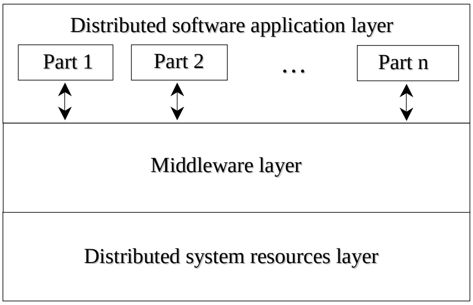
1.3 Middleware-based systems

Designers of different DSAs often have to solve similar tasks, such as how to make application parts discover each other in order to communicate, how to make their applications work in a heterogeneous environment, or how to deal with partial failure. This observation became one of the reasons for the appearance of the middleware concept.

Middleware comprises system software that provides a set of reusable common services and network programming mechanisms. Middleware resides between the applications and the underlying operating systems, network protocol stacks, and hardware [SSRB00]. The middleware coordinates the interactions among application parts by providing functionality that bridges the gap between software applications and the low-level hardware and software [SchSch01].

Middleware helps developers to increase their productivity by shielding them from (potentially) error-prone low-level details of the runtime environment. Middleware simplifies the development of DSAs by offering high-level programming abstractions conceptually closer to application requirements than the low-level programming methods. For example, middleware can offer distribution transparency by hiding low level network programming into a reusable object-oriented framework for remote operation invocations.

Figure 1-2 Middleware-based system


Distributed software application layer
Part 1
Part 2
...
Part n
Middleware layer
Distributed system resources layer

In a commonly used model of a middleware-based system, we can find one or more middleware layers (Figure 1-2) between the DSA layer and the layer of distributed system resources [SchSch01]. For simplicity we consider only one middleware layer. The middleware layer provides to the DSA layer high-level distributed programming interfaces, e.g., for invoking operations on remote objects, or for accessing common domain-independent services, such as directory, transaction, and security services. The middleware layer takes care of any details on behalf of the DSA about the allocation, scheduling and coordination of resources in the distributed resources layer, in order to hide the peculiarities of individual operating systems, and help eliminate many tedious, error-prone and non-portable aspects of low-level OS programming.

Over the past decade, various middleware technologies have addressed the complexities associated with the development of DSAs. We consider two of these technologies: object middleware and component middleware.

Object middleware

Professionals have widely applied the Object-Oriented (OO) approach to handle complexity in software design. The OO design emphasizes on abstraction, encapsulation, modularity and inheritance. These principles provide reusability of system behavior, support for incremental software development and systematic system decomposition. We can perceive an object-oriented computer program as a collection of objects, which communicate with each other to achieve a common goal [Booch91].

The application of the OO approach to distributed environments resulted in the advent of *object middleware*. In a distributed environment, inter-object communication may cross the boundaries of a single computer. The middleware takes care of communication-related issues, such as the synchronization between communicating parties, reliability of communication, heterogeneity issues of different hardware and software involved in the communication, and interoperability between different vendor products. CORBA [CORBA] and DCOM [DCOM] constitute two examples of object middleware technology.

Component middleware

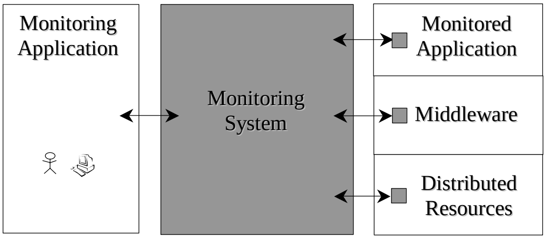
The industrialization of the software production process led to the evolution of general software development into component-based software development. *Component middleware* integrates the middleware concept with the concept of component-based software development. We can compare components to object types, except that components represent coarse-grained prefabricated building blocks, which we can use to assemble our software applications in a manner similar to electronic circuits and mechanical parts [Szy98]. In the recent years, several component middleware technologies emerged, such as, COM+ [COM+][.NET], EJB [EJB] and CORBA Components [CCM][CORBA3]. The companies developing these technologies aim to support rapid software development from reusable and composable off-the-shelf software components.

Further in the text we often use the term “middleware” to refer both to object and component middleware technologies. A DSA built using middleware technologies we call a *middleware-based application*.

1.4 Monitoring of middleware-based applications

In general, monitoring requires instrumenting the application and/or its execution environment. Instrumenting the application allows one to monitor the application execution *directly*. Instrumenting the execution environment allows one to monitor application execution *indirectly*, based on the services that the application uses from its environment. The middleware and the distributed resources layer comprise the execution environment of a middleware-based application. The monitoring system can observe execution aspects at each layer of the middleware-based system. Figure 1-3 shows a general scenario for monitoring middleware-based applications. This scenario represents a middleware-based system with instrumented layers.

Figure 1-3 Monitoring of a middleware-based application



The monitoring system communicates with the instrumented components in each layer, in order to collect monitoring data and deliver it to the monitoring application. Note that we consider any instrumentation code necessary for enabling observation of the monitored application as part of the monitoring system.

The instrumentation in the monitored application layer allows monitoring of the execution aspects of specific application logic. Developers typically build application level instrumentation per application, which makes development costs proportional to the number of monitored applications. The instrumentation in the middleware layer allows monitoring of application execution aspects that rely on services of the middleware. Object and component lifecycle, and remote operation invocations, constitute two examples of such execution aspects. The instrumentation in the distributed resources layer allows monitoring of application execution aspects in terms of utilization of low-level resources, such as, OS processes, network bandwidth, CPU load, etc. Instrumenting the middleware or the distributed resources layers makes the instrumentation generic to applications built with this middleware, because the middleware shields the instrumentation from the applications.

In a distributed environment, the monitoring application and the monitoring system themselves require communication between their physically remote parts. Hence, designers of monitoring applications and monitoring systems may choose to benefit from the advantages of middleware. Figure 1-4 shows a middleware-based refinement of the general scenario for monitoring of middleware-based applications (presented in Figure 1-3).

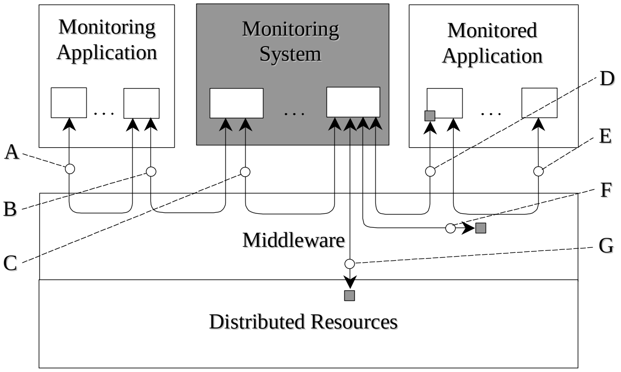


Figure 1-4
Middleware-based monitoring of middleware-based applications

We categorize the interactions between the different system parts involved in monitoring into seven types:

- (A) The parts of the monitoring application interact with each other;
- (B) The monitoring application interacts with the monitoring system;
- (C) The parts of the monitoring system interact with each other;
- (D) The monitoring system interacts with its application level instrumentation;
- (E) The parts of the monitored application interact with each other;
- (F) The monitoring system interacts with its middleware instrumentation;
- (G) The monitoring system interacts with its distributed resources instrumentation.

This categorization illustrates that using the middleware to build monitoring systems combines well with monitoring of middleware-based applications. In general, the communication mechanisms in the middleware handle interaction types A, B, C, D, E. Interaction types F and G represent local interactions, namely between a local agent of the monitoring system and the instrumentation. The monitoring system uses interactions of type C for transferring information to remote locations.

Monitoring communication behavior

Communication behavior constitutes an essential part of the overall behavior of a middleware-based application. The middleware offers a set of basic building blocks, such as remote operation invocations, which designers use to build the communication behavior of their applications. During runtime, the middleware mediates each individual invocation. Hence, instrumenting the middleware can make information about individual invocations available to the monitoring system and subsequently to monitoring applications, such as testing suites and management systems. Instrumenting the middleware however, cannot provide information about any application-specific relation between individual invocations. Designers can obtain this information by instrumenting the application, given they have access to its design and/or its implementation source code.

1.5 Problem statement

In this section we identify several problems in the area of monitoring. We focus on three groups: problems related to monitoring in a distributed environment, problems related to monitoring at the middleware layer, and problems related to the design of monitoring systems in general.

1.5.1 Monitoring in distributed systems

Several basic characteristics of distributed systems give rise to monitoring problems: concurrency, hardware clocks, common resources, and scalability.

A typical distributed system allows for physically parallel execution of concurrent activities. Monitoring the combined progress of such activities may prove difficult because of, for example, the lack of a global hardware clock in low cost distributed systems.

A distributed system may include different operating systems. To reduce the cost of a distributed system, organizations often use low cost operating systems that do not provide strict mechanisms for scheduling of the distributed system resources. DSAs running on such distributed systems may compete for a *common resource*. Since one may consider a monitoring system that monitors a DSA as a DSA itself, the sharing of common resources in distributed system may cause various performance problems.

Distributed systems may change their size dynamically, depending on how many different locations participate in the system at any moment. This gives rise to *scalability* issues in monitoring systems that operate on large and dynamic distributed systems.

Below we formulate the following five problems related to monitoring of a DSA:

i – *Inability to accurately establish temporal relations between observed events.* Testers require to establish the temporal order among the (occurrence of) events representing executed application activities for the purpose of, e.g., locating errors in a DSA prototype. The traditional perception of absolute global time seems insufficient to reflect the relativistic aspects of asynchronous physically distributed systems that suffer from noticeable communication delays [Pratt86]. Absolute global time cannot support the analysis of temporal relations in low cost distributed systems for two main reasons [SchMa94]: (1) the lack of a global hardware clock, and (2) the negative architectural impact of centralized mechanisms for measuring time – a centralized time server presents a single point of failure and a bottleneck for system performance;

ii – *Inability to establish causal relations between observed events*. Reasoning about the causal relations among observed events has applications in analysis of DSAs, e.g. detecting global conditions necessary for distributed breakpoints [CoMa91]. In general, without access to the design of the monitored application we cannot make inferences about causality from post-execution information. Although the limited causal semantics of the “happened-before” relation [Lamp78] seems to provide the basis for expressing causal relations between events, the complexity and inefficiency of the *logical clock* mechanisms used to implement this relation, discourage the use of logical clocks by designers of monitoring systems;

iii – *Inconsistent view on application behavior.* The measurements performed during application runtime, may lead to additional delay (also called *overhead*) in the execution of the monitored application. As a result, the monitoring system cannot measure correctly the original application behavior because the act of measuring changes the application behavior – an effect similar to the “uncertainty principle” from the quantum theory [Heis27]. A monitoring system that produces such delay may change the behavior of the monitored application so much, that the monitoring data obtained from the system does not represent consistently the behavior of the original unmonitored application to the users of the monitoring system [HeBr89]. By consistency we mean, for example, order among observed event occurrences, number of events, etc.;

iv – *Undesirable monitored application behavior*. The overhead of the monitoring system, may lead to undesirable monitored application behavior, such as slow application response time. Furthermore, a monitoring system often shares the resources of the distributed environment, such as communication infrastructure, storage and processing power, with the monitored application. In operating systems that do not provide strict resources scheduling mechanisms, the monitoring system may deprive the monitored application from a limited resource [Shaer98], which as a result may produce undesirable behavior. For example, parts of the monitored application may fail to communicate with other parts (communication timeout), due to the excessive use of communication bandwidth by the monitoring system for communicating monitoring data among its remote parts;

v – *Scalability*. In a large distributed system, the monitored application may consist of many application parts that produce monitoring data. Furthermore, the monitoring application may have many individual consumers of monitoring data (e.g., several instances of some analysis software. Transferring (possibly frequently) large amounts of monitoring data from many producers to many consumers may lead to performance problems in the monitoring system and undesirable application behavior in the monitored application [Sam95].

Observe that problems (*iii*) and (*iv*) also apply to the monitoring of software applications running on standalone systems.

1.5.2 Monitoring at the middleware layer

Monitoring at the middleware layer requires instrumentation of the middleware product used to build the monitored application. A middleware product may comprise various technologies. Designers who need to create a middleware instrumentation may find that this task requires extensive knowledge about the middleware's internal mechanisms, in order to avoid exposing to the monitoring application incomplete information about the internal state of the middleware.

A middleware can provide standardized monitoring interfaces to help designers develop the instrumentation. In *reflective* middleware, for example, application developers can access some of the middleware internal mechanisms through such interfaces, in order to make the middleware more configurable to fit different application requirements, and more adaptable to meet changing environment or user requirements [Weg03]. Current middleware products however, either do not support reflection at all, or provide very limited support for it [Blair98i].

We identify the following two problems related to monitoring at the middleware layer:

vi – *Limited support for monitoring of remote interactions at the middleware layer*. Existing middleware [CORBA][DCOM][EJB] products provide limited standardized access to their communication mechanisms. As a consequence, monitoring systems [Rack01][KR + 00][Logean00] often resort to using proprietary interfaces and ad-hoc middleware instrumentation to supply the necessary information about the distributed interactions of middleware-based applications. Besides the risk of exposing incomplete information about the internal middleware state, this approach also reduces the flexibility and reusability of a monitoring system with different middleware products;

vii – *High-costs for middleware instrumentation*. Manual instrumentation of the middleware can make the software development process expensive and error prone. For example, the CORBA middleware uses the Proxy pattern [BuMe + 96] to provide distribution transparency by generating proxy objects for every individual application object type. Instrumenting the proxy objects [SMST] allows one to monitor the communication between remote application parts. Many application object types (as in a large middleware-based DSA) lead to proportionally many proxy objects that need instrumentation. Development of tools for automatic instrumentation has the potential to reduce instrumentation cost.

1.5.3 Design of monitoring systems

In this section we identify two problems relating to the design of monitoring systems in general:

viii – *High costs for designing monitoring systems*. Several design steps seem to reappear in the design of existing monitoring systems in one or another form, such as: definition of a model of the monitored application [Bates85][KQS92][Hof + 94][Rack99][BeAb02], design of an instrumentation of the monitored application [KQS92][LWSB97][LDKK98], and design of monitoring tools (applications) [JLSU87][KBTB97][RaLe97][Rait00][Rack01]. Reinventing those apparently useful steps can raise the costs for development of monitoring systems. Therefore, designers may benefit from a design approach that integrates and unifies the important steps during the design of monitoring systems. Such methodological support would allow designers to reduce development costs and increase the quality of their monitoring systems;

ix – *Lack of standardized monitoring service and implementation architecture*. Monitoring systems perform many common *monitoring activities*

[Sam95], such as generation, processing and dissemination of monitoring data. Nevertheless, designers often structure their monitoring system in a specific way (e.g., [OLT03]), integrating common monitoring functions in such a way, that it becomes very difficult and expensive to reuse the resulting monitoring system with different monitoring and monitored applications. As a consequence, designers reinvent the support for common monitoring activities in every monitoring system they design.

1.6 Scope and objectives

We set ourselves the overall goal to contribute to the area of monitoring DSAs. Below we further limit the scope of our work and define a list of concrete objectives that address the problems we identified earlier.

We limit our work to the monitoring of middleware-based DSAs. We consider only object and component middleware. Furthermore, we focus on monitoring the aspects of communication behavior of middleware-based applications, such as the remote operation invocations among objects or among component instances. We aim to support the testing and validation phase of DSAs with the results of our work. In this, we limit ourselves to providing partial order relations among events observed in the application behavior, for the purpose of conformance testing, debugging, and visual presentation. We believe that one can use most of our results to support application management during the operation and maintenance phase of DSAs, although we do not explicitly consider management in this thesis.

We address the problems identified in section 1.5 with four concrete objectives:

- *Develop a design approach for monitoring systems.* This approach defines important stages in the design of monitoring systems. By following this approach, designers reduce the development costs and increase the quality of their monitoring systems. This objective addresses problem (*viii*);
- *Propose an architecture for a generic monitoring system*. This architecture includes the definition of the service of a generic monitoring system and a generic software architecture that implements the service. With the proposed architecture we aim to increase the reusability, flexibility, and scalability of monitoring systems. This objective addresses problems (*v*) and (*ix*);
- *Define a monitoring model for object and component communication*. The model uses events to represent operation invocations in middleware-based DSAs. The model allows one to analyze temporal and causal relations

among the events. The model allows a monitor to provide to the monitoring application a consistent view on monitored application behavior. This objective addresses problems (*i*), (*ii*), and (*iii*);

– *Develop a middleware instrumentation and an instrumentation approach for monitoring object and component communication*. The middleware instrumentation provides the information necessary to analyze communication behavior of a running middleware-based DSA, in the terms of the monitoring model for object and component communication. The instrumentation alleviates undesirable monitored application behavior compared to existing approaches. The instrumentation approach includes the design of instrumentation tools that automate the process of instrumenting application objects and component. This objective addresses problems (*iv*), (*vi*), and (*vii*).

The first two objectives address problems general to the monitoring of DSAs and the design of monitoring systems. The second two objectives address problems specific to the monitoring of communication aspects in middleware-based DSAs, as well as some of the more general problems.

Although we have identified the problem of undesirable application behavior due to possible competition over common resources in low cost operating systems, we shall not address this problem explicitly. A solution to this problem requires the investigation of (optimal) deployments of the monitoring system, which falls out of the scope of this work. Furthermore, we limit our work to monitoring at the level of software (thus we do not consider hardware monitoring). Hence, further in the text we use the terms “monitoring” and “software monitoring” interchangeably.

1.7 Approach

In this section we describe our research approach. We start by presenting a fictitious example. We use this example to explain our approach.

1.7.1 Example: tree monitoring

Consider the following example:

Example: Managing tree growth

The municipality in town X needs to maintain the power lines above ground free from growing vegetation (namely trees). At the same time, the municipality does not want to cut vegetation unnecessarily, because the folks of town X like their green environment very much.

The example describes a typical management task. The municipal worker Mr. T[1] has the responsibility to solve the management task of maintaining tree growth (Figure 1-5). Mr. T knows something about management. He decomposes the management task into several management activities: gathering information (monitoring) about the managed system, analysis of the information, and then controlling the system so that it fulfils its purpose. In this case, the managed system consists of the trees in town X. To solve the management task, Mr. T has to design a management system that performs each of the management activities. We shall concentrate on the approach Mr. T follows in the design of the monitoring sub-system.

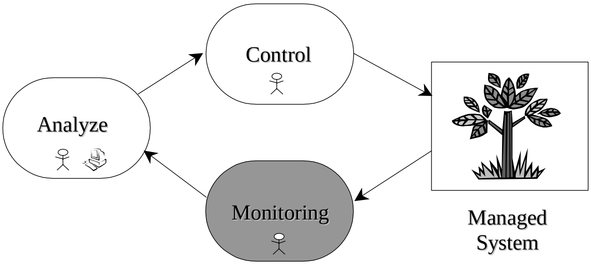


Figure 1-5 The management system for tree maintenance

Mr. T goes to the municipal archive to investigate whether other municipal projects have done monitoring. He finds out that although the municipality has done projects on monitoring, he cannot reuse most of the results, because designers have followed unclear ad hoc approaches specific to every particular application. As a consequence, Mr. T has difficulties extracting any common issues about monitoring from previous projects. Based on the information collected during the visit to the municipal archives, Mr. T identifies the requirement that he needs to define his approach as clearly and as explicitly as possible, if he wants to allow his colleagues to use his results in possible subsequent projects.

Mr. T decides to define a *design approach* based on a generalization of his observations on how previous projects designed monitoring systems for their management tasks. He noticed that designers spent most of their time explaining what to monitor, how to interpret measurements, etc. Therefore, Mr. T decided that his approach will consist of two steps: (1) define a monitoring model that captures and explains all necessary aspects of the managed system, and (2) implement that model by identifying activities for measuring the model aspects, allocating activities to roles (that Mr. T can assign to field workers), and designing the documents required

[1] Mr.T represents a fictional male character. No offence intended to the female municipal workers around the world.

for the system to work, e.g. brochures explaining the model, and forms in which workers fill in measurements. Mr.T follows his design approach to make a monitoring system for tree maintenance.

Using the project requirements (trim trees when necessary, but not too much) and his knowledge about trees Mr. T defines a *monitoring model* of a tree that only takes into account the aspects of a tree that have a relation to the task of maintaining its size (Figure 1-6).

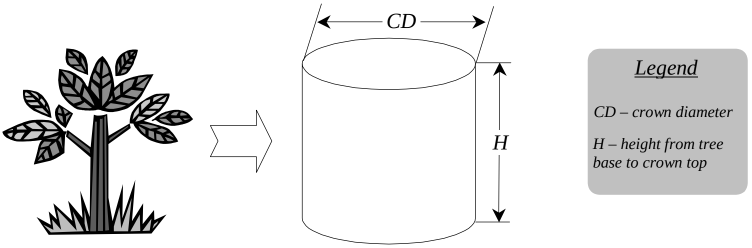


Figure 1-6 The monitoring model of a tree developed by Mr. T

The model defines the outline of a tree as a cylinder with only two parameters that someone can easily measure using the right instruments – crown diameter and tree height.

Mr. T uses the monitoring model to complete the design of the monitoring system: he defines a schedule for measuring trees once a month, allocates several municipal field workers to the role of measuring, and designs the necessary brochures and forms.

Below we summarize the important decisions made by Mr. T:

- *Design approach*: Mr.T decided to define a design approach, which although simple, allowed Mr. T to structure his approach to designing a monitoring system;
- *Monitoring model*: Mr. T identified that the definition of a monitoring model presents an important step in the design of a monitoring system. The model allowed Mr.T to make sure that everybody involved in the tree management task understand the tree aspects important to the completion of the task.

1.7.2 Approach to software monitoring

We consider software monitoring similar to the situation described in the example from section 1.7.1. In software monitoring, a design approach would enhance the general software design process with concrete steps for the design of monitoring systems. A monitoring model defines the aspects of application behavior required by the monitoring application.

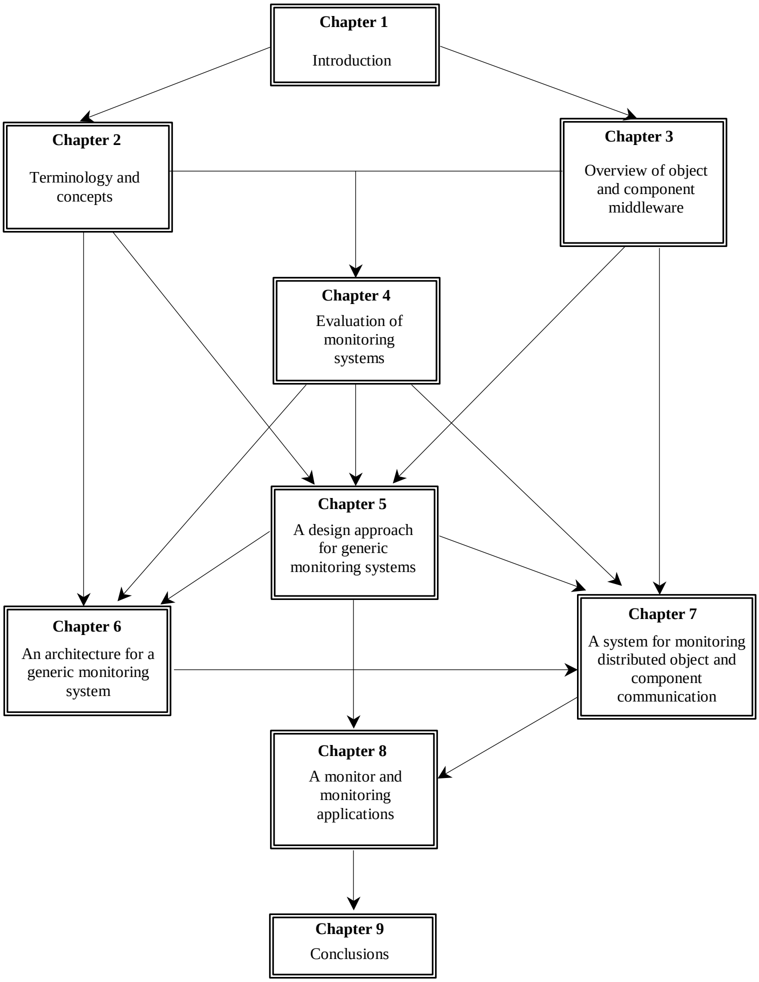
Therefore, our approach addresses the goals of our research in the following steps:

1. Present the concepts and terminology in monitoring, make an overview of object and component middleware technologies, and evaluate the most relevant existing systems for monitoring object and component communication;
2. Based on the evaluation in the previous step, identify certain problems in the area, and use these problems to define requirements for a system for monitoring object and component communication. We distinguish two groups of requirements: generic – requirements relating to problems of monitoring DSAs in general, and specific – requirements that relate to problems specific to monitoring middleware-based DSAs;
3. Propose an approach for the design of monitoring systems. The approach consists of four stages. Stage one deals with the design of a Generic Monitoring System (GMS). The design of a GMS considers generic monitoring functionality, independent from a particular monitored or monitoring application. Stage two deals with the specialization of a GMS for a particular monitored/monitoring application. This step includes the development of a monitoring model. Stage three deals with the design of an instrumentation for a particular monitored application. Stage four deals with the design of a monitor for a particular monitoring application;
4. Follow stage one of the design approach to build an architecture for a GMS. The GMS architecture addresses the generic requirements for monitoring identified in step 2;
5. Follow stages two, three, and four to create a system for monitoring object and component communication. In this step we address the specific requirements identified in step 2. We define a monitoring model for object and component communication, design a middleware instrumentation for that model, and design a basic monitor for visualization of object and component communication;
6. To prove our concept, we provide prototype implementations of the GMS and the instrumentation for object and component middleware. These two constitute the prototype of the Monitoring of Distributed Object and Component Communication (MODOCC) system. We also implement a prototype of a basic monitor for the MODOCC system;
7. To validate the usability of our approach, we apply the MODOCC system and the basic monitor to three monitoring applications: (a) model-based conformance testing of object and component designs, (b) monitoring for testing and debugging of component communication for DSAs built with the Distributed Software Components (DSC) [BaBa98] component middleware, (c) the validation of the Open Service Access API implementation of a UMTS Application Platform.

1.8 Thesis structure

We structure the thesis in chapters with dependencies between the chapters, as shown on Figure 1-7.

Figure 1-7 Thesis roadmap


Chapter 1
Introduction
Chapter 2
Terminology and concepts
Chapter 3
Overview of object and component middleware
Chapter 4
Evaluation of monitoring systems
Chapter 5
A design approach for generic monitoring systems
Chapter 6
An architecture for a generic monitoring system
Chapter 7
A system for monitoring distributed object and component communication
Chapter 8
A monitor and monitoring applications
Chapter 9
Conclusions

Chapter 2 presents the basic terminology and fundamental concepts in monitoring of distributed software.

Chapter 3 presents an overview of object and component middleware technologies.

In Chapter 4 we present and evaluate several existing monitoring systems, focusing on the support for object and component communication in a distributed environment.

Chapter 5 describes a design approach for monitoring systems. The approach consists of four stages: GMS design, GMS specialization, instrumentation design, and monitor design.

Chapter 6 proposes an architecture of a GMS. The GMS addresses only generic requirements for monitoring. This chapter also reports on our GMS prototype.

Chapter 7 describes the design of an instrumentation for monitoring distributed object and component communication for middleware-based applications. In this chapter we present a monitoring model, and a design of a middleware instrumentation. This chapter also presents the prototype implementation of the instrumentation. Together, the prototypes of the GMS and the instrumentation form the MODOCC system.

Chapter 8 presents the design and implementation of a basic monitor for visualizing object and component communication. This chapter also presents the use of the MODOCC system and the monitor prototype in three monitoring applications.

In Chapter 9 we present our conclusions, list the contributions of our work and discuss possible directions for further research.

Chapter 2

Terminology and concepts

This chapter presents the basic terminology and fundamental concepts in monitoring distributed software. The presented material will help designers to understand software monitoring and the important characteristics of monitoring systems.

2.1 General discussion

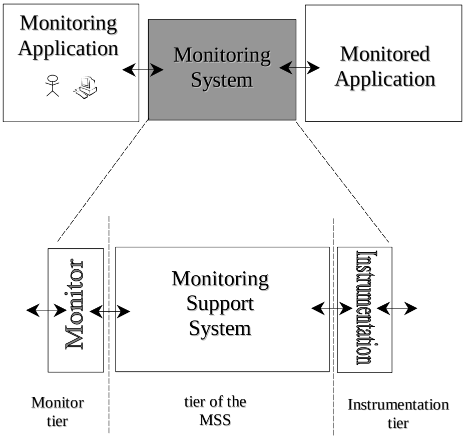
In Chapter 1 we introduced the notion of monitoring system and its supporting role in monitoring. We assume that in general the monitoring application and the monitored application may belong to different domains, and therefore their designers may use different technologies to build them for different platforms. Moreover, the designers of the monitored application may develop it without (future) monitoring in mind. Vice versa, the designers of the monitoring application may build it independently of a particular monitored application. Consequently, the monitoring system should bridge any conceptual and technological gaps between the two domains in order to allow monitoring. Hence, we deal with two separate design concerns: the concern of a particular *monitored application domain*, and the concern of a particular *monitoring application domain*.

Furthermore, we notice that a monitoring system often performs common functions independent of any specific monitored application and monitoring application. Hence, we deal with a third design concern: the concern of *domain-independent monitoring activities*.

2.1.1 Basic terminology

We consider separation of concerns a basic principle in software design for managing complexity. Using this principle, we decompose the monitoring system into three vertical layers, also called *tiers*, to deal with the three concerns identified above (Figure 2-1).

Figure 2-1 Decomposition of the monitoring system



The *monitor tier* represents the monitoring application in the monitoring system. The monitor tier contains monitors. For simplicity, we consider only one monitor in Figure 2-1. A monitor concentrates the knowledge the monitoring system has about a monitoring application. A monitor requests and receives monitoring data from the monitored application on behalf of the monitoring application. A monitor extracts information from the monitoring data, and presents this information to a monitoring application: some software application or a human *operator*. In principle, a monitor may observe the execution of multiple monitored applications and a monitored application may have multiple monitors.

The *instrumentation tier* represents the monitored application in the monitoring system. The instrumentation tier contains instrumentations. For simplicity, we consider only one instrumentation in Figure 2-1. An instrumentation includes all software components added to or modified in a monitored application and/or its execution environment, in order to prepare that application for monitoring. As such, the instrumentation concentrates all conceptual and technological knowledge necessary to capture aspects of the monitored application execution required by the monitoring application.

The tier of the *monitoring support system* (MSS) performs monitoring activities independent from both the specific monitoring application domain and the monitored application domain. An example of such activities constitutes the dissemination of monitoring data in a distributed environment, in which the MSS collects monitoring data from the instrumentation and delivers it to the monitor.

2.1.2 Aspects of a monitoring system

We structure the rest of the chapter using the following three aspects of a monitoring system:

- *Information aspect*: We discuss the way designers model the monitored application in order to capture requirements for information about application execution. We focus on a *monitoring model* and its role in the design of monitoring systems;
- *Functional aspect*: We discuss the various *monitoring activities* performed by the components of the monitoring system;
- *Performance aspect*: We discuss the *overhead* of the monitoring system and how overhead affects the quality of the information presented to the monitoring application by the monitoring system.

2.2 A monitoring model

In general, we consider it impractical (and even impossible) for a monitoring system to observe and present to its users every aspect of the execution of a monitored application. Therefore, the monitoring application requires from the monitoring system an abstract model of the monitored application. This model may include both structural and behavioral aspects.

A Monitoring Model (MM) of the monitored application represents aspects of application execution that the monitoring application finds interesting to observe [Hof + 94]. Designers use an MM to model individual application executions (also called *runs*). An MM differs from a design model in that designers use a design model to model the complete application behavior, i.e., all possible executions of an application. The resulting application model designers often use as a starting point for implementing that application.

In the scope of our research, we seek to define an MM suitable for describing behavioral aspects of the execution of middleware-based applications such as object and component communication. In the next section we discuss the basic modeling concepts that we use to construct such a model.

2.2.1 Modeling concepts

An MM models a software application in terms of *entities* and the *behavior* of these entities.

Entities

The entity concept represents some physical or logical “thing” associated with the monitored application. For example, an entity may represent a process, an object, or a component. Entities may participate in *relations* to form the structure of the monitored application, e.g., the “association” relation between objects, and the “containment” relation between a compound component and one of its sub-components.

Entity behavior

The behavior of an entity represents the dynamic characteristics of the entity during application execution. We consider as dynamic characteristics the *activities* that an entity performs (such as sending a message to another entity) and possible *relations* among those activities (such as the order in which an entity performs two activities).

We distinguish two ways of modeling entity behavior for monitoring: *status-based* and *event-based*.

Status-based modeling

Status-based modeling abstracts from the activities that an entity performs and focuses on the information that the entity maintains at discrete moments of time. We call this information the *status* of the entity. Status-based modeling models activities in a system indirectly, since any changes in the status of an entity result from performing activities.

A *status vector* represents the status of an entity [FelEr89]. A status vector consists of *status variables*. A status variable represents an individual part of the information maintained by an entity. For example, if the entity represents an object, a status variable may correspond to an object attribute. To monitor a status, the monitoring system generates an instance of the status vector by recording the values of the status variables at the required moment of time. We call this instance a *status report*.

Event-based modeling

Event-based modeling directly models the activities that an entity performs. The *event* concept represents the successful completion of some activity performed by an entity of the monitored application. An event either happens, in which case we consider the corresponding activity to have completed, or does not happen, in which case we cannot say anything definite about the activity progress except that it did not complete. We call this property of an event *atomicity*.

An event has three types of attributes: *time*, *address*, and *information*. A time attribute (also referred to as "time of occurrence") represents in some way the moment at which the result of the associated activity becomes available for use. For example, the value of the computer clock at the moment of the event occurrence describes the time of occurrence according to this particular clock. An address attribute (also referred to as "event source") represents the place where the result of the activity has become available. For example, the IP address of the computer host at which the event occurred, can serve as a value of an address attribute for that event. An information attribute (also referred to as the "effect" of the event) represents the result of the activity. For example, the text of an e-mail message received by someone can serve as a value for the information attribute of the event that represents the successful receiving of that e-mail message by its recipient. When an event occurs, the monitoring system generates an *event report* that contains values for each event attribute. The monitoring system then sends the event report to interested monitors.

We introduce the general term *monitoring report* to denote a status report or an event report.

Relations among events

Analysis of complex activities, such as a synchronous operation invocation, requires the decomposition of each activity into a collection of simpler related activities. For such purposes, a monitoring model may model the *relations* among events.

Relating events allows for reasoning about the distributed computation that produced these events [RST91]. Relating events finds application in, for example, distributed breakpoints, detection of race conditions, and management functions. In load balancing and fault tolerance, for example, analyzing the relation among certain events may reveal poorly performing software components that a management component needs to migrate to another hardware platform in order to reduce system load.

We categorize relations among events in a monitoring model into *temporal* and *causal* relations [HSV99].

Temporal relations

We usually perceive time as an absolute measure of the progress of everyday activities. Temporal relations represent the ability to order events on a linear scale according to their time of occurrence. For example, if we use a clock (assuming we have one) having a certain resolution[2] to measure the time of completion of two activities, we can say one of the following about events e_1 and e_2 that represent these activities: *e_1 occurred earlier than e_2*, *e_2 occurred earlier than e_1*, or e_1 and e_2 (seem to) have occurred *at the same time*. This means that using such a clock we can establish a *temporal order* among all events that we observe. Clocks usually also allow us to measure the distance between event occurrences.

Let E represent the set of all events in an application execution. Let C represent some clock in the every day meaning. For simplicity, we assume that the accuracy of C suffices to measure correctly the time of occurrence of every event in E.

Definition 2-1 Temporal precedence

We define the "temporal precedence" relation $TR_E^C \subset E \times E$ as $TR_E^C = \{ (e_1, e_2)$: e_1 **occurred earlier than** *e_2 according to clock $C\}$.*

Observe that TR_E^C has irreflexive, asymmetric, and transitive properties, hence it defines a strict partial order on E.

Causal relations

The Merriam-Webster's online dictionary defines causality as "the relation between a cause and its effect or between regularly correlated events or phenomena". Hence, a causal relation defines for an event how its occurrence depends on the occurrences of other events.

In an every-day interpretation, the fact that e_1 causes e_2, implicates that e_1 temporally precedes e_2. The classical mechanics of the physical world also considers this intuitive notion of causality, where the cause precedes its effect. This motivates us to make the following definition of a basic property of causality:

Definition 2-2 Basic property of causality

A causal relation implies a temporal precedence between the cause and the effect, such that the cause occurs earlier than its effect.

Observe that the opposite does not always hold: the mere fact that events occurred in some temporal order, does not imply that these events relate causally. We illustrate this by the fallacy "*Post hoc, ergo propter hoc*", which we traditionally interpret as "after this, therefore because of this" [Giere98]. The causality property also gives us the following interesting causal property of temporal precedence:

[2] We define the resolution of a clock as the smallest unit with which the clock updates its value when it advances one step.

Definition 2-3 Causal property of temporal precedence

If e_1 does not temporally precede e_2 (i.e., according to some clock e_2 occurs earlier than or at the same time as e_1) then e_1 could not have caused e_2.

Note that for this property to hold we need to say something about the accuracy of the clock used to determine the temporal relation: for e_1 that have caused e_2, the clock accuracy should suffice to determine that e_1 temporally precedes e_2. We call such a clock *infinitely accurate*.

Given an infinitely accurate clock, the causal property of temporal precedence allows us to use temporal precedence to reason about possible causal relationship between events (in a single application execution) by ruling out the cases in which we consider causal relationship impossible because the effect cannot occur earlier than (or at the same time with) its cause.

We define two types of binary causal relationships: *realized causality* and *potential causality*.

Definition 2-4 Realized causality

We consider a relation $R \subset E \times E$ "realized causality" iff for $\forall$ $(e_1, e_2) \in R$, e_1 ***definitely*** *causes e_2, i.e., e_2 could not have occurred if e_1 hadn't occurred.*

Note that according to the basic property of causality and given an infinitely accurate clock C, a realized causality relation $R \subseteq TR_E^C$.

In general, reasoning about causality with the certitude of realized causality requires knowledge about the rules by which the application operates – i.e., the design of its intended behavior [HSV99]. We provide a second definition with a more loose causality condition, because in general, the monitoring application may not have access to the design of the monitored application.

Definition 2-5 Potential causality

We consider a relation $R \subset E \times E$ "potential causality" iff $\forall$ $(e_1, e_2) \in R$, e_1 ***may*** *have caused e_2.*

Note that we want potential causality to preserve the basic property of causality. Therefore, according to the causal property of temporal precedence and given an infinitely accurate clock C, a potential causality relation $R \subseteq TR_E^C$. Therefore, we consider TR_E^C as the biggest possible potential causality relation for an application execution (i.e., any pairs outside of TR_E^C cannot participate in a potential causality relation according to the causal property of temporal precedence). Figure 2-2 illustrates the relation between potential causality and temporal precedence. Furthermore, a realized causality relation implies a potential causality,

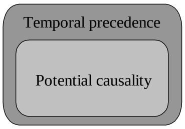


Figure 2-2 The relation between potential causality and temporal precedence

because the definite cause (realized causality) of some event also represents one of all possible candidates for a cause (potential causality) of that event.

2.2.2 Status-based vs. event-based modeling

A status-based model of application execution represents the information associated with entities by abstracting from any particular activities that entities performed to produce that information. In contrast, event-based modeling focuses on representing the actual activities performed by entities, (possibly) abstracting from some of the information aspects produced by activities. Hence, the choice of a modeling technique depends on the aspects of application behavior, on which designers of monitoring systems want to focus.

Monitoring distributed object and component communication constitutes one of the goals of this thesis. During monitoring we require explicit knowledge about the communication-related activities performed by objects and components of the monitored application, such as establishing of a connection between physically remote hosts, marshalling, transmitting and un-marshalling of parameters, and processing of (possible) intermediate error conditions. Therefore we use event-based modeling.

In the next section we introduce an event-based monitoring model of distributed application execution. In Chapter 7, we use this model as a basis for the definition of a monitoring model for object and component communication.

2.2.3 An event-based monitoring model

In this section we present a monitoring model [SchMa94][RaSi96]. In this model, a distributed application consists of entities called *processes* (also called threads, fibres, or light-weight processes) that can communicate with each other. Each process sequentially performs activities. At this point we make no other assumptions about any relations between the activities performed by a process. We model the successful completion of an activity using the event concept defined earlier.

We denote all events in an execution of a DSA as

$$E = E_1 \cup E_2 \cup \ldots \cup E_M,$$

where M represents the total number of processes participating in this execution and

$$E_i = \{e_{i,1}, e_{i,2}, e_{i,3},\ldots\}$$

represents the set of events corresponding to activities performed by process p_i. Since in each process activities occur sequentially, their times of occurrence (according to an infinitely accurate clock) allow total and strict ordering of the corresponding events. Let the binary relation $\rightarrow_i \subset E_i \times E_i$ represent the temporal precedence relation among the events of process p_i. The temporal precedence relation on the set E_i corresponds to the largest potential causality relation on the set E_i.

Processes communicate with each other by using point-to-point *messages*. The communication of a message takes unpredictable finite time to complete. At this point, we make no other assumptions about availability of shared resources, FIFO order of messages, communication infrastructure, or synchronization mechanisms such as CSP-guards [Hoare78]. We define the binary relation $\rightarrow_{msg} \subset E \times E$ such that for every message m sent from process p_i to process p_j there exist events $e_k \in E_i$ corresponding to a send activity, and $e_l \in E_j$ corresponding to a receive activity, for which the following holds:

$$e_k \rightarrow_{msg} e_l$$

Observe that relation $\rightarrow_{msg}$ corresponds to a realized causality relation, because a send event represents the definite cause of a receive event, following the initial assumption that processes communicate via point-to-point messages only.

We define the "causal precedence" binary relation [RaSi96] as the union of the temporal precedence relations among the events of every process and the realized causality relation among events representing message exchanges between processes:

$$\rightarrow \; = \; \left(\bigcup_{i=1}^{M} \rightarrow_i\right) \cup \left(\rightarrow_{msg}\right)$$

The causal precedence relation defines a strict partial order on E (it has the irreflexive, asymmetric, and transitive properties). Defined this way, the causal precedence relation corresponds to the "*happened before*" relation [Lamp78]. Note that we consider the causal precedence relation defines a potential causality relation on the set E.

Using the causal precedence relation we define a *distributed execution* (also referred to as distributed computation) as the structure $DC = (E, \rightarrow)$.

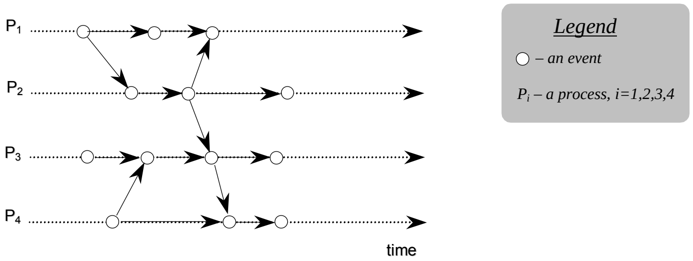


Figure 2-3 A distributed execution

Figure 2-3 shows an example of a distributed execution. The arrows between events represent the causal precedence relation.

Using relation $\rightarrow$ we define the "*concurrency*" relation

$$\| \in E \times E,$$

where

$$e_1 \| e_2 \text{ if and only if } \neg(e_1 \rightarrow e_2) \text{ and } \neg(e_2 \rightarrow e_1).$$

If one can say that two events belong to the concurrency relation we consider these events causally independent.

2.2.4 Using the monitoring model

A monitoring model defines the aspects of application execution about which monitors require information. Designers of the instrumentation use the monitoring model to implement the code for monitoring these aspects. We call this *model-driven instrumentation* [Hof + 94]. During monitoring, the instrumentation performs measurements on the monitored application and as a result generates status and event reports. When the monitor receives these monitoring reports, it analyzes them in order to create an instance of the monitoring model that characterizes one application execution.

In everyday life, we regularly order and plan our activities. We do this using a clock, i.e. by taking a *timestamp*. Hence, we feel used to the notion of accurate global time at our disposal. In distributed systems however, we usually do not have access to a global clock that has sufficient accuracy for the purpose of timestamping. The inaccuracy of the clocks embedded in most widely used affordable computer architectures, may significantly affect our ability to restore the temporal precedence relation among events. To circumvent this problem, we use a special mechanism called *logical time* in order to reason about order among event in terms of potential causality without the need of an infinitely accurate global clock.

Logical time

We define logical time as a mechanism that allows ordering among the events of a distributed computation [RaSi96]. We use a system of *logical clocks* to measure the pace of logical time. We confine the discussion on logical clocks to the monitoring model introduced in the previous section. The system of logical clocks generates timestamps, which we assign to events. A timestamp represents a concrete (as opposed to abstract) data structure with values, which a program can process, e.g., serialize and transmit over a communication infrastructure.

We define the relation that assigns timestamps to events as

$$LC{:}E \mapsto T,$$

where T represents the logical time domain. LC maps a timestamp $LC(e) \in T$ to every event $e \in E$.

We define the partial order relation $<$ in T, such that

$$e_1 \rightarrow e_2 \Rightarrow LC(e_1) < LC(e_2).$$

In other words, the relation LC preserves the basic property of causality, i.e., the potential cause occurs earlier than its effect. We call this the *clock consistency* condition.

The following property

$$e_1 \rightarrow e_2 \Leftrightarrow LC(e_1) < LC(e_2),$$

makes a logical clock system *strongly consistent*. Strong consistency effectively means that an external observer, who uses a logical clock to assign to events timestamps in the time domain T, can restore the causal precedence between events just by analyzing their corresponding timestamps. We consider strong consistency of the logical clock system crucial to the goals of this thesis, because (provided we have a system of logical clocks) it allows monitors to determine the order of events in an application execution from event reports.

A system of logical clocks has two major properties: *topology* and *metrication* [Smith80]. The topology represents the structural framework that timestamps impose on events. For example, we can use a directed a-cyclic graph to represent a partial order topology. In this graph, nodes represent events and an arrow between two nodes represents a causal precedence relation between the events corresponding to these two nodes.

The metrication represents the mechanisms used to measure the value of a logical clock. The definition of the metrication for a system of logical clocks requires two steps: (a) determining the data structure of the logical clock, e.g., the data structure of a timestamp, and (b) defining a set of rules for generating timestamps and updating the data structures within the system of logical clocks to ensure the strong consistency condition.

We use the following general data structure for the system of logical clocks: each process p_i maintains a local logical clock lc_i, which we use to record the progress of logical time only within process p_i, and a global logical clock gc_i, which records the local view of p_i on the progress of logical time in the whole system [RaSi96]. We use the values of lc_i and gc_i to generate timestamps for events occurring in process p_i.

A system of logical clocks uses the following two update rules:

- *R1* governs how a process updates its local logical clock;
- *R2* governs how a process updates its global logical clock.

Existing logical clock systems

Based on metrication characteristics we classify existing logical clock systems into three types: scalar, vector and matrix.

Lamport developed the *scalar* clock system [Lamp78]. This system uses timestamps in the domain of non-negative integers. A total order represents the topology imposed on the timestamps generated from this clock system. In this system, each process p_i maintains a single counter that represents both lc_i and gc_i. The rule *R1* increments the counter each time a new event occurs in the process. The rule *R2* updates the counter with information about the counter of some process p_j each time a message arrives from that process. This system does not satisfy the strong consistency condition, thus we cannot use it to achieve our goals.

Schwarz and Mattern [SchMa94] have developed a system of *vector* clocks. This system uses timestamps in the domain of M-dimensional vectors of non-negative integers, where M represents the number of processes in the application execution. A partial order represents the topology imposed on the timestamps generated by this clock system. In this system, each process p_i maintains a vector gc_i that keeps information about the local progress of the other processes, including its own progress at the i-th position: $gc_i[i] = lc_i$. Rule *R1* increments lc_i each time a new event occurs in the process. Rule *R2* updates gc_i with the information from the gc_j of some process p_j each time a message arrives from that process. Vector clocks satisfy the strong consistency property.

Fischer and Michael [FiMi82] have proposed a system of *matrix* clocks. This system uses timestamps in the domain of MxM matrices of non-negative integers. A partial order represents the topology imposed on the timestamps generated by this clock system. In this system, each process p_i maintains a matrix gc_i that keeps information about what all other processes know about each other's progresses (hence the matrix), including its own progress at the i-th column of the matrix, which is the lc_i. Matrix clocks update their value according to rules similar to the vector clocks and they satisfy the strong consistency property.

We chose vector clocks to provide logical time for our monitoring system, because a system of vector clocks provides strong consistency. We disregard matrix clocks, because although they satisfy the strong consistency condition too, we do not need the additional information matrix clocks provide.

2.2.5 The vector clock system

In this section we introduce the metrication of a vector clock system, which consists of a data structure and updating rules.

We define the global logical clock gc_i as a vector of non-negative integers with $i \in [1,\ldots, M]$ where M represents the number of processes. For a gc_i, $gc_i[i]$ represents the progress of process p_i (hence $gc_i[i]$ represents the local logical clock lc_i), and $gc_i[j]$, $\forall\, j \neq i$ represents the knowledge of p_i about the local progress of p_j. The vector gc_i constitutes the view of p_i on logical time in the system.

The vector clock system uses the following *R1* and *R2* rules:

– *R1*. When an event occurs in process p_i, the process updates its local logical time in the following way:

$$gc_i[i] := gc_i[i] + 1$$

Note that initially $gc_i[j] = 0$, for $j \in [1,\ldots, M]$

– *R2*. For each message m that process p_i sends to another process, p_i attaches (called *piggybacking*) to the message as a context the value of its own global logical clock at the moment of sending. Upon receiving of a message m with a message context gc, p_i performs the following steps:

a) p_i updates its global logical time in the following way:

$$gc_i[j] := \max(gc_i[j], gc[j]), \text{ where } j \in [1,\ldots, M]$$

b) p_i performs *R1*;

c) the receiving process handles the message m.

For an every event, we define a timestamp of the vector clock system as the value of the global logical clock gc of the process in which an event occurs, taken at the moment of the event occurrence. Therefore, a timestamp of a vector clock system corresponds to a vector, hence we call it a *vector timestamp*.

In order to reason about order among events, we need to compare their corresponding vector timestamps. We define the following three binary relations to establish order among vector timestamps:

$$v_1 \leq v_2 \Leftrightarrow \forall\, x: v_1[x] \leq v_2[x]$$

$$v_1 < v_2 \Leftrightarrow v_1 \leq v_2 \text{ and } \exists\, x: v_1[x] < v_2[x]$$

$$v_1 \parallel v_2 \Leftrightarrow \neg(v_1 < v_2) \text{ and } \neg(v_2 < v_1)$$

Given these relations among vector timestamps and the rules for advancing the vector clock system, the following property holds [RaSi96]: if two events e_1 and e_2 have timestamps v_1 and v_2 respectively, then

$$e_1 \rightarrow e_2 \Leftrightarrow v_1 < v_2$$
$$e_1 \parallel e_2 \Leftrightarrow v_1 \parallel v_2$$

Thus, an isomorphism exists between the set of partially ordered events of a distributed execution and their timestamps. This also means that the vector clock system satisfies the strong consistency condition.

2.2.6 Implementation issues of vector clocks

The rules for advancing vector clocks require certain functionality from the programming environment used for implementing an instrumentation. These requirements may influence the choice of the monitoring system designer for an implementation technology. We distinguish the following requirements:

- Vector clocks require from the programming environment a means to associate data structures (context) to processes (or threads) in the monitored application. This would make sure that designers can properly implement rule *R1* for every process in the system;
- Vector clocks require from the programming environment a means to associate context with messages between processes. This would make sure that designers can properly implement rule *R2* by means of piggybacking necessary clock information in messages exchanged between processes.

The size of the vector clock has impact on memory usage its maintenance, and to the size of the structure piggybacked with each message. The size of the piggybacked structure depends on the number of processes in the distributed system. In the case the monitored application uses a large number of processes, this dependence may lead to an unbounded growth in size of information piggybacked with each message.

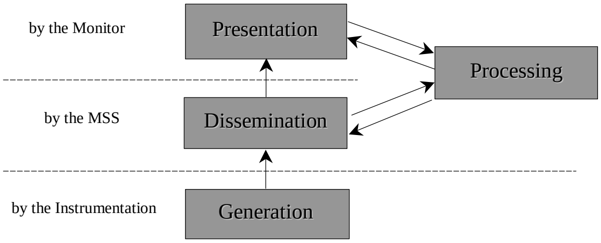
The search for an efficient implementation of the vector clock system produced several results, such as the differential technique [SiKs92], direct dependency technique [FoZw90], adaptive technique [JaJou94], and others. All these techniques consider the full number of processes in the distributed computation. They achieve a reduction of the size of the vector clock structures at the cost of additional processing. Existing work on this topic indicates that when we want to consider all processes in the system, it seems we cannot have a strongly consistent logical clock representation more compact than vector time [SchMa94]. Others approached the efficiency problem of the clock data structure by reducing the number of processes considered in a vector clock system (and hence the size of the vector) using clustering of the processes in the distributed application into domains [Sum92] [Ward01]. This means that these approaches disregard some of the causal relationships among events belonging to different clusters. We consider this assumption too severe, because it will not allow us to reason about distributed communication among objects from different clusters, and therefore we do not consider these approaches.

Furthermore, we consider improving the efficiency of the vector clock implementation out of the scope of this thesis. In Chapter 7 we shall provide a basic implementation derived directly from the definition of a vector clock system in this chapter. Designers can replace the implementation with a more efficient strongly consistent implementation provided that it considers all processes in the system (no clustering).

2.3 Monitoring activities

We discuss common monitoring activities using a functional model for monitoring (Figure 2-4).

Figure 2-4
Functional model for monitoring



This model consists of four basic groups of monitoring activities [Sam95]:

- *Generation*. An instrumentation measures and packages monitoring data to make it available for the MSS;
- *Dissemination*. The MSS collects monitoring data from an instrumentation and delivers it to interested monitors;
- *Processing*. The MSS analyzes monitoring data coming from the instrumentation in order to convert it to a format and level of detail appropriate for monitors. A monitor analyzes monitoring data to extract information for the monitoring application;
- *Presentation*. A monitor offers a view on monitoring data from the MSS appropriate for the monitoring application.

If we consider how the monitoring system handles a single monitoring report, the monitoring system typically performs monitoring activities in certain order. Generation comes first because it provides the monitoring data. Then the MSS disseminates monitoring data to a monitor. During dissemination, the MSS may process monitoring data before delivering it to a monitor. In the end, the monitor presents monitoring data to a monitoring application. During presentation, the monitor may also need to analyze and process the monitoring data in order to extract information necessary for the presentation activities. Note designers may prefer other scenarios in which a monitoring system may realize only a limited selection of the listed monitoring activities. For example, the MSS may store monitoring data in some place for future reference right after generation. Monitors can later use this monitoring data without any additional processing from the MSS.

In general the monitoring application may have requirements on the timely delivery of monitoring data. Based on these requirements, we distinguish two types of monitoring: *online* and *offline*. Online monitoring allows observation and potentially control of applications at runtime [Rack01]. In case of online monitoring, a monitoring application poses real-time constraints on the overall time it takes to generate, process, disseminate and present monitoring data. In case of offline monitoring, a monitoring application poses no such time constraints. Hence, the monitoring application may obtain monitoring data at an arbitrary time after its generation by the instrumentation.

In the next four sections we discuss in detail each of the groups of monitoring activities.

2.4 Generation activities

The instrumentation generates monitoring data by *measuring* certain aspects of the monitored application and *packaging* the resulting values into monitoring data.

Measuring may require access to the internal mechanisms of the monitored application. For software monitoring, instrumentation designers enable this access by installing special programs called software *sensors* (also called *probes*) in the application or its environment [Rackl01]. Packaging involves the formatting of measured values into monitoring data.

We distinguish between the generation of the following types of monitoring data: status reports, event reports, and monitoring traces.

2.4.1 Status report generation

Status report generation consists of measuring variables in a status vector and packaging these values into a status report.

The instrumentation generates a status report in three ways: *event-driven*, *time-driven*, and *on-demand* [Sam95][Shaer98]. In case of event-driven generation, the instrumentation measures a status vector because some event occurs in the monitored application. In case of time-driven generation, the instrumentation measures the status vector periodically. In case of on-demand generation, the instrumentation measures the status vector upon a request from the MSS.

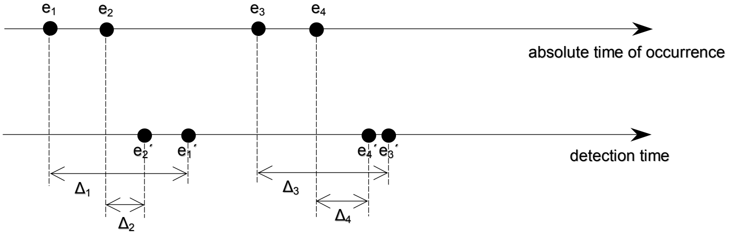
2.4.2 Event report generation

Event report generation consists of the detection of an event, measuring the values of event attributes, and packaging these values into an event report.

The instrumentation detects events as a result of the execution of some software sensors. We define *event detection* time, as the moment the instrumentation learns about the occurrence of an event. We define *detection delay*, as the distance between the time of event detection and the actual time of event occurrence. In software monitoring, the execution of the monitoring software always introduces some detection delay resulting from, e.g., executing sensor code.

Detection delay may vary between individual events, e.g., due to the varying system load that may influence the speed at which sensor code executes in a distributed environment, or the use of incorrect computer clocks for measuring the time of event occurrence. Varying detection delays give rise to the following type of problems (Figure 2-5).

Figure 2-5
Inaccurate order of events due to variable delay



The four primitive events, e_1, e_2, e_3, and e_4 occur on the absolute time scale in a certain order. According to the corresponding detection times (Δ_1, Δ_2, Δ_3, Δ_4) however, events may appear to have occurred in a different order, as shown on the lower scale of time.

We consider the problem described above as one of the reasons to use logical clocks for event timestamping instead of computer clocks. Logical clocks allow for measuring the relative order (compared to the absolute order) between events, without the need to measure distances between absolute moments in time. Hence, by using logical clocks we can ignore detection delay.

2.4.3 Monitoring trace generation

A *monitoring trace* (or simply “trace”) represents a collection of monitoring reports generated over some period of time [Sam95]. Trace generation includes the caching of monitoring reports at the instrumentation, and packaging monitoring reports into a trace.

We distinguish the following two applications of traces: to reduce the size of monitoring data, and to minimize the use of the communication infrastructure for monitoring related traffic. The instrumentation achieves size reduction by removing information common to all reports in the trace and providing this information only once per trace.

A monitoring trace may suggest order among the individual reports in the trace. This order may represent, for example, the order in which the instrumentation generates the reports.

We distinguish two types of monitoring traces: *complete* and *segmented*. A complete trace contains all monitoring reports from one uninterrupted use of the monitoring system, which we call a *monitoring session*. A segmented trace contains only the monitoring reports generated during a time period in a monitoring session.

In a distributed deployment of the monitored application, the instrumentation may consist of physically distributed parts. As a consequence, each instrumentation part can only provide segmented traces to the MSS. Therefore, in a distributed environment, the MSS collects several segmented traces rather than one complete trace.

2.5 Processing activities

The MSS processes monitoring data generated by the instrumentation to make sure that monitors receive monitoring data they require. We distinguish the following types of processing activities: event correlation, filtering, validation, and trace manipulation.

2.5.1 Event correlation

We discussed in section 2.2.1 that event relations provides means for analysis of the behavior of distributed applications. We define *event correlation* as the process of determining the relation among two or more events from the monitored application behavior. We consider two basic forms of event correlation in the MSS: ordering of events, and detection of composite events.

Ordering of events

The MSS may order event reports according to the time of occurrence of the corresponding events. In this way monitors do not need to perform the ordering themselves. This (potentially) reduces the monitor's computational effort for analyzing monitoring data by shifting ordering responsibility to the MSS, and the size of monitoring data by discarding additional ordering information (such as the information required to maintain vector clocks consistent) before sending monitoring data to the monitor.

Detection of composite events

We define a *composite event* as the composition (also called *combination*) of several events (including other composite events). With each composite event we associate a *detection condition*, which determines when the composite event occurs. For example, one may specify a composite event $ce = (e_1 \wedge e_2) \vee e_3$, using the logical conjunction and disjunction operators. The detection condition for ce states that ce occurs when both events e_1 and e_2 occur or when e_3 occurs.

Monitors can use these composite events to coordinate various tasks that depend on the completion of more than one activity in the monitored application.

Some monitoring systems [Hold89][Sam95][Shaer98] use descriptive languages to allow users to define a specification of an arbitrary composite event during runtime. The monitoring system processes event specifications in order to detect composite events.

2.5.2 Filtering

The MSS performs *filtering* to determine the relevancy of monitoring data. A *filter* specifies the monitoring data required by a monitor. We assume that monitors may change their requirements during runtime. During filtering, the MSS processes monitoring data in such a way that a monitor receives only monitoring data that satisfies the requirements in its filter.

Filtering may result in:

- Discarding of monitoring reports. The MSS may discard generated monitoring reports that does not match any monitor's requirements;

- Reduction of content. The MSS may strip monitoring reports from some of their attributes, when monitors do not require all the information contained in generated monitoring reports.

Filtering helps to alleviate monitoring overhead. For example, the processing of a filter by the MSS may result in the reconfiguration of the instrumentation, so that it does not generate unnecessary monitoring data in the first place. Hence filtering (potentially) reduces the overhead of the instrumentation in terms of CPU and memory, and reduction of the overhead of the MSS in terms of communication bandwidth.

2.5.3 Validation

The MSS performs validation to ensure *validity* of the monitoring data delivered to monitors. We define validity as the correspondence of the generated monitoring data with the monitoring data received by monitors. The MSS may encounter validity problems such as, wrong order of monitoring data (due to, e.g., communication delays), corruption in monitoring data (due to, e.g., transmission errors, faulty hardware, or a security breach), missing or duplicated monitoring data (due to, e.g., use of unreliable communication infrastructure, or partial failures in the distributed system).

Validation activities deal with detection of validity, restoring validity, and security.

Detection of validity

The MSS detects validity of monitoring data according to some *criteria*. For example, using an SHA-1 [SHA95] checksum on a monitoring report allows the MSS to detect at various stages of the dissemination of the report whether the current report differs in some way from the originally generated one. Calculating a new SHA-1 checksum and comparing it with the checksum of the original would show any corruption of the monitoring data (or of the checksum itself).

Restoring validity

When the MSS detects invalid or missing monitoring data, it can discard the data, (if possible) may request the data again from the instrumentation, or may try to restore the validity of the data. For example, an MSS may have the validity requirement that the order of generating monitoring reports from the instrumentation corresponds to the order of deliverying the reports at monitors. When the MSS detects reports in the wrong order (e.g., due to communication delays), the MSS reorders the monitoring reports before sending them to monitors.

Security issues

Validation may also have a security aspect. In some cases, the MSS may have to determine authenticity of origin and integrity of transmitted monitoring data. To do this, the MSS can use techniques, such as digital signatures [GeRo97], to sign monitoring data.

2.5.4 Trace manipulation

Trace manipulation consists of merging and splitting monitoring traces.

Merging

The MSS *merges* segmented traces to construct a new trace. The merged trace represents a broader collection of events in the monitored application than each of the original segmented traces. The MSS can build a complete trace of application behavior during one monitoring session by merging all segmented traces. When monitoring traces imply a certain order among the reports, an important issue during merging becomes the preservation of that order in the resulting trace.

Splitting

The MSS *splits* a trace into two or more segmented traces to reduce the amount of reports in one trace. Each of the resulting traces contains a subset of the reports from the original trace. The MSS may perform trace splitting in combination with filtering to produce different segmented traces to satisfy several different filters.

2.6 Dissemination activities

The MSS performs dissemination activities to make sure that the required monitoring data reaches monitors on time. We distinguish two types of dissemination activities: collecting monitoring data from the instrumentation, and delivery of monitoring data to monitors.

We discuss each of the activities for the cases of online and offline monitoring (section 2.3.)

2.6.1 Collecting monitoring data

The MSS *collects* monitoring data generated by the instrumentation, in order to process it (we discussed processing in section 2.5) and deliver it to monitors. We consider two major issues regarding collection: storage and generation configuration.

Storage

In case of online monitoring, the instrumentation may generate monitoring data at rates and with size, which monitors cannot deal with. In such cases, the MSS can either store the data or discard it. Providing an intermediate storage for monitoring data, allows monitors to receive monitoring data at the rate and size they can deal with.

In case of offline monitoring, the MSS provides persistent storage for monitoring data, such as a database. Monitors may access the monitored data at any time, even after the monitored application has completed its execution.

Generation configuration

Ideally, during online monitoring, the instrumentation should not generate monitoring data that monitors do not require. Nevertheless, monitor's may change their monitoring data requirements during runtime, and as a result the instrumentation may end up sending to the MSS monitoring data that no monitor requires. The MSS may discard this data to free allocated resources, however, collecting the data from the instrumentation already occupies resources in the MSS. To deal with this, the MSS can dynamically *configure* the instrumentation, so it only generates requested monitoring data. The MSS may perform such (re-)configuration during runtime by switching sensors in the instrumentation on and off.

2.6.2 Delivery of monitoring data

The MSS *delivers* monitoring data to monitors. In order to receive monitoring data, a monitor needs to register its presence to the MSS. We choose not to discuss any registration details, such as authentication, establishing of trusted connections, etc. Instead, in this section we concentrate on the methods for delivery of monitoring data. We distinguish two main delivery methods: subscription-based and request/response-based.

Subscription-based delivery

Subscription-based delivery starts when a monitor submits to the MSS a *specification of interest*. A specification of interest contains the monitor's requirements for monitoring data. Compared to the monitoring model, which defines all application aspects that a monitor may observe, the specification of interest defines additional constraints on the monitoring data. This allows the MSS to filter out any irrelevant monitoring data, and to pass to the monitor only relevant monitoring data. The specification of interest may represent a filter (discussed in section 2.5.2), or a specification of one or more composite events (discussed in section 2.5.1).

After a monitor subscribes for monitoring data, the MSS starts to *notify* it about new (and relevant) monitoring data. The MSS has the initiative for sending monitoring data and thus has the control over the timeliness of the delivery. This makes notifications suitable for online monitoring, because the MSS can notify monitors about monitoring data as soon as it becomes available.

Request/response-based delivery

In request/response-based delivery, a monitor submits a *request* for monitoring data to the MSS, and waits for a *response* from the MSS within certain real-time constraints. Along with the request, the monitor may submit a *selection criteria*. The request instructs the MSS to deliver as a response to this request, monitoring data with features described in the selection criteria. The selection criteria may represent a concrete instruction for measurement that the MSS has to delegate to the instrumentation.

In case of online monitoring, the MSS processes a request in the following way: if the instrumentation has already generated the requested monitored data, the MSS immediately sends the data to the monitor as a response to that request; otherwise, the MSS forwards the request to the instrumentation so that it generates a status report with the requested information.

In case of offline monitoring, the MSS stores the generated monitoring data for future use. At a certain moment, the monitor passes a selection criteria that the MSS uses to select some monitoring data. Based on the nature of the selection criteria, we distinguish two types of request/response-based delivery for offline monitoring: *data retrieval* or *information retrieval*. We consider the following main differences between the two types of retrieval [Weide01]:

- Information representation. Data retrieval works with the well-defined logical structure of the monitoring data. In contrast, information retrieval works on unstructured monitoring data;
- Method of selection. Data retrieval uses a direct method of selection based on facts about the values organized by the structure of the monitoring data. In contrast, information retrieval uses a method of selection that satisfies the needs of the monitor with “a degree of success”.

We choose to consider only monitoring systems that use a well-defined logical structure of their monitoring data based on a detailed monitoring model. The MSS uses data retrieval in which the designer of the system determines the rules for matching selection criteria to monitoring data during the design of the monitoring system. A typical example for data retrieval constitutes the use of database management systems for persistent storage of monitoring data in the MSS. In such cases, the monitor uses, e.g., Structured Query Language (SQL) [SQL99], to define the selection criteria and issue a request to the MSS. The result of the request yields monitoring data with structure and values precisely matching the selection criteria.

2.7 Presentation activities

A monitor performs presentation activities in order to convey information about the monitored application behavior to the monitoring application. The monitor may present this information to a software program or to a human user. The monitor presents information to a software program by selecting the proper data formats and encoding. In this section we concentrate on the ways a monitor presents information to a human user.

Presentation to a human user represents the process of making information accessible through the human senses. Human comprehension has certain limitations: the maximum number of chunks of information an individual can simultaneously comprehend, roughly equals seven, plus minus two [Miller56]. Hence we can formulate the fundamental dilemma of presentation: how can designers build monitors that efficiently present information to their human users?

Contemporary computers mainly generate sounds and graphics to convey information to their users.

Graphical information

Humans respond well to graphical information [Tufte83]. We find graphics captivating and appealing, if well designed. A visual representation in the form of graphics, can communicate information in a rapid and efficient way. Therefore, visualization represents an interface between two powerful information processing systems, the human mind and the modern computer [GEC98]. Static 2D/3D computer graphics have the following representational qualities [Bertin99]:

- Presence. Characterizes when the observer can recognize the presence or the absence of something on the display;
- Quantity. Reflects the observers opinion on the size of an object, or the number of objects;
- Distinction. Relates to the observer's ability to discover differences between individual things;
- Order. Reflects the observer's opinion on the order of things.

Sound information

Sound, and in particular musical sound, has its own representational qualities:

- Intensity. Characterizes the loudness of the sound. Louder sounds can relate to presence and distinction;
- Pitch. Characterizes the frequency of the vibrating body producing the sound. For example, the ear distinguishes well high pitch sounds from low pitch sounds, which relates to distinction;
- Timbre. We determine the timbre of a tone by the number of the subsidiary overtones we hear and their relative intensity. For example, we can use timbre to represent quantity, as well as presence.

The Network Auralizer PeeP [GiCo00] gives an example of the experiments for representing monitored information to human users using sound. This system allows a network administrator to associate certain events with certain sounds. These sounds can represent, for example, background alerts. Chirps of a cricket may represent a normally loaded network, while a chorus of an increasing number of crickets and other animals can represent a highly loaded network. Still, at the current state of computer technology, sound does not convey information in a manner as versatile and as expressive as graphical representation. We consider this one of the reasons why graphical presentation dominates presentation of monitoring data.

We limit this discussion to the following graphical presentation activities: scaling time, animation and replay, multiple views, and interactive display.

2.7.2 Scaling time

We often consider time the most important dimension in a diagram, because it suggests temporal order among its elements. A diagram can demonstrate temporal order between two objects in an absolute or a relative manner. Relative order focuses on the order as a precedence of one time value to another, while absolute order includes information about the position of a moment according to an absolute time scale. For example, a UML Message Sequence Diagram [UML1.4] shows relative order of message passing between objects, while performance diagrams in general, provide discrete units of time, which on a global scale allow for measuring distance between two particular measurements.

The scale of time presents an important issue in absolute time order. When working with large periods of time in a diagram, users can adjust the time unit for convenient viewing.

2.7.3 Animation and replay

Monitors use animation and replay presentation activities to visualize the behavior of the monitored application.

Animation

The monitor may use animation to convey information about changes in the state of displayed objects. Moving, adding, removing, and coloring of graphical objects on the display presents the behavior of the monitored application. For example, adding and removing graphical objects to and from a diagram respectively can represent the creation or destruction of application objects. In a similar way, the appearance of an arrow between two graphical objects on a diagram for a short time can represent message passing between application components.

An important issue during animation constitutes the time left to viewers to comprehend the information, because it takes about five seconds for a human to accept a new chunk of information [Simon82].

Replay

Replay stands for a special animation technique in which the user can control the animation [Bates85], e.g., rewind, suspend/resume playing, as many times as necessary. For example, during debugging, the monitor may introduce events from the execution of a distributed application on the display in a particular order, using a time scale convenient for the user, or using an interactive step-by-step replay allowing the user to determine when to proceed with processing a new portion of monitoring data.

2.7.4 Multiple views

We consider abstraction a powerful technique for dealing with complexity. We choose to ignore inessential details of a complex object, dealing instead with a generalized, idealized model of that object [Shaw81].

A *view* represents an abstraction visually. The monitor can use an abstraction to deal with high volume of information in a diagram. When a diagram shows too many objects, a selection of fewer abstract objects can represent a large groups of objects with common characteristics.

Monitors can present various views of the same monitoring data. Users can use these views to compare or study monitoring data from different angles.

2.7.5 Interactive display

A monitor may offer to its users interactive features. We consider two major categories of user interaction: *navigation* and *control*.

Navigation

The graphical user interface presents a basic form of navigation, as for example in window-based systems: opening, closing, moving and scrolling of windows. Furthermore, monitors that use multiple views can provide to their users possibilities to navigate among different views.

We consider the navigation of the time dimension another important navigation aspect for monitors that present animation and replay to their users. This kind of monitors should allow fine control to the time dimension such as positioning and choice of time scale (zooming the resolution in or out).

Control

Some monitors allow manipulation of the monitored application based on the monitoring data. Application steering presents an example [RaLe97], in which the user views and steers system execution by sending commands through the graphical user interface. Users may also want to control the monitoring software in terms of online configuration, as for example, changing the granularity of monitoring data, or enabling and disabling various sensors within the application instrumentation.

2.8 Performance of monitoring systems

We distinguish two main performance issues in a monitoring system: monitoring overhead and information consistency.

2.8.1 Monitoring overhead

With overhead we refer to the impact of the monitoring system on the execution of the monitored application. High overhead may compromise the satisfaction of users of the monitored application, which also (indirectly) influences the satisfaction of the users of the monitoring application.

A monitoring system produces overhead as a result of two factors: *intrusion* and *resource sharing*.

Intrusion

We consider intrusion the additional delay that sensor execution introduces in the original application behavior. Intrusion may lead to differences between the monitored application behavior and the behavior of the original uninstrumented application.

Resource sharing

With resource sharing we refer to the effect of the combined execution of the monitored application and the components of the monitoring system, on the resource allocations in a distributed environment. Without a strict mechanism for scheduling the resources of the distributed environment, the monitoring system and the monitored application may compete for limited resources. For example, a non-real-time system, such as the Windows operating system, cannot provide guarantees that the monitoring system would not “steal” extra CPU time for its data processing purposes that the monitored application may need to execute its application logic correctly.

Undesirable effects of overhead

We distinguish two undesirable effects of overhead on the monitored application: change in behavior and unacceptable response time.

In the first case, the delay introduced by the execution of sensors may become so great that some application service may not complete on time. As a result, the application behavior of the instrumented monitored application may become logically different than its original behavior in such a way that the application cannot fulfill its purpose anymore, failing the expectations of its users.

In the second case, the user response of the monitored application becomes so slow (due to intrusion or resource sharing or both) that users may find the application practically unusable.

2.8.2 Information consistency

Information consistency relates to how true monitoring data represents the behavior of the monitored application to the observer. For example, an implementation of the instrumentation that uses the internal computer clock to timestamp events cannot guarantee a consistent view on application behavior in a distributed system, because different computer clocks may cause events to appear to the observer in a different order than their actual order of occurrence.

2.8.3 A method for performance assessment

In this section we present a method for assessing the performance of the monitoring system with respect to monitoring overhead and information consistency. Helmbold and Bryan [HeBr89] originally proposed this method, and Logean [Logean00] extended it to the form we present here. This method uses a model of distributed applications identical to the one presented in section 2.2.3. We use this method in Chapter 8 to evaluate the performance of our monitoring system.

We view the behavior of a monitored application from three perspectives: the behavior of the unmodified and unmonitored original application, the behavior of the instrumented and monitored application, and the observed behavior presented by a monitor (Figure 2-6). Note that as a consequence of monitoring overhead, these three behaviors may differ from each other. We shall not discuss the behavior of an unmonitored instrumented application, although inactive (turned off) instrumentation may still have some impact on the monitored application behavior.

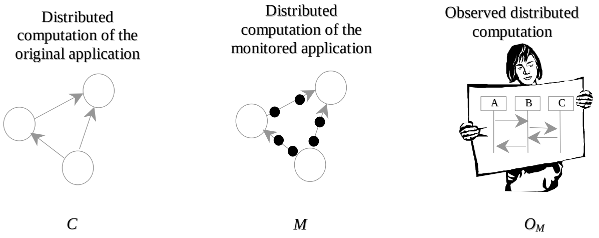

Distributed computation of the original application
Distributed computation of the monitored application
Observed distributed computation
A
B
C
C
M
O_M

Figure 2-6 Three types of behavior

We denote with C a particular distributed computation of the original application, with M a particular distributed computation of the instrumented application, and with O_M an observation of the distributed computation M. We define C, M and O_M in the following way:

$$C = (E_C, R_C)$$
$$M = (E_M, R_M)$$
$$O_M = (E_{O_M}, R_{O_M}),$$

where E_C, E_M, E_{O_M} represent the sets of events in the corresponding computations, and R_C, R_M, R_{O_M} represent the causal precedence order relations on the corresponding sets of events.

We distinguish two sets of events in E_M: $E_{M,M}$ represents the events produced by the monitored process itself (i.e., they do not have corresponding events in E_C), while $E_{M,C}$ represents the events in M that correspond to events in the original behavior C. We define:

$$E_M = E_{M,C} \cup E_{M,M} \text{ and } E_{M,C} \cap E_{M,M} = \varnothing.$$

Furthermore, we define $R_{M,C} = \{(e_1, e_2) : (e_1, e_2) \in R_M \wedge e_1, e_2 \in E_{M,C}\}$, as the partial order relation on $E_{M,C}$.

Based on the three different perspectives on application behavior, we define three properties of a monitoring system: *non-interference*, *accuracy*, and *correctness*:

- Non-interference characterizes the similarity between a distributed computation of the original application and a distributed computation of the monitored application. A non-interfering monitoring system allows the monitored application to do the same things as the original (uninstrumented) application;
- Accuracy characterizes the similarity between a distributed computation of the monitored application and an observed distributed computation. An accurate monitoring system presents to its users a precise view on the monitored behavior;
- Correctness characterizes the similarity between a distributed computation of the original application and an observed distributed

computation. A correct monitoring system produces a view that can reflect each of the computations possible in the original application.

Figure 2-7 Three properties

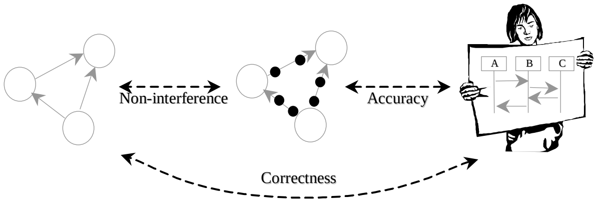


Figure 2-7 [Logean00] illustrates the properties of a monitoring system. We define three degrees of each property: *total*, *strong* and *minimal*, in the following way:

$$\text{non-interference}\begin{cases}\text{total} & E_C = E_{M,C} \text{ and } R_C = R_{M,C}\\ \text{strong} & E_C = E_{M,C} \text{ and } R_C \subset R_{M,C}\\ \text{minimal} & E_C = E_{M,C}\end{cases}$$

$$\text{accuracy}\begin{cases}\text{total} & E_{O_M} = E_{M,C} \text{ and } R_{M,C} = R_{O_M}\\ \text{strong} & E_{O_M} = E_{M,C} \text{ and } R_{M,C} \subset R_{O_M}\\ \text{minimal} & E_{O_M} = E_{M,C}\end{cases}$$

$$\text{correctness}\begin{cases}\text{total} & E_{O_M} = E_C \text{ and } R_C = R_{O_M}\\ \text{strong} & E_{O_M} = E_C \text{ and } R_C \subset R_{O_M}\\ \text{minimal} & E_{O_M} = E_C\end{cases}$$

Total non-interference of a monitoring system means that the monitored distributed computation preserves both the events from the original distributed computation and their order. Strong non-interference means that the monitored computation preserves the events and the order, but that it contains some additional relations as a result of the overhead and additional synchronization imposed on the monitored application by the monitoring system. Minimal non-interference preserves only the events. In this category we also put a monitoring system that preserves only some relations among the events of the original distributed computation.

Total accuracy of a monitoring system means that the observed distributed computation preserves the events of the monitored distributed computation as well as their order. Strong accuracy means that the observed distributed computation preserves the events and their order but that it contains some additional relations among events introduced during monitoring. Minimal accuracy preserves only the events.

Total correctness states that the observed computation completely represents the original computation. Strong correctness implies that the observed computation preserves all events and their order, but that it contains some additional relations among events introduced during monitoring. Minimal correctness preserves only the events.

Note that one can easily see that strong non-interference and strong accuracy imply strong correctness, and total non-interference and total accuracy imply total correctness.

In general, we consider monitoring systems that do not fulfill all minimal properties [Logean00][HeBr89] as *useless*. Monitoring systems that employ vector clocks provide causal precedence relation among the events in a distributed computation. The strong consistency of causal precedence relation allows for the accurate restoring of the relation among events at the observer from their timestamps. Such monitoring systems automatically have total accuracy.

Chapter 3

Overview of object and component middleware

This chapter provides an overview of object and component middleware. In this chapter we discuss middleware terminology and the related concepts that we use throughout this thesis.

We start by introducing object orientation. We discuss communication middleware and object oriented technologies, and how they converged to object middleware. We discuss component-based middleware and its relation to object middleware. At the end we discuss monitoring capabilities in object and component middleware. We close the chapter with conclusions.

3.1 Object orientation

Designers model a complex software system by decomposing it into smaller interrelated parts, each of which they further refine independently from the other parts. In such a system decomposition, designers need then comprehend fewer system parts at once, this way operating within the capacity of human cognition [Parnas85]. An Object-Oriented (OO) decomposition defines a view of a software system as a set of *objects*. An object represents a tangible software entity that exhibits some well-defined autonomous behavior [Booch91]. Objects collaborate together to perform the higher level behavior of the software system.

3.1.1 The object concept

An object has *state*, *identity* and *behavior* [Booch91]. The state of an object consists of the object’s properties. For example, an object that represents a dog may have a property “color”. Object identity represents a special object property that distinguishes it from all other objects [KhoCo86]. The behavior of an object consists of the activities an object can perform. Referring to the same example, the dog object may “bark”, “sit”, etc. Furthermore, an object can send and receive messages to and from other objects respectively, in order to collaborate with these objects.

3.1.2 The class concept

According to the Merriam-Webster Online Dictionary, a class represents “a group, set, or kind sharing common attributes” [M-W]. In object orientation, the class concept represents a group of objects with common characteristics. These characteristics may include common properties (state) and common activities (behavior). Classes and objects differ substantially. While an object represents a concrete entity that exists in space and time, a class represents only the abstract essence of an object. In other words, a single object represents one class *instance*.

Classes suggest a class structure in an object-oriented design, in which the designer reuses functionality, i.e., the designer groups system parts (objects) with similar functionality by classifying these parts into groups of related abstractions (classes) [Gold84]. By realizing the abstraction (the class), a designer can build as many instances (objects) of these abstractions, without the need to realize all aspects of the state and behavior of each instance independently.

A class has an *interface* and an *implementation*. The interface of a class defines a common view on the state and behavior of all objects of that class. An interface emphasizes on the abstraction a class maintains to characterize its objects, while hiding the details on how the class realizes the behavior of the objects of that class. In contrast, the implementation of a class deals with the details of the realization of the behavior of the objects of that class.

Classes can relate among each other. Examples of commonly used relations constitute the *generalization* (also called *inheritance*) - a “kind of” relation, *aggregation* – a “part of” relation, and *association* - denotes that objects of a class “knows” about objects of other classes (e.g., knows how to send messages to them).

In modern programming languages, such as C + +, Java, and Smalltalk, the interface of a class consists of the declaration of object *attributes* and object *method signatures* (sometimes also called *operation signatures*). Object attributes constitute the state of an object of that class, while method signatures define the different types of activities that objects of this class can perform. We sometimes also refer to the interfaces of an object as its *service*, where a service of a software entity represents its externally visible functionality [ViPi + 00] (as opposed to functionality confined to the inside of the entity, assuming some encapsulation capabilities from that entity). The implementation of a class in a programming language consists of the declaration of the implementations of all class methods with signatures defined in the class interface. A method implementation consists of (a sequence of, in imperative programming languages such as C++) statements defining what this method actually does in terms of the basic functionality offered by the particular programming language. An *object reference* uniquely represents the object identity. Attributes may have as values object references representing the knowledge of an object about other objects [Lieb86].

The following lines written in the Java programming language illustrate the declaration of both the interface and the implementation of a simple class called Friend:

```
class Friend
{
   /* attributes */
   private String name = null;
   /* methods */
   public String getName() { return name; }
   public void setName(String n) { name = n; }
}
```

All objects of the class Friend have a name attribute that determines their state and two methods that allow other objects to access the value of the name attribute – one for setting and one for getting its value.

An *object model* defines the principles according to which we can build object-oriented designs using classes and objects. The classical object model [Booch91] includes the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence. While we consider these principles important, we discuss discus only these principles that relate directly to monitoring issues (e.g., concurrency) in more detail.

3.1.3 Object lifecycle

During runtime, we *create* objects of some class to use their functionality, and *destroy* existing objects when we do not need them anymore. The *lifecycle* of an object extends from the time of its creation, when the execution environment allocates resources for this object, to the moment of its destruction, when the environment reclaims back object resources. Some OO systems allow *persistence* of object state. Object persistence represents the property of an object through which its state transcends the object lifecycle or execution environment (i.e. the object's location moves from one environment to another environment) [Booch91]. In order to achieve persistence an object needs to have the ability to *serialize* its state into bytes that we can record on a storage device or we can transmit over a communication medium.

3.1.4 Objects and concurrency

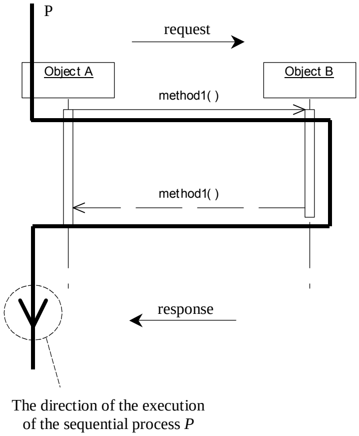
Modern operating systems often involve concurrent execution of sequential activities. We commonly use the concept of processes (also called *fibers* and *threads*) to represent sequential activities executing on multiple hardware processors, or achieving the illusion of concurrency on a single processor by means of some time-sharing mechanism that governs how much time each process gets executed by the single processor. Programming concurrent systems requires dealing with problems such as deadlocks, starvation and race conditions. Object-oriented programming alleviates the concurrency problems by hiding concurrency inside reusable abstractions [LiJo89], such as *active* and *passive* objects. An active object represents a separate process. An active object can initiate activities on its own. An object not associated explicitly with a process and thus only offering functionality through its interface to other objects, we call a *passive* (also *reactive*) object.

3.1.5 Object communication

We mentioned earlier that objects collaborate by passing messages to each other. In modern programming languages, objects communicate through a mechanism called a *method call*. During a method call, one object (the caller) calls a method on another object (the called) by sending to the called object a *request* message comprising some parameters. The called object accepts the call by performing the activity corresponding to the called method from its implementation, using the information from the request message parameters. When the execution of the method completes, the called object prepares a *response* message containing the result of the activity. The calling object may (*synchronous* method call) or may not (*asynchronous* method call) wait for the completion (and possible results) of the execution of the method.

In Figure 3-1, we illustrate how a typical non-distributed object-oriented runtime environment handles method calls. In this figure, object *A* calls method *method1()* on object *B*.

Figure 3-1 A method call



Some process *P* handles a method call in the following way: when *P* reaches the statement of the method call in the implementation of *A*, it starts executing the method in the implementation of *B*. We consider this act of process *P* as the sending of a request message. *P* uses the parameter values from the method call to execute sequentially the statements of the method body. After process *P* completes the last statement of the method body, it returns to execute the statement following the method call in the implementation of *A*. We consider this act of process *P* as the sending of the response message (if any). The results of the method call represented by the parameters of the response message, become available in the implementation of *A*.

3.2 Object middleware

We consider *communication middleware* the software that (1) allows communication between the (potentially distributed) parts of software programs and (2) provides to developers transparencies from the specifics of the underlying communication infrastructure. We start this section by briefly introducing several communication middleware technologies with historical significance to the development of object middleware. Then we introduce object middleware in detail.

3.2.1 Inter process communication

Inter-process communication (IPC) represents a mechanism allowing two processes on the same host computer to communicate with each other. We can find IPC in every major multi-tasking OS. IPC specifies communication at the level of bits and bytes.

Typically, IPC consists of three techniques for process communication:

- shared computer memory – processes communicate through regulated access to shared memory blocks on the same computer;
- synchronized execution – an example of this technique constitutes the "semaphore", which provides mutual exclusion of simultaneous execution of fragments of code within different processes;
- message passing – processes communicate by exchanging messages. Compared to shared memory, message passing requires less synchronization between the communicating processes.

System V IPC [Bach86] constitutes an example of an IPC implementation that has influenced IPC mechanisms in many operating systems.

We consider the main disadvantage of IPC that designers have to deal with details of the representation of more complex data structures in terms of bytes and bits. Combined with the diversification of cheap computer hardware, this leads to problems with program portability stemming, for example, from machine word byte order. The increased availability of networks stimulated the developments of technologies that allow communication between processes on different machines using a higher level of data abstraction than the one supported by IPC.

3.2.2 Remote procedure call

Remote Procedure Call (RPC) represents a technology allowing one process to call (initiate execution of) a procedure from another (possibly remote) process's address space, in the same way as if this procedure belonged to the address space of the caller process. By procedure we mean a named block of behavior that may take an input in order to produce an output. By address space of a process we mean the data (memory) and instructions (code statements) one process can potentially manipulate and perform, respectively.

RPC follows tightly the client / server model. This model defines two roles: a *server* that offers some functionality and a *client* who makes use of this functionality by submitting *requests* to the server and receiving back *responses* to these requests. RPC requires the definition of server functionality using a formal Interface Definition Language (IDL). Programmers use tools to process the IDL and generate automatically all necessary code for interacting with the underlying communication infrastructure. Furthermore, these tools also generate the programming language specific representations that deal with the bits and bytes of complex data structures used as parameters to the procedures. For example, the generated code performs automated marshalling and unmarshaling of the data structures used in programming languages. This resolved the portability problem of IPC we mentioned earlier, because IDL tools for a particular platform hide from programmers the low-level (bits and bytes) representation of data for that platform.

The Open Group promotes DCE RPC [DCE] as a true OS independent RPC standard. Various OS vendors offer their own implementations of RPC. To our understanding however, DCE RPC does not guarantee interoperability between different RPC implementations because each implementation may use its own protocol for serialization of complex structures over a communication infrastructure.

3.2.3 Object middleware

Object middleware came as an answer to the need for a common object oriented infrastructure and a common set of object services on which to build various distributed applications. Object middleware combines object orientation, the client-server approach, and RPC style distributed communication.

Using object middleware, application designers still develop their applications as collections of collaborating objects, however, these objects can now make *remote* method calls (with a semantic similar to the RPC procedure calls) across the boundaries of a single execution environment.

Object middleware separates the interface from the implementation of a class of objects. Object middleware requires the definition of the interface of an object class using an IDL language independent from any programming language. For simplicity, we consider that an object may have only one interface. Similarly to the interface of classical objects, an interface defined in the IDL language consists of attributes and method signatures called *operations*. The attributes present the object's state to other remote objects. Operations define methods that other remote objects may access. The IDL language also allows the definition of complex data types to use for the definition of operation parameters, results, exceptions, etc. To promote reuse in object-oriented style, interfaces may extend other interfaces in a way similar to the generalization relation among classes.

Designers define the class implementation in a particular programming language. For the users of an object however, the details of its implementation lay hidden behind the class interface written in IDL.

In object middleware, the object reference represents a persistent data structure describing among other things, the location of the remote object implementation. Hence, programmers can use an object reference to access the functionality of a remote object regardless of both the programmer's and the object's actual locations.

Object lifecycle revisited

The lifecycle of middleware objects comprises two additional activities as compared to classical objects: activation and deactivation. Below we list the four activities of the object lifecycle:

- Creation - during creation, the middleware creates an unique reference for the object. Nevertheless, in contrast to classical object orientation, the object cannot yet serve requests. At this stage the middleware may have reserved resources for the object execution or may have deferred allocation of these resources to a subsequent object activation;
- Activation – object activation explicitly enables the object to serve requests on its interface. The middleware must have finished all allocations necessary for the normal operation of the object;
- Deactivation – deactivated objects cannot serve requests until a subsequent activation. The middleware may release allocated resources;
- Destruction – the middleware destroys the object and releases all resources required for the object execution.

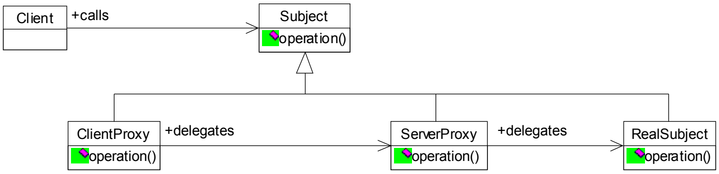
We separate the lifecycle of an object into these four activities because some applications require availability of server objects over a long period of time, despite possible intermediate shutdowns due to imminent hardware failure or maintenance. In case of such events, the applications may deactivate their server objects, store their state somewhere, fix the problems, load the object states, and activate the objects at a later moment. After activation, clients use the same original object references.

Object communication revisited

We denote with *operation invocation* the remote method call mechanism of the middleware. For the moment, we shall consider only a synchronous operation invocation, i.e., one in which the caller waits for the completion and (possible) results of the operation invocation.

The operation invocation uses the Proxy design pattern [BuMe + 96] to provide to developers transparency from physical distribution. Figure 3-2 shows the UML class diagram of an abstract operation invocation.

Figure 3-2 The Proxy pattern as applied in object middleware



A Subject object represents the interface of a remote object called RealSubject. A ClientProxy and a ServerProxy object mediate the invocations on the RealSubject by offering an interface identical to the RealSubject's interface. The Client interacts locally only with the ClientProxy, which hides from the client any distribution aspects of the invocation. The RealSubject interacts locally only with the Server proxy, which also hides the distributed aspects of the invocation but on the server side.

The middleware uses the Broker pattern [BuMe + 96] to handle the interactions among both proxies. Figure 3-3 shows the UML collaboration diagram of an actual operation invocation.

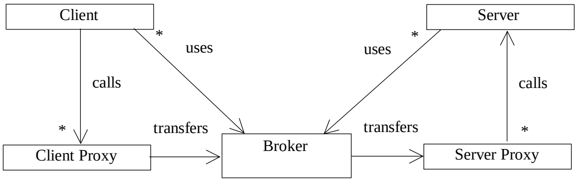
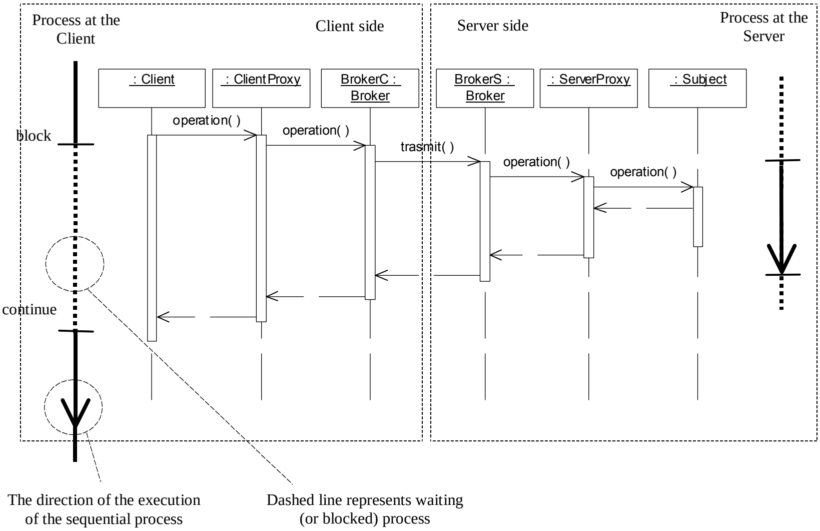


Figure 3-3 The Broker design pattern

A *broker* in object middleware has the responsibility for managing the lifecycle of objects and the communication among objects. In the diagram, the Broker object represents a distributed entity that interacts locally both with the ClientProxy and ServerProxy to handle the operation invocation.

Figure 3-4 shows the UML message sequence diagram of an actual synchronous operation invocation together with an indication about the processes involved in the communication, and in addition the diagram shows the distribution of the Broker (BrokerC on the client side and BrokerS on the server side).

Figure 3-4
Message sequence diagram of an operation invocation



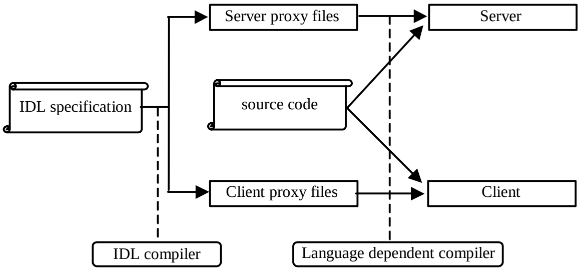
In the context of some process, a Client object initiates an operation invocation using the object reference of the remote Subject object. The ClientProxy object takes control of the operation invocation transparently to the Client object. The ClientProxy object delegates the operation invocation to the Broker object at the client side. The client process then waits until the processing of the invocation at the server side completes. The Broker object at the client side communicates the data associated with the invocation to the Broker object on the server side, using the underlying communication infrastructure and taking care of any details such as time outs and minor communication errors. The Broker then assigns some process at the server side to handle the operation invocation and delegates the invocation to the ServerProxy. The ServerProxy calls the Subject server object to process the operation invocation. In analogy to the classical object method call, the chain of interactions on the forward direction of the operation invocation represents the sending of a request message. After completion, the server sends any results back to the client in the reverse order of operation interaction. The return direction of an operation invocation represents the sending of a response message. After the completion of the operation invocation, the client process resumes its execution. Observe that only the two broker objects communicate over the network. This way both the client and the server objects involved in a synchronous operation invocation make only local (i.e., within the same single execution environment) interactions.

Some object middleware technologies, such as CORBA [CORBA], allow asynchronous remote invocations. In contrast to a synchronous invocation, an asynchronous invocation does not require the calling process to block and wait for a result. At a later moment, the calling process can check for the completion of an asynchronous invocation and eventually acquire the result. We do not discuss the asynchronous invocation in detail. Suffice to say, different vendors often have their own different interpretations about the realization of the asynchronous operation invocation.

Developing object middleware-based applications

Developers must follow particular steps in order to build applications with object middleware. Figure 3-5 presents what we consider a basic programming model for object middleware. In our opinion, one can find the elements of this model more or less in every contemporary object middleware product.

Figure 3-5 A programming model for object middleware



The IDL compiler processes an IDL specification to generate programming language specific mappings for the complex data types in that specification. In addition, for every interface in the specification, the IDL compiler generates proxy files for the client and for the server respectively. Developers use the client proxies to perform calls on server objects, and server objects use the server proxies to accept calls on operations of their server objects. Developers then use a specific programming platform to develop the client and the server parts of their object middleware-based applications.

In addition to the basic communication service of object middleware, developers can also use common object services. An object service consists of a collection of specific remote objects that provide functionality covering a particular aspect of working with remote objects in an application domain independent way (hence “common” services). Examples of object services constitute: directory services allowing discovery and management of object references, transaction services allowing the use of transaction contexts in DSA, and security services allowing the enforcement of various security policies on remote objects.

3.3 Component middleware

The market for software products demands increasingly more complex distributed software. To meet these demands, the software industry turned its attention to *component* technology – a new paradigm for software manufacturing that promises to improve cost efficiency while preserving or improving flexibility, nimbleness and competitive edge of a software product. We consider, among others, the following benefits of component technology: short time-to-market in terms of reduced complexity, improved application productivity in terms of increased reuse of existing code, and programming by assembly rather than engineering.

We call *component middleware*, a middleware that allows one to build software applications using software components.

3.3.1 Software components

A *software component* represents a self-contained and reusable binary unit that provides a unique service, which developers can use either individually or in composition with the services provided by other software components [Szy98]. In this text, we use the term "component" as a shorthand notation for "software component".

A component can participate in a composition together with other components. Components participate in compositions by using each other's services. To our understanding, components represent reusable assets in the design and implementation of new component-based software.

A *component specification* represents the description of a software component. A component specification consists of two main elements: a description of the component service and a description of the component dependencies on the services of other components. We describe the service of a component in the form of one or more interfaces. An interface represents a contract established between a party providing the specific service and all parties that potentially use that service. Component dependencies represent "uses" relationships with interfaces of other components. Furthermore, a component specification may describe various additional component characteristics, such as requirements on the deployment environment for a component, resources necessary for execution of the component, and a description of the binary package of the component.

A component encompasses a certain amount of application functionality. System designers have the task to determine the actual granularity of their components. Too large components can reduce reusability and flexibility of future designs. Too small and too many components may reduce efficiency of software development.

Generally, a component represents a part in some functional decomposition of a complex software system. To allow further decomposition, designers use the notion of *compound* components. A composition of several somewhat simpler components, realizes a *compound* component. We consider the relation between a compound component and the components that participate in it, similar to the “aggregation” relation between object classes.

Components run in special environments called *component containers*. A component relates to its component container in a similar way as a computer program relates to its operating system. During runtime, the container uses a component as a prototype for creating *component instances*. A component instance represents the real world entity whose behaviour contributes to the behaviour of the particular component software.

A *component model* defines the rules by which we implement, deploy and instantiate a component.

3.3.2 Comparison between objects and components

In order to understand the difference between objects and components we need to compare their basic features. Let us note that objects relate to classes, in a similar way as component instances relate to components. Thus we compare classes to components, and objects to component instances.

Classes vs. components

The interface of a class has similarities with the specification of a component. Both the class interface and the component specification provide common views on the instances of classes and components, respectively, by abstracting from the unimportant (during high-level design) implementation details. Components however, have a more elaborate specification than class interfaces, because components represent self-contained and deployable binary units. When compared to components, classes appear underspecified, because classes often have tight relations with other classes, and we can see these relations only from the class implementation (e.g., inside the source code). This makes classes less self contained then components. In contrast, the component model deliberately makes dependencies among components explicit by employing a well-specified “uses” type of relationship with other components.

A component specification may consist of additional (compared to the class interface) information, such as component dependencies, resources (e.g., video, images, etc), and any sub-components in case of compound components. Furthermore, some component middleware supports component specifications independent from a particular programming language. An example constitutes the CORBA Component Model [CCM], where the component specification consists of XML-based component descriptors. In contrast, we usually define the class interface in the same language as the class implementation. We consider the limited portability and interoperability of class specifications as a reason for the advent of the component concept.

We use classes to categorize conceptual objects of arbitrary (often low) granularity. Compared to classes, components represent coarser building blocks. For that reason, designers often cannot sell their classes independently but instead package them into class libraries. In contrast, the idea of selling components as common-off-the-shelf building blocks plays a central role in component middleware.

Designers often use the OO technology to implement the internal mechanisms of a component. For example, designers may implement the internal structure of a component as a collection of middleware objects, each implementing one component interface. Furthermore, vendors often present component middleware products as superstructures on their object middleware products. Examples of this constitute most of the existing commercial component middleware products, such as COM+, products compatible with the EJB or with the CCM specifications.

In this thesis we assume that designers use OO technology to build components, and that a component middleware relies on object middleware.

Objects vs. component instances

Running a component means that the component container creates one or more instances of that component. A component container provides component instances with basic common services, such as transactions, security, persistence, and event service. To run in a particular component container, the component model requires from a component only a minimum common functionality. This allows the component instance to use a variety of complex services of its environment (the container), without having to implement them itself. In comparison with objects, the object-oriented runtime library of some OO platform corresponds to the component container. The object-oriented runtime library that runs object-oriented programs however, does not offer that wide variety of services to objects as containers do to component instances.

In the context of an OO programming language, users can uniquely identify both objects and component instances by means of their references. Component instances however, may offer more than one interface to their users, where users can access each interface through a separate reference. Thus, the same component instance may in fact have several identities or views, each representing the functionality accessible through a reference to a different interface of that component instance.

Generally, both objects and component instances have similar lifecycles – creation, activation, deactivation and destruction. Depending on the component model however, components may offer several additional features related to their lifecycle, such as alternative persistence mechanisms and various session management possibilities, as in Enterprise Java Beans [EJB].

Over the past ten years, object middleware underwent an evolution marked by the definition and standardization of a number of object services, and the appearance of several major vendor technologies, such as CORBA, DCOM, and Java RMI. In order to use these achievements, many component models incorporated an existing object middleware and delegated to it the task for dealing with distributed communication.

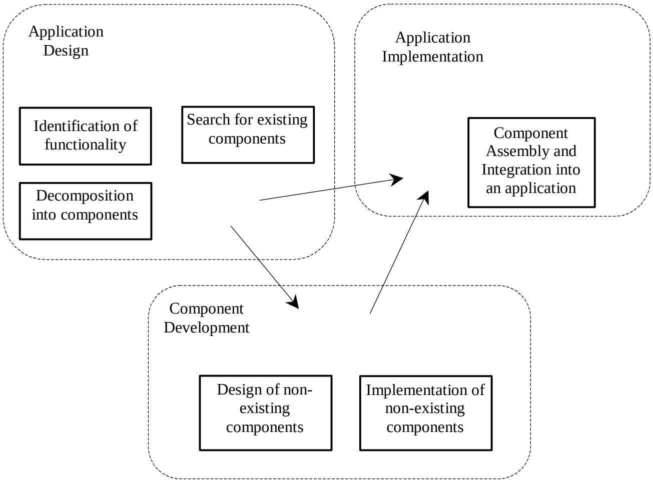
Component instances communicate using two basic styles:

- Object-style operation invocations on interfaces provided by the component instances (and implemented internally by middleware objects). This style of communication comes with the use of object middleware in component middleware;
- Event-based communication. This style uses mechanisms similar to the event service in object middleware. A component event represents a message without a definite receiver. In contrast, an operation invocation always has a well-defined (by a reference) receiver. In some component models, such as the CCM, the component specification allows designers to specify the type of events their components produce and consume.

3.3.3 Developing component-based applications

In this section we present a very high level model for developing component-based applications (Figure 3-6). In our opinion, one can find the elements of this model more or less in every component-based development methodology.

Figure 3-6 A model of component-based application development

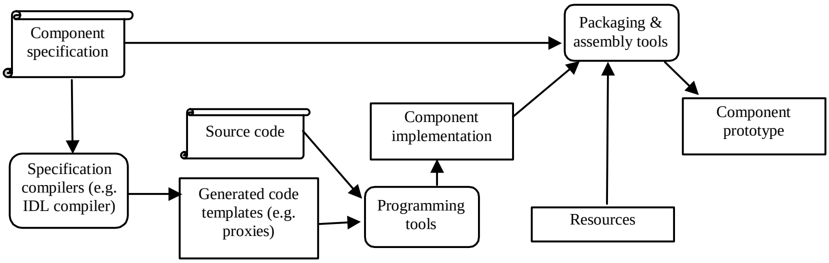

Application Design
Identification of functionality
Search for existing components
Decomposition into components
Application Implementation
Component Assembly and Integration into an application
Component Development
Design of non-existing components
Implementation of non-existing components

During component-based application design, designers identify the future behavior of the application using some common approach, such as requirement analysis. In the next step designers decompose the identified behavior into groups of similar functionality suitable for encapsulation in reusable components. At this point designers may search for existing components matching the required functionality. Typically, some company implemented the (identified) existing components and made them available on the market.

The result of the application design phase constitutes a list of existing application components, a list of non-existing application components that designers need to implement, and any additional application design that governs how application components interact with each other to form the application functionality as a whole. Component developers design and implement non-existing components during the component development phase. The application engineers integrate all (existing and implemented) components together into a prototype of the final application, during the application implementation. We do not discuss iterative development, but designers may use it as they see fit.

Figure 3-7 illustrates the details of the "component development" phase.

Figure 3-7 Component development


Component specification
Packaging & assembly tools
Source code
Component implementation
Component prototype
Specification compilers (e.g. IDL compiler)
Generated code templates (e.g. proxies)
Programming tools
Resources

Specification compilers take the component specification as an input and generate the shell of the component. A shell may consist of, among other things, programming language mappings for complex data structures, and code templates such as proxy files for the client and server roles during operation invocation. From the generated shell and the source code supplied by component developers, programming tools generate the component implementation. The packaging and assembly tools combine the resulting component implementation, component specification and any resources (such as graphics for the visual interface of the components, if any) into a deployable component package that represents the component prototype.

3.3.4 Discussion of existing component models

There exist a number of component models each integrated in different vendor products. We consider the Enterprise JavaBeans (EJB) [EJB], the component model of .NET technology [.NET], and the CORBA Component Model (CCM) [CCM], as the most significant ones from the perspective of strong commercial support for their standardization and development into mature products.

The Microsoft Corporation offers the .NET component model as a result of the evolution of their software technology since the early Windows operating system till the present. The SUN corporation introduced the EJB component model to mark the advances in the evolution of SUN's early Java technology. The OMG consortium developed the CCM, as part of the latest CORBA 3.0 standard. The huge commercial participation in the OMG (800 + companies) resulted in a quite broad component model standard that encompasses many issues not considered by other component models. We consider this one of the reasons why no mature middleware product of a large software vendor supports the full CCM specification yet[3].

[3] Here we mean that we did not know about such product before we finished the work on this text.

Nevertheless, in this section we choose to discuss the basic features of the CCM to illustrate the essential properties of a component model.

The CCM features several major advances to the (object middleware) CORBA standard: an abstract component model, a component implementation framework, a component container model, and packaging and deployment facilities. CCM represents the industry's first component standard taking into account multiple programming languages. We take a closer look at the abstract component model.

CCM Abstract component model

The abstract component model allows component designers to capture how designers view a CORBA component. This includes what services a component offers, which services of other components it requires, what mechanisms components can use to interact with each other, what configurable properties a component may have, and the details of the component lifecycle.

A CORBA component can *inherit* features from one prototypical component and can *support* multiple interfaces. A CORBA component has features called *ports*, to some of which (depends on the port type, see below) designers can connect the ports of other components. According to the abstract component model, a component may have ports of five different types (Figure 3-8):

- Attributes. Similarly to object attributes, a component attribute represents an element of the component state. An attribute has a name and a value. Other components can read and set the values of a component;
- Facets. A facet offers functionality. A facet has a name and corresponds to one interface with a separate reference (object reference to the middleware object implementing this interface). Each facet represents the component by embodying a view that corresponds to a role in which a client may act on the component;
- Receptacles. A receptacle represents the dependency of a component on some functionality provided by another component. A receptacle represents a named holder of a reference to facets of other components. A receptacle can store multiple references and developers can configure the references in it both during design-time (component assembly) and runtime. Third parties can configure a receptacle from outside of its corresponding component;
- Event sources. An event source represents a named connection point that acts as a producer of component events. Multiple consumers may connect to an event source;

– Event sinks. An event sink represents a named connection point that acts as a consumer of events of other components. Multiple event sources may connect to an event sink.

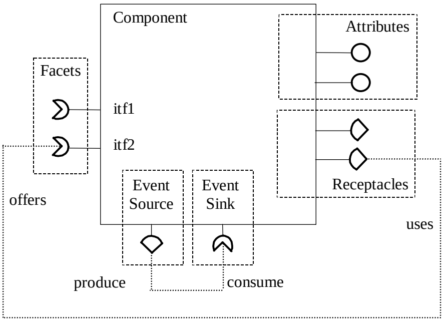


Figure 3-8 A CORBA component

A component reference identifies a component instance. The abstract component model defines mechanisms for navigation among the ports and introspection of a component using its reference.

The CCM allows two modes of component communication: synchronous via a CORBA-style operation invocation and asynchronous via event notification, where the event notification may occur in two forms: direct subscription to an event source and mediated via the CORBA event service.

3.4 Monitoring capabilities in object and component middleware

Having had a closer look at object and component middleware we can now see why monitoring does not present a major functional requirement in an object or component middleware product. Object and component technologies emerged to enhance the production of software by reducing development costs, time to market, and in general make software a commodity that companies can sell off-the-shelf. Monitoring capabilities in the middleware do not enhance software production in general. These capabilities would enhance software production for a class of applications that benefit from these capabilities – monitoring systems for middleware-based applications.

For the reasons above, monitoring of the execution of the internal middleware mechanisms often requires the instrumentation of that middleware. Designers who need to create a middleware instrumentation may find that this task requires extensive knowledge about the middleware's internal mechanisms, in order to avoid exposing partial or unsafe information to the monitoring application. Furthermore, an ad hoc instrumentation of some middleware product may render the monitoring system difficult to maintain (e.g., in case the vendor releases a new version of the middleware, designers may have to re-develop the instrumentation). Some middleware products however, allow for installing middleware “add-ons”, which permit instrumentation in a more *generic* way than others. We generally refer to such products as reflective middleware.

3.4.1 Reflective middleware

We define *reflection* as a technique that allows a software system to support a self-representation in the form of a meta-model. A meta-model represents certain aspects of the software system and its behavior as a collection of meta-objects. Reflection allows one to inspect and/or manipulate the system during runtime [Blair98ii]. Reflection serves the purpose to enhance openness and flexibility in software systems [Yas92]. Reflection differs from ad-hoc system hacks that designers sometimes use to provide reflective capabilities, because reflection pays special attention to providing a consistent meta-model that preserves system integrity.

Reflective middleware constitutes an example of the application of the reflection principle to the middleware concept. In reflective middleware, designers can access some of the middleware internal mechanisms through reflection, in order to configure that middleware to fit different application requirements, or to adapt it to meet changing environment or user requirements [Weg03].

For the purpose of monitoring, we find reflective middleware interesting because it can provide inspection of the behavior of objects and component instances in a generic, application-independent way. Current middleware technologies however, either do not support reflection at all, or provide very limited support for it [Blair98i].

We further focus our interest in reflective middleware to aspects matching the goals of this thesis. Thus, we consider lifecycle reflection and message reflection (of the middleware communication mechanism).

Lifecycle reflection

Lifecycle reflection represents a meta-model mechanism in the middleware that allows inspection and manipulation of the lifecycle of objects and components independently from the lifecycle patterns prescribed by the application design.

In the CORBA [CORBA] object middleware, the Portable Object Adapter (POA) specification defines a standard mechanism for installing custom POAs. Developers can use this mechanism to introduce a special monitoring POA with minimum changes for the CORBA-based distributed application. The monitoring POA can provide consistent and safe information about object creation, activation, deactivation and destruction.

Other middleware technologies, such as COM+ and EJB, provide lifecycle reflection support in their debugging framework. Monitoring of applications using their debugging mode execution, however, may produce a significant monitoring overhead that application users cannot tolerate.

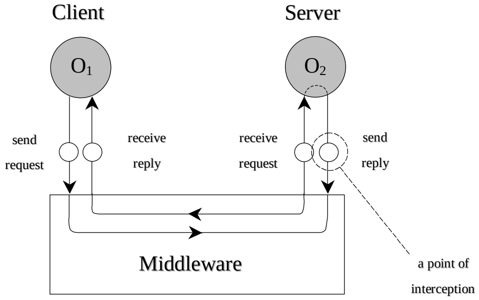
In Chapter 7, we shall use the POA mechanism in CORBA to implement instrumentation capable of monitoring lifecycle information about application objects for CORBA-based software.

Message reflection

Message reflection represents a mechanism in the middleware that allows inspection and possibly manipulation of messages exchanged between application parts independently from the communication patterns prescribed by the application design.

In the CORBA object middleware, the Portable Interceptors (PI) specification defines a mechanism for installing custom components called *interceptors*, on the path of remote operation invocations.

Figure 3-9 Portable Interceptors – points of interception



The PI specification defines the access to the meta-object that reifies an invocation at four moments of its execution (Figure 3-9): directly before sending the request at the client, after receiving the request at the server, before sending a response, and after receiving a response. At the four points, the PI specification determines the invocation information designers can use for monitoring purposes, e.g., operation parameter values or results. The PI specification also defines a method of transparent installation of custom-made interceptors for the CORBA programming language mappings for C++ and Java.

Other middleware products, such as the COM+ middleware [Kath00], also provide message reflection similar to the CORBA portable interceptor mechanism.

3.4.2 Requirements on reflection mechanisms for middleware monitoring

Based on our initial intention to use a middleware-based approach we can define several requirements on the reflection mechanisms that we want to use for monitoring:

- Transparent use. The reflection mechanisms should conform to the middleware principle, by providing transparency to the application layer;
- Information consistency. The reflection mechanism should provide consistent information, meaning that under any circumstances it should not expose to the monitoring system partial or unsafe information about the middleware activities.

3.4.3 Adding message reflection to middleware

When the middleware supports limited message reflection or does not support it at all, instrumentation designers have to add it to the middleware. For the purpose of monitoring, we may need to add inspection capabilities to the communication mechanism of the middleware. We need to select proper mechanisms to do that.

Wegdam [Weg03] makes a survey of various mechanisms for adding message reflection to the middleware. Among others, these include sniffing the wire protocol, middleware interceptors, modification of stubs and skeletons, wrapping, debugging interfaces and composition filters.

We consider middleware interceptors the best choice as they represent a transparent solution, which we can use to monitor during the normal operation of the monitored application. Furthermore, the PI specification for the CORBA middleware has undergone many iterations of refinement of its safety issues, and we consider it very mature to use for monitoring. Middleware interceptors have the disadvantage that they provide only predefined and (intentionally) limited (for integrity purposes) access to the invocation mechanism. For example, according to the CORBA specification, the middleware executes custom portable interceptors in a thread context separate from the application context, which means that the custom interceptors do not have direct access to the thread context of the calling/called objects, which may pose problem when implementing the rules of a logical clock system. Because of this limitation of PI, we consider using PIs in combination with another technique that does not have this limitation.

We consider the modification of the middleware client and server proxies as another promising technique for adding monitoring functionality to the communication mechanism of the middleware. In object middleware, the proxies offer access to the caller/called objects' thread context of the operation invocation before and after the object request broker processes the invocation. In general, we consider modification of the proxies more unsafe than interceptors, because the designer has full control (as compared to the safeguard restrictions in the interceptors) over the operation invocation, which may easily lead to errors. Furthermore, we find proxy modification more intrusive to the monitored application than interceptors, because it often requires recompilation of the monitored application after instrumentation. Nevertheless, proxy modification for object middleware still represents a relatively transparent solution as the IDL compiler (that we consider a part of the middleware) generates them automatically, and hence designers do not participate in the process. In CORBA, a stub and a skeleton represent the proxies for use in the client and the server, respectively.

In Chapter 7 we shall use both CORBA Portable Interceptors and modification of the stubs and skeletons to provide monitoring of remote operation invocations for the CORBA object middleware.

3.5 Conclusion

We can conclude that object middleware has emerged as an evolution of the object oriented approach towards distributed environments. Furthermore, vendors often promote component middleware as a superstructure on object middleware technology. As a result, designers of middleware-based applications rely on the object middleware to handle the distributed communication among component instances, and on object oriented languages to build their components.

We can use lifecycle and message reflection to monitor object and component communication at the middleware layer. This reduces the impact of monitoring on the application. Nevertheless, we rarely find sufficient reflection capabilities in existing middleware. Therefore, in order to allow monitoring at the middleware layer we need to provide some additional instrumentation. We consider as promising instrumentation techniques (1) message reflection through interceptors and (2) proxy modification for monitoring of communication information, and (3) custom POAs in CORBA for monitoring lifecycle information.

Chapter 4

Evaluation of monitoring systems

In this chapter we present and evaluate several existing monitoring systems. With this evaluation we intend to show how well existing monitoring systems support monitoring of object and component communication in a distributed environment.

We start with the definition of evaluation criteria. The evaluation criteria represent the reference point from which we evaluate each system. We then present four monitoring systems: OLT, HiFi, MOTEL, and MIMO. In each presentation we focus on issues relevant to the evaluation criteria. We then summarize our findings (relative to the criteria) into advantages and disadvantages of the presented systems. We conclude the chapter with a list of requirements that our monitoring system should satisfy. We categorize the requirements in generic requirements relating to monitoring of distributed applications in general, and specific requirements that relate to monitoring of object and component middleware-based applications.

4.1 Evaluation criteria

We establish our evaluation criteria using the outstanding problems identified in Chapter 1:

- *Architecture*. A good architecture of a monitoring system establishes the basis for an effective monitoring system. We consider the following qualities of the architecture of a monitoring system: flexibility, extensibility, reusability, and scalability;
- *Middleware instrumentation for monitoring of communication behavior*. Middleware instrumentation allows transparent monitoring of middleware-based applications. Monitoring of communication behavior requires access to the internal middleware communication mechanisms. We consider (1) whether a monitoring system supports middleware

instrumentation for monitoring of communication behavior, (2) what middleware instrumentation mechanisms designers use, (3) the transparency of these mechanisms for monitored application developers including tool support for automatic generation, and the reusability of the instrumentation, (4) the applicability of the instrumentation design to other types of middleware, and (5) the applicability of the instrumentation implementation to different products of the same middleware technology;

- *Support for analysis of concurrent activities.* Establishing temporal order and causal relationships among events in the monitored application may prove difficult. We consider (1) how a monitoring system orders events and (2) what type of causal information (if any) it provides about relationships between events;
- *Dealing with overhead*. Software monitoring inevitably introduces some overhead in the monitored application. Overhead may influence both the users of the monitored application and the users of the monitoring application. To the users of the monitored application overhead may translate into undesirable application behavior. To the users of the monitoring application overhead may translate into inconsistent views on the monitored application behavior. We consider how a monitoring system deals with overhead, i.e., (1) the accuracy and usefulness of the information it provides, and (2) how much the monitoring system interferes (intrusion delay and resource sharing) with the monitored application.

We structure the evaluation of each system using the criteria in the order as defined above.

Comments on the selection of monitoring systems

In the course of our research we have examined several monitoring systems. These monitoring systems roughly fall into the following several main categories of monitoring systems:

- *Application and network management*: MIMO [RLRS00], AppCenter [APPC], TIVOLI [TIVOLI], SILK [SILK], SNMP [SNMP];
- *Reliable messaging*: ISIS [BiJo97], Horus and Ensemble [Bi + 00], JORAM [JORAM];
- *High performance and parallel computing*: ZM4/SIMPLE [Hof + 94], Coral [Zor00], Falcon [Gu + 95];
- *Distributed debuggers*: GDB [SPS02], MAD environment, EMU and ATTEMPT tools [Kranz97], POET [KBTB97], OLT [OLT03];

- *Distributed computing in general*: OMIS [LWSB97], GEM [Sam95], HiFi [Shaer97];
- *Object middleware*: 2K Monitoring Services [Mao99], SmartStubs [SMST], DYNO [Rait00], CorbaTrace [CTRACE], OrWell [WeiKu98], MOTEL [LDKK98];
- *Enterprise monitoring*: BMC Patrol [BMC00], ARM [ARM98].

We have decided to present and evaluate in detail only four monitoring systems because these systems (when compared to all the rest we know about) treat most closely the problems addressed in our evaluation criteria. We start with the Object Level Trace (OLT) – a commercially available system for testing and debugging of distributed software, part of the IBM's Distributed Debugger platform. We select OLT because it represents to date the only[4] commercial product for monitoring that exerts features of interest to us. The other three approaches result from academic research projects with some industrial participation. These systems comprise HiFi, MOTEL and MIMO. The HiFi system deals with various issues regarding scalability of monitoring systems in large distributed environments. The MOTEL system demonstrates the application of formal techniques to the analysis of middleware-based applications. The MIMO system demonstrates a systematic approach for monitoring and management of middleware-based systems.

4.2 OLT

Object Level Trace (OLT) constitutes an extension to the IBM Distributed Debugger that enables developers to trace and debug multilingual, distributed applications. Object Level Trace (OLT) allows developers to monitor the flow of control in a distributed application, and to seamlessly debug client and server code from a single workstation [OLT03].

According to academic sources [Ward01], OLT resulted from collaboration of the IBM Corporation with the research group that developed the POET debugger [KBTB97] at the University of Waterloo, Canada. IBM chooses not to provide any documents on the design approach followed in the development of the OLT monitoring system. We establish our evaluation on the information from user guides for administrators and developers on the IBM official site.

[4] We considered all systems we knew about by the time we finished working on this manuscript.

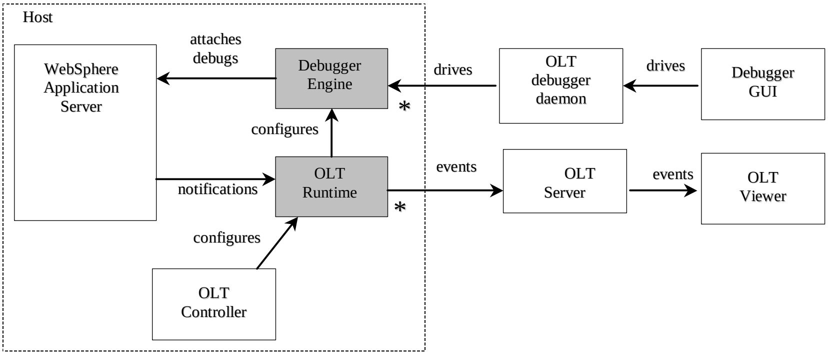
OLT models distributed applications at three levels: hosts, processes/threads, and objects. The host represents an execution environment, e.g., the Java Virtual Machine running an instance of the WebSphere Application Server (WAS). A process/thread represents sequential execution in the common sense of the term. An object in OLT represents a programming language level object. For OLT, a distributed computation consists of events and relations among events, where events represent completed activities of the communication behavior and lifecycles of hosts, objects and processes (threads).

OLT supports tracing and debugging of distributed applications built with Java and C++ for the WAS and the Component Broker of IBM. OLT tightly integrates with IBM's remote debugger. For Java, OLT records Java method calls from a client application to distributed business objects, servlets, JSP, or EJBs residing on WebSphere Application Servers. OLT supports in a similar way C++ programs on several computing platforms from IBM.

4.2.1 Architecture

Figure 4-1 illustrates the architecture of OLT. For brevity we shall examine in detail only the architecture of the Java version of the OLT. The documentation claims that OLT implementations for other platforms comply to this architecture.

Figure 4-1 The OLT architecture



In order to use OLT, developers have to install the OLT runtime and the IBM debugger engine on each host running WAS. An OLT controller configures the OLT runtime so that it knows how to contact the OLT Server and how to configure the debugger engine. The OLT runtime has several responsibilities: (1) to collect notifications from WAS instances, process them and send the resulting monitoring data to the OLT Server, and (2) to start up the debug engine and configure it properly so it knows how to attach to the WAS instances and to the debugger daemon. The OLT Server collects the monitoring data from all OLT runtimes connected to it. The OLT server sends the data to the OLT viewer that shows it to the operator. The debugger engine allows debugging of the WAS instance on a particular application host. Debugger engines connect to a debugger daemon. A debugger daemon can manage several debugging engines. A debugger GUI connects to the debugger daemon. Developers can deploy the components of the OLT and the distributed debugger in several ways. In a most typical scenario, an operator places the debugger GUI and the OLT Viewer on one host so that he (or she) can debug and trace the distributed application form a central place. Developers may deploy the OLT Server and the OLT debugger daemon on separate hosts to minimize the overhead from sharing processing resources with the application hosts.

The designers of OLT specifically created its architecture for integration with remote debugging, which developers usually perform from a central console. Thus OLT uses a centralized architecture in which the OLT Server represents the focal point of all communication. This makes OLT more appropriate for handling debugging tasks in a small testing environment than in a large distributed environment. OLT does not address any scalability requirements beyond the ones present in traditional remote debugging, and any heterogeneity issues beyond the ones present in IBM products involved in the monitoring process.

4.2.2 Middleware instrumentation for monitoring communication behavior

OLT supports monitoring of Java RMI remote objects, but it does not provide explicit support for object middleware such as CORBA, although since JDK 1.3 SE Sun has bundled an implementation of the CORBA standard with the standard Java technology. Nevertheless, testers still can monitor and debug CORBA applications with the OLT indirectly at the level of the proxy objects that CORBA uses – stubs and skeletons.

OLT automatically performs the necessary instrumentation, so that developers can concentrate on debugging. For Java applications, OLT uses the debugging mode of the Java virtual machine to intercept the application execution and perform various measurements.

OLT offers to developers the OLT Viewer and OLT Controller tools that allow viewing and controlling OLT-enabled debugging sessions, respectively. The OLT Viewer presents monitoring information on a diagram that shows the communication between entities (hosts, processes and objects) ordered on a time scale. The OLT Viewer does not provide explicit information about distributed objects or components. The operator has to “know” what the object method call means – a Java remote method invocation, a HTTP request/response, or a CORBA invocation. The OLT Viewer can save diagrams to a file, however it uses a proprietary format which limits the application of external analysis tools, e.g., third-party performance analyzers. In our opinion, OLT monitoring system lacks openness[5], i.e. it does not offer any interfaces for development of custom monitoring tools.

4.2.3 Support for analysis of concurrent activities

The strength of OLT comes from the way it deals with concurrent execution in a distributed environment. OLT employs a vector clock system to impose a partial order among the observed events in a distributed computation. The implementation of the logical clock uses Fidge/Mattern style of vector timestamps for each event in the system [Ward01][6]. The OLT Viewer visualizes communication events in the system matching them appropriately into method calls and remote invocations. The viewer shows order between events in two modes: relative for partial order and total using scalar timestamps from the physical computer clocks. In the latter case, OLT tries to synchronize the clocks on the hosts to maintain a low drift between the clocks on the different hosts.

The partial order relation that OLT uses coincides with the causal precedence relation we defined in Chapter 2, because of the use of vector clocks. Thus, OLT provides information about potential causality between observed events.

4.2.4 Dealing with overhead

A Java machine in debug mode has a considerably lower performance than in normal mode. Developers perform monitoring with OLT in a controlled test environment as opposed to a normal operational deployment. During testing, developers can tolerate a certain level of intrusive overhead (delay) because their primary task consists of removing errors as opposed to using the monitored application for some business logic.

According to the available documentation, OLT allows one to monitor communication aspects of the instrumented (Java in debug mode) monitored application. The OLT Viewer restores both the observed events

[5] The IBM web site does not offer any additional information on the OLT architecture, except administrator and usage guides. Hence, we made this conclusion based on the lack of any APIs for extending OLT with other monitors and analysis tools, such as the OLT Viewer.

[6] We establish this statement on the comments of Paul S. Ward, who has participated in the development of OLT. For more information consult [Ward01].

and the causal precedence relation between them (using vector clocks). This makes the OLT a *totally accurate* monitoring system (see Chapter 2, section 2.8.3). Furthermore, we suspect that OLT meets the *minimal non-interference* property in most cases of monitoring, because it uses the Java debugging platform for making the instrumentation, which typically preserves the original set of events in an application behavior (otherwise we wouldn’t manage to debug applications at all).

4.3 HiFi

The Hierarchical Filtering (HiFi) monitoring system described in [Shaer98] represents an effort to produce a scalable, high-performance, dynamic, flexible and non-intrusive monitoring architecture for large scale distributed systems.

HiFi models the behavior of distributed applications using an abstract event-based model. Designers can use HiFi events to represent both the completion of some activity and the status of entities in the monitored application. The HiFi architecture does not define a concrete monitoring model. HiFi leaves this to the designers of a particular instrumentation. The HiFi system views a distributed application as a collection of (potentially distributed) event producers representing instrumented application parts. The monitoring system collects events and disseminates them to event consumers (monitors). HiFi models monitor requirements for information using the metaphor of a filter. HiFi offers a descriptive language for the definition of filter specifications. The filter specification contains a description of event characteristics that the system uses to determine the relevancy of an event to a monitor.

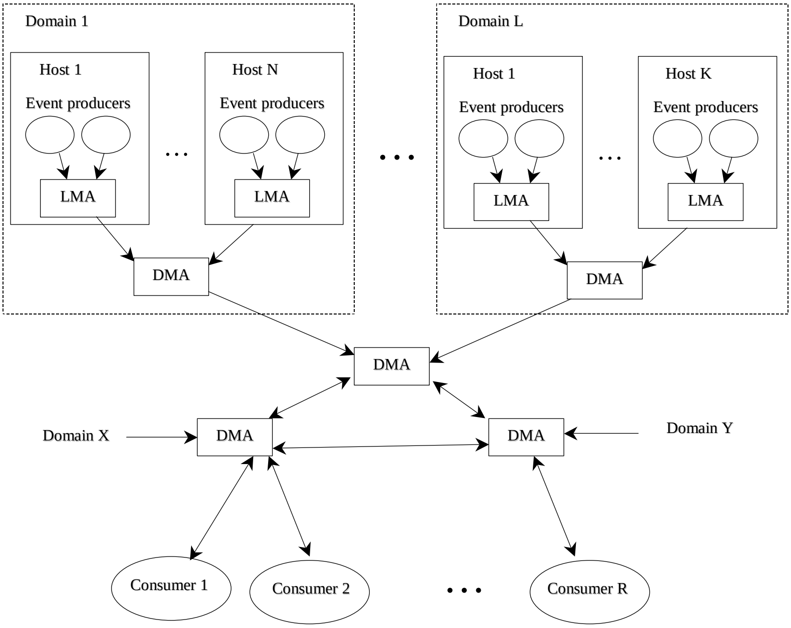
HiFi allows the description of complex events using another declarative language. This language allows the definition of correlations between events using logical “and”, “or” and “not” operators.

The HiFi system clearly separates the role of a producer of monitoring data from the role of a consumer. Nevertheless, HiFi does not consider explicitly the development process of consumer applications (monitors). HiFi considers control actions performed by the consumers on producers, which makes it suitable for management purposes. We address only monitoring (not control) in this thesis.

4.3.1 Architecture

The generic architecture proposed by HiFi proposes solutions to a number of problems identified in large scale applications of monitoring, such as dealing with large amounts of monitoring data, and large amounts of producers and consumers of monitoring data. HiFi proposes solutions for these problems based on hierarchical filtering of information, flexible management of monitor demand for information, and dynamically re-configurable deployment of the monitoring system. Figure 4-2 shows the architecture of the HiFi system.

Figure 4-2 The hierarchical architecture of HiFi



In the HiFi architecture, software entities called monitoring agents (MAs) perform monitoring activities. MAs and their interactions, based on message passing, constitute the HiFi monitoring system.

HiFi requires the organization of the deployment of a distributed application into several domains. Each host has a Local Monitoring Agent (LMA) that belongs to a domain. LMAs collect monitoring data from the event producers. Each domain has one Domain Monitoring Agent (DMA) which collects the monitoring data from the LMAs of that domain. The DMAs participate in a peer network using a reliable group communication infrastructure to exchange monitoring data. The HiFi architecture defines communication and management protocols for the HiFi agents. HiFi supports a dynamic monitoring agent hierarchy, allowing new agents to appear and leave, based on the current needs for information and the system's utilization level.

The HiFi system supports adaptable monitoring agents to alleviate the load of the system. HiFi allows load adaptation during instrumentation and deployment, and dynamic reconfiguration of the agent hierarchy during runtime. The HiFi system maintains the state of its current configuration using a special environment specification language.

A monitoring agent offers its functionality through four services: subscription, instrumentation, event filtering and control. The subscription service provides the following functionality: processes the consumer demands, receives and processes monitoring results, and controls the agent activities. The instrumentation service deals with all aspects of the instrumentation process in HiFi. It offers a number of interfaces and tools that assist the developer in making the instrumentation. The event filtering service handles event processing tasks and filters out events matching the currently installed filters. The control service holds the responsibility for the actual dissemination of monitoring information and for the handling of control actions on the monitored application coming from consumers.

We consider the HiFi architecture highly versatile. Nevertheless, the HiFi architecture and system implementation may prove too complex for cases when we require monitoring of smaller systems in which scalability issues do not present a real problem. In our opinion, a generic architecture for monitoring should facilitate the handling of simpler monitoring scenarios as well.

4.3.2 Middleware instrumentation for monitoring of communication behavior

The HiFi designers acknowledge the need to present automated instrumentation support to application developers, so that they can easily adapt their applications to the HiFi monitoring system. HiFi designers suggest an application independent Code Instrumentation Process (CIP) suitable for instrumenting the monitored application (e.g., its C + + source code). The CIP uses Event Reporting Criteria derived from the subscription information of monitors to create instrumentation for the generation of the necessary events. These criteria strongly couple the instrumentation with the particular monitored application. HiFi does not explicitly consider techniques such as instrumentation at the middleware level. The instrumentation of HiFi stays at the lower abstraction (than middleware) level concepts of the UNIX environment.

4.3.3 Support for analysis of concurrent activities

We consider the major disadvantage of HiFi, the missing support for monitoring the causal precedence of events from the distributed computation of the monitored application, which excludes any analysis based on causal order. Instead HiFi provides total order among events by using timestamps generated from physical computer clocks synchronized using the NTP protocol. This reduces the area of application of the HiFi system to ones that can tolerate a certain amount of errors (from NTP) in the ordering relation.

4.3.4 Dealing with overhead

The HiFi prototype does not include an instrumentation for object or component middleware. The HiFi implementation attempts to alleviate overhead from resource sharing by distributing the processing of monitoring data, e.g., the hierarchical filtering, over the monitoring agents of the system. Furthermore, HiFi provides basic load distribution functionality in the MAs. Based on its current load, an MA may decide to change the size of its internal event queues. Furthermore, under certain conditions a DMA may decide to create new monitoring agents on less loaded hosts on which to delegate some of its event processing. Since we do not consider resource sharing in this thesis we do not discuss load distribution in HiFi in detail. From the report [Shaer98], we can conclude that HiFi covers the *minimal non-interference* feature in most cases of distributed execution of the monitored application, i.e., HiFi can monitor all events that the observed application can produce when unmonitored.

HiFi can provide temporal order among events as accurate as the NTP implementation it uses. Hence, HiFi does not guarantee order among events. Thus we consider HiFi *minimally accurate*, i.e., it can provide accurate information about all detected events, but it reports some of the relations wrong.

4.4 MOTEL

The Monitoring and Testing Tool (MOTEL) described in [Logean00], presents an approach for monitoring and testing communication services built on top of object middleware, such as CORBA.

The method of the MOTEL approach includes expressing application properties at the level of system design using linear time temporal logic [Diet00], and checking for the violation of these properties during the execution of the system prototype. MOTEL consists of a system for monitoring events in middleware-based applications, and of a testing system that uses the information collected by the monitoring system to check for violation of predefined properties. The design of the MOTEL system addresses a broad spectrum of monitoring problems. The MOTEL designers have applied the system to industrial applications for testing and verification of telecommunication services. We consider the major strength of the MOTEL approach the integration of formal methods for verification of software applications with the testing phase of the software development process.

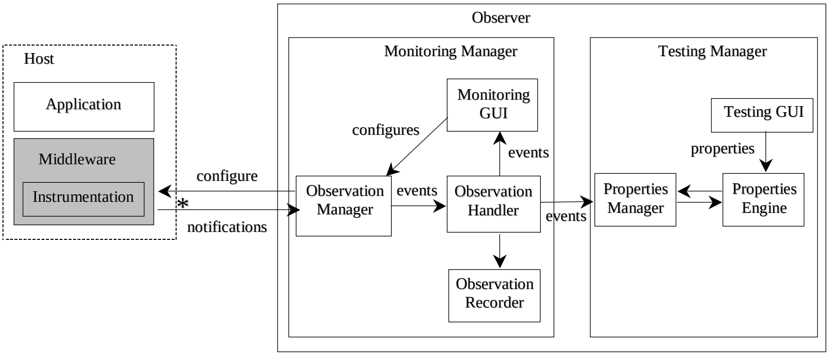
MOTEL defines an event model for monitoring lifecycle and communication behavior of object middleware-based applications. The model uses an event concept similar to our definition from Chapter 2. The model considers events at four different levels: object, thread, process and system level. The object level includes events related to performing communication between objects, such as incoming and outgoing calls. To the process level belong events related to the lifecycle of objects and threads. To the thread level belong events related to message exchange between threads. System level events relate to system wide object availability (the whole distributed system) and to the process lifecycle. MOTEL does not consider component middleware.

MOTEL defines an event as a tuple consisting of an identifier and attributes.

4.4.1 Architecture

Figure 4-3 presents the architecture of the MOTEL system.

Figure 4-3 The MOTEL architecture



The Observation Manager holds the responsibility to configure the instrumented middleware on each host for generating notifications about “interesting” events occurring on that host. The Observation Manager receives notifications and sends the corresponding events to the Observation Handler, which has the responsibility for keeping the event order consistent. Developers may choose to store events for offline analysis or send the events to the Monitoring GUI for online visual presentation. Furthermore, the Observation Handler sends the events to the Properties Manager, which dispatches these in an appropriate form to the Properties

Engine, which can detect violation of system properties in the current distributed computation. The Properties Engine contains a number of components such as an automata optimizer, a property translator, an automation handler and an error handler, which we shall not discuss in further detail. Using the Testing GUI, the user can specify additional properties for testing using the MOTEL system.

Let us note that although dedicated to testing, as a consequence of the modular architecture of MOTEL, designers can use the Monitoring Manager module for generic monitoring with little extra effort (as the authors of MOTEL claim). Furthermore, designers can easily (again a claim of the authors that seems reasonable to us) replace the components of the Monitoring Manager with different implementations to meet new monitoring requirements. Nevertheless, we consider the main disadvantage of the MOTEL system its centralized architecture. In an attempt to solve this problem, MOTEL authors suggest deployment of a separate Observer instance (called a Partial Observer Agent) on each host, that can automatically monitor and test properties local to that host. The deployment still needs a central Observer instance in order to analyze properties that span several hosts. The MOTEL architecture does not further elaborate on distribution or scalability aspects.

The instrumentation and the Observation Manager comprise the "specific" part of the system that depends on the application domain of the monitored application. The other components treat monitoring data (the events) in a general way. The MOTEL system however, does not identify further general services and generic activities, such as a dissemination component, subscription components, etc.

4.4.2 Middleware instrumentation for monitoring of communication behavior

MOTEL performs instrumentation automatically at the middleware layer. MOTEL provides a concrete instrumentation method for the CORBA object middleware. Designers do not need to change anything in their software applications in order to monitor the events identified in the event model. This way designers can concentrate on expressing the behavioral constraints and properties for testing, and devise adequate testing scenarios covering various aspects of the functionality of the monitored application. The MOTEL middleware instrumentation however, uses a specific mechanism (called Message Filters) provided by (an old version of) IONA's Orbix CORBA product [ORBIX]. The more recent and more generic Portable Interceptors [CORBA] mechanism can support the same middleware instrumentation in a vendor independent way. This standard has matured recently, after the completion of the MOTEL work.

4.4.3 Support for analysis of concurrent activities

MOTEL acknowledges the importance of relating event by their order of occurrence. MOTEL employs a system of vector clocks to obtain vector timestamps. These vector timestamps allow the MOTEL system to restore the Lamport's "happened before" relation (causal precedence, Chapter 2, section 2.2.3) between events.

4.4.4 Dealing with overhead

The designers of the MOTEL system intend its use in a testing environment. Hence, the testers using MOTEL may tolerate some intrusive overhead (delay) in the monitored application behavior, because testers want to locate and remove errors rather than use the monitored application's functionality.

The MOTEL system comes with an evaluation with respect to the consistency of monitoring information. This evaluation gives the users of the MOTEL system a precise description of the differences between an un-instrumented unmonitored application execution and the view that the MOTEL system provides on a monitored execution. According to this evaluation the MOTEL system satisfies the *minimal non-interference* property in most cases of application execution, and *total accuracy*. Total accuracy follows from the use of the vector clock system for event timestamping. For details on the evaluation consult [Logean00].

4.5 MIMO

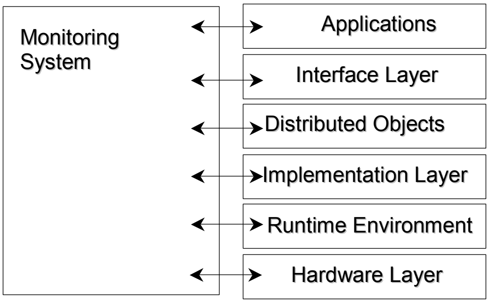
The Middleware Monitoring (MIMO) approach [Rack01] represents a development in the direction of provisioning on-line tool support in heterogeneous middleware environments.

The MIMO approach consists of three major parts: a monitoring infrastructure, the Multi-Layer Monitoring (MLM) information model, and a framework for development of online tools.

The MIMO monitoring infrastructure consists of a design and an implementation of a generic monitoring system, suitable for application to various middleware technologies including object middleware.

MIMO models the monitored application using an entity-relationship model, where entities represent the identifiable things that the monitoring system can observe. Based on analysis of several applications of common types of middleware, MIMO defines the MLM model as shown in Figure 4-4.

Figure 4-4 The MLM model



The MLM information model presents a method for classification of observable entities in a layered architecture, such as the one used in common middleware platforms. A monitoring system based on MLM allows gathering of information at all layers of the model. The MLM allows only one type of general relation among entities. This way MLM makes only weak assertions about relations among entities. Furthermore, entities can participate in relations only between adjacent layers, thus forming a layered relational structure. For example, a concrete implementation of MLM may *relate* a set of interfaces to an application, a distributed object may *offer* an interface, programming language objects may *implement* distributed objects, and processes *execute* those objects on a particular hardware platform. Monitors can use the MLM to provide presentations of the structure of the monitored application during runtime, based on the entities and their relations. Note that the MLM reflects only application structure and not application behavior.

MIMO contains a framework for the development of online tools (monitors) called MIVIS. The MIVIS tools framework consists of a generic monitor that developers of monitoring applications can extend with additional visual components. Each visual component represents a view on the monitoring data coming from the monitoring system. The MIVIS framework supports the development of portable and extendible monitors, using Java and JavaBeans components as implementation technologies.

4.5.1 Architecture

Figure 4-5 presents the conceptual architecture of the MIMO monitoring system.

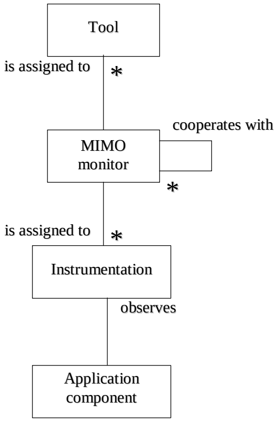


Figure 4-5 The MIMO architecture

The MIMO architecture uses terminology slightly differing from ours. A monitoring system consists of the following types of components: a tool, a MIMO monitor, an instrumentation and an observed application component. Tools (or monitors, in our terminology) represent the users of monitoring data. The MIMO monitor (the MSS, in our terminology) represents the core component of the monitoring system. Tools communicate with the MIMO monitor in order to receive monitoring data. Several MIMO monitor instances may cooperate among each other exchanging monitoring data in a large distributed environment. The MIMO monitor associates with one or more instrumentation instances. The instrumentation supplies the data it observes from components (parts) of the monitored application. Note that the MIMO architecture keeps the structure of the monitoring system independent to the MLM model or any other model of the monitored application, by separating generic monitoring functionality from domain-specific functionality with the help of clearly-defined interfaces between the instrumentation and the MIMO monitor. Furthermore, the interface between the MIMO monitor and the tools completes the separation of concerns, as tools often belong to a specific domain (of the monitoring application) as well.

We consider the main advantage of the MIMO architecture the separation of concerns provided by the interfaces MIMO defines between instrumentation, the monitoring core, and the tool. We however, consider these interfaces limited. We make the general observation that the MIMO architecture focuses on defining the service of the MIMO monitor, while the service of the instrumentation and the tools defines only basic functionality.

Furthermore, MIMO identifies an interface between the instrumentation and the monitored application. In general, we consider this interface application specific and thus irrelevant for a generic approach to monitoring. We would leave the definition of this interface for instrumentation developers.

MIMO deals with scalability issues using a general interface for monitor-to-monitor communication. The MIMO approach suggests a scheme for assignment of monitor instances to various nodes of the distributed environment so that one MIMO monitor always observes the entities of the monitored application on each node. MIMO monitors exchange the data among them using some proprietary protocol. The filtering mechanism that MIMO uses may present problems in large scale systems, because the MIMO monitor typically determines the relevancy (i.e., to discard or not) of monitoring data too close to the tools. In a large distributed environment, this leads to the use of resources for transporting irrelevant monitoring data. An alternative approach similar to the hierarchical organization of HiFi would allow finer control over unnecessary trafficking of monitoring data by using hierarchical filtering techniques.

4.5.2 Middleware instrumentation for monitoring of communication behavior

MIMO defines an MLM specialization to model object middleware-based applications. MIMO designers haven't considered monitoring of distributed component-based applications.

MIMO designers developed an instrumentation that uses the MLM specialization for object middleware to structure the monitoring data it sends to the MIMO monitor. As we showed earlier, the MLM only treats the structure of the monitored application, but not its behavior. MIMO defines only a basic event model (consumer / producer), that delegates to instrumentation developers the responsibility for defining any specific events that represent application behavior.

MIMO suggest two modes of instrumentation: by using *intruders* and by using *adapters*. Intruders represent middleware instrumentation that transparently integrates into the monitored application. In contrast, adapters represent application level instrumentation that developers have to create in order to prepare their applications for monitoring with MIMO. Since we focus on middleware instrumentation, we further discuss only MIMO intruders.

MIMO provides middleware intruders for CORBA and DCOM. The CORBA Intruder represents the MIMO instrumentation for ORBACUS for C + + - based applications. The CORBA Intruder allows for monitoring of events representing object lifecycle activities and the communication between CORBA objects. The CORBA intruder wraps some of the ORB interfaces in order to provide the necessary observation points. Mainly because of the immaturity of the Portable Interceptors specification by that time, MIMO hasn't used interceptors for providing ORB independent instrumentation.

The DCOM intruder represents the MIMO instrumentation for the distributed object technology of Microsoft. The DCOM Intruder allows monitoring of the same set of events as the CORBA Intruder, but for DCOM. The DCOM intruder contains an universal wrapper of COM objects that uses a special DCOM feature to delegate all interactions to the intruder before the object implementation.

4.5.3 Support for analysis of concurrent activities

MIMO does not define an explicit monitoring model of the monitored application behavior. We consider this a serious disadvantage because, for example, dealing with concurrent activities may prove too difficult for users extending the monitoring system with their own instrumentation.

MIMO provides a temporal order among events as accurate as its timestamping mechanisms. For object middleware instrumentation, MIMO uses NTP to generate events. MIMO designers also consider the CORBA Time service as an alternative, but do not provide an implementation.

MIMO does not provide information about causal relationships between events.

4.5.4 Dealing with overhead

The MIMO object middleware instrumentation allows for monitoring middleware-based applications during their normal operation (as opposed to monitoring in a test environment only). MIMO designers provide performance figures for the intrusive effects of their object middleware instrumentation. Using MIMO yields 60-90% monitoring delay per remote operation invocation. Although high, we consider this overhead acceptable for many applications.

We consider MIMO *minimally non-interfering*, because its middleware instrumentation preserves the events in the monitored application behavior.

As discussed in the previous section, MIMO only provides temporal order with the accuracy of the NTP implementation used. Hence, MIMO does not guarantee that monitoring tools can restore correctly the order of events according to the order of event occurrence. Therefore, we consider MIMO, *minimally accurate*, i.e., monitoring tools can monitor all events MIMO detects, but some of the relations among these events may not be reported or reported wrongly.

4.6 Summary

In this section we present a summary of the main advantages and disadvantages of each presented monitoring system.

4.6.1 OLT

Advantages

- Automatic and transparent instrumentation (Java debug mode), applicable to different monitored applications;
- Monitoring of object communication at the level of the OO programming language used to build the monitored application (Java, C + +).
- Causal precedence ("happened before", potential causality) of observed events;
- Total accuracy as a consequence of having causal ordering implemented using a vector clock system;

Disadvantages

- Supports debugging applications only, which makes the development of more general applications of OLT very difficult;
- Architecture does not scale for large distributed environments.
- Architecture does not provide open interfaces for building monitoring extensions or adding monitors;
- May produce high intrusive overhead in the application, because it uses the debugging mode of the execution environment. OLT cannot operate in normal mode;
- Does not support the monitoring of operation invocation for object middleware such as CORBA;
- IBM provides very limited information about the OLT mechanisms and architecture, which does not encourage the evolution of this system by other parties.

4.6.2 HiFi

Advantages

- Hierarchical filtering. Allows better management of the flow of monitoring data to interested consumers in a large distributed environment;
- Hierarchical organization of peer monitoring agents. Allows for flexible and scalable deployments;

- Declarative approach to defining events. Allows flexible event correlation and filtering;
- HiFi architecture defines generic monitoring services within the monitoring system;
- HiFi architecture separates instrumentation from the generic monitoring functionality. This increases reusability of the monitoring system;
- Considers instrumentation tools important for the integration of monitoring systems in the software development process.

Disadvantages

- Does not provide support for monitoring causal order among events;
- Satisfies only minimal accuracy, which may prove insufficient for applications that require accurate analysis of application execution;
- Does not provide a specialization of the instrumentation process for object or component middleware.

4.6.3 MOTEL

Advantages

- Strong formal apparatus behind the definition and evaluation of properties for distributed computations. This increases confidence in the correctness of the results produced by the system;
- Separates the generic monitoring functionality from the domain specific instrumentation functionality;
- Support for causal precedence (“happened before”, potential causality). Demonstrated the usefulness of the “happened before” relation for verification of the behavior of middleware-based applications;
- Fine control over event generation in the instrumentation;
- System satisfies the total accuracy property. The formal evaluation of its overhead gives users confidence about the information they get from the monitoring system.

Disadvantages

- Centralized architecture. The MOTEL designers target to support monitoring in testing environments, and as a consequence, the MOTEL architecture does not scale for use in large distributed environments;
- Does not support component middleware;
- Provides middleware instrumentation only for one particular middleware product;

- Uses vendor-specific instrumentation of the object middleware for inspection of object behavior;
- Although modular, the MOTEL system does not explicitly define general monitoring services and interfaces. This makes it difficult for designers to use its generic parts for different monitoring applications.

4.6.4 MIMO

Advantages

- The MLM model of middleware-based applications. Allows for classification of system entities suitable for presenting abstractions of the structure of monitored applications;
- MIMO separates the concerns about specific and generic monitoring activities using clearly defined interfaces between instrumentation, MIMO monitor and tools;
- Supports monitoring of communication in object middleware (CORBA, DCOM);
- MIMO addresses some scalability issues, by defining a basic architecture for distribution of the logic of the monitoring system among several MIMO monitors.

Disadvantages

- No support for monitoring of causal relationships among events;
- MIMO system satisfies only minimal accuracy, because it uses physical computer clocks to order events;
- MIMO does not support component middleware;
- MIMO system may experience scalability problems. The centralized monitoring component can experience scalability problems in large distributed environments, where the volume of unnecessarily trafficked monitoring data may become significant.

4.7 Conclusions

We can conclude that none of the evaluated systems address sufficiently all of the problems related to our evaluation criteria. In particular, these systems fail to combine (1) a mechanism for monitoring of object and component middleware with (2) a scalable and flexible architecture and (3) an explicit monitoring model suitable for analysis of temporal and causal relationships. Based on the advantages and disadvantages that we summarized above, we define two groups of requirements for our monitoring system:

- *Generic* – Requirements relating to monitoring in distributed environments in general;
- *Specific* – Requirements relating to monitoring of communication behavior in object and component middleware.

Furthermore, we distinguish two types of users of a monitoring system:

- *Monitoring user* – or simply *user*, who uses the information he obtains from the monitoring system for some purpose. Monitoring users show interest mainly in the properties of the monitored application;
- *Monitoring designer* – or simply *designer*, who wants to apply the monitoring system to one or more applications in order to prepare them for monitoring. Monitoring designers show interest mainly in the properties of the monitoring system that potentially reduce the cost for monitoring.

In the next sections we define generic and specific requirements from the perspectives of both types of users of a monitoring system. We shall also indicate for each requirement the general qualities of a monitoring system it relates to, such as performance, usability, scalability, maintainability, reusability, configurability, and portability.

4.7.1 Generic requirements

From the perspective of the user we define the following requirements:

- *Causal precedence*. Monitoring data should contain the necessary information that allows reasoning about causal precedence (see Chapter 2, section 2.2.3) among events from application runs. Note that causal precedence also automatically provides partial temporal order among events in the system. This requirement improves usability for monitoring applications that require temporal and causal order;
- *Online monitoring*. The monitoring system should provide the possibility for timely delivery of monitoring data to monitors, for the purpose of controlling the monitored application based on some analysis from a decision making component in the monitoring application, such as debugging and application management. This requirements addresses performance;
- *Monitoring model*. The monitoring system should provide an explicit monitoring model that defines all aspects of application structure and behavior contained in the monitoring data. This requirement improves usability in general.

From the perspective of the designer we define the following requirements:

- *Large number of producers*. The monitoring system should support a large number of remote application parts producing monitoring data at various rates. This requirement improves scalability;
- *Large number of consumers*. The monitoring system should support a large number of monitors with different requirements for monitoring data. This also implies that the monitoring system should adapt to the requirements from monitors during runtime. This requirement improves scalability;
- *Overhead from intrusion*. The monitoring system should minimize the overhead from injected delay in application behavior by providing a configurable instrumentation that generates only monitoring data relevant to monitors. This requirement improves performance;
- *Separation of generic from specific functionality*. The monitoring system should clearly separate the instrumentation (which we typically consider specific to the domain of the monitored application) from application domain independent monitoring functionality. Furthermore, the monitoring system should clearly separate the tools (monitors) that perform analysis (which we typically consider specific to the domain of the monitoring application) from the application domain independent monitoring functionality. This requirement increases reusability, maintainability and flexibility of the monitoring system.

4.7.2 Specific requirements

From the perspective of the user we define the following specific requirements:

- *Communication and lifecycle*. The monitoring system should provide information about the communication and lifecycle behavior of application objects and/or component instances in middleware-based applications. Communication behavior includes operation invocations and remote method calls (e.g., synchronous and asynchronous), and lifecycle behavior includes creation, activation, deactivation and destruction of objects and component instances. This requirement addresses usability for middleware.

From the perspective of the designer we define the following specific requirements:

- *Transparent middleware instrumentation*. The monitoring system should provide instrumentation of the middleware that conforms as much as possible to the transparency principles of the middleware. This requirement improves the reusability;

- *Middleware tool support*. The monitoring system should minimize manual work by, e.g., entirely automating the instrumentation process. This reduces time for enabling monitoring and increases the quality of the instrumentation. This requirement improves usability, efficiency, and maintainability.

Chapter 5

A design approach for generic monitoring systems

This chapter presents a design approach for monitoring systems. With this design approach we intend to reduce design time and development costs by capturing the important issues in designing monitoring systems and organizing them into appropriate guidelines.

We start with a general discussion in which we identify basic design questions. Based on these questions we decompose the design process into four separate stages. We then elaborate on the steps of each stage in separate sections.

5.1 General discussion

In Chapter 2 we decomposed the monitoring system using the separation of concerns principle. We applied this principle twice, first to separate functionality that deals with the domain of the monitored application from functionality that deals with the monitoring application, and then to separate domain-independent monitoring activities from domain-specific activities. This resulted in the decomposition of the monitoring system into three tiers: an instrumentation tier, a monitor tier and the tier of the MSS. We use this decomposition as a starting point to define a design approach that allows designers to build a monitoring system in a systematic way.

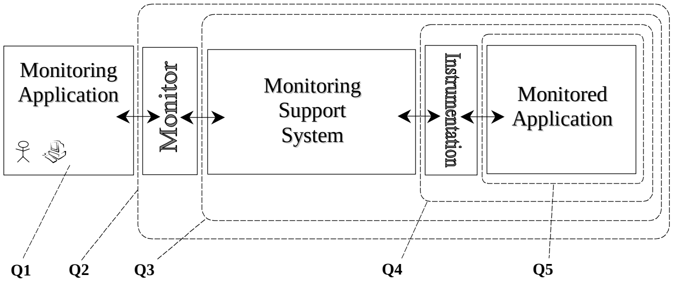
We choose to analyze the design process from the perspective of the monitoring application, because it represents the motivation for monitoring. The monitoring application requires certain monitoring information about the monitored application. For this the monitoring application requires certain functionality from the monitoring system.

We call *service* of a system, the externally observable functionality of that system [ViPi + 00]. In general, to define a service we need to describe the interactions of the systems with its environment (i.e., users), any relations between these interactions, and the information exchanged with these interactions. The service of a monitoring system offers to the monitoring application means to obtain the required monitoring information. We shape our design approach by emphasizing the importance of the information required by the monitoring application, the monitoring data that represents the information, and the measurements performed on the monitored application necessary for acquiring the monitoring data.

5.1.1 Design questions

We want to design a monitoring system modeled as shown on Figure 5-1 (see also Chapter 2, section 2.1.1). In relation to any monitoring system we can phrase a set of five generic design questions (Q1 to Q5) that help formulating and structuring the required monitoring information. A dashed-line round-cornered rectangle shows the scope to which a design question applies.

Figure 5-1
Decomposition of the monitoring system



By answering these questions, designers prepare for the actual design of a concrete monitoring system.

- Q1: For what purpose do we want to monitor?
- Q2: What information do we need to fulfill the monitoring purpose?
- Q3: What monitoring data does the monitor require from the MSS?
- Q4: What monitoring data does the MSS require from the instrumentation?
- Q5: What measurements does the instrumentation have to perform?

To properly introduce the design questions lets consider the following example:

Example 1
Debugging of application faults

A developer needs to locate and remove errors from the prototype of a distributed application. When an error condition occurs, he wants to edit the erroneous source code from the Integrated Development Environment (IDE).

Q1: *Why does the developer want to monitor a prototype of a distributed application?*
The answer to this question determines the purpose for monitoring. This question helps designers to identify the domain of the monitoring application.

In Example 1, the statement "to locate and remove errors from the program's prototype" represents the monitoring application domain. In this case, monitoring plays a role in locating errors during the debugging and testing stage of the software development process.

Q2: *What information does the developer need in order to locate and remove errors?*
To answer this question, designers need to define a high level description of the information delivered by the monitoring system. In the monitoring system, the monitor has the responsibility for presenting this information to the monitoring application.

In Example 1, when an error occurs that the monitoring system can detect, the developer requires the IDE to open and show the appropriate source code file, highlight the statement that may have caused the problem, and provide a human-readable explanation of the error. In this case, the highlighted statement in the source code and the error explanation represent the needed monitoring information.

Q3: *What monitoring data does the monitor require from the MSS?*
The monitor concentrates the knowledge that the monitoring system has about the monitoring application. A monitor requires monitoring data in order to extract and present information to the monitoring application. This question focuses the designer's attention on execution aspects of the monitored application (as a whole), about which the monitor requires information.

In Example 1, the monitor requires from the MSS monitoring data containing information about the file names, line numbers of the statements producing errors, and error codes for all errors that occur in the monitored application. The monitor requires that monitoring data in the order of the occurrence of the errors in the monitored application. Using this information, the monitor can access the source files, and can retrieve human readable error descriptions from some database with error code descriptions.

Q4: *What monitoring data does the MSS require from the instrumentation?*
The MSS concentrates domain independent functionality responsible for the collection, aggregation and dissemination (and possibly processing) of monitoring data. To answer this question, designers need to focus on the monitoring data that the instrumentation (instrumented application parts) provides to the MSS.

In Example 1, when the instrumentation detects an error in a particular application part, it generates monitoring data that comprises the line number of the statement that produced the error, the name of the source file where the statement resides, and the error code. Note that because monitors require delivery of monitoring data in the occurrence order, the instrumentation has to provide to the MSS extra information that allows it to reorder monitoring data, e.g., a timestamp reflecting the moment of error occurrence. This implies that the MSS collects the monitoring data from the instrumentation at the distributed monitored application parts, performs reordering according to the time of occurrence of the errors, and delivers the monitoring data to monitors in the proper order. Note that the MSS does not need to pass timestamps to the monitor, as it already guarantees ordered delivery.

Q5: *What measurements does the instrumentation perform?*

The instrumentation concentrates the knowledge of the monitoring system about the monitored application. The instrumentation performs measurements in order to generate monitoring data containing information about the execution aspects of the monitored application, which the monitoring application considers interesting. The answer to this question determines the type of measurements and indirectly the mechanisms, which the instrumentation uses to perform the measurements.

In Example 1, the instrumentation uses the debug interface of the Java virtual machine running in debug mode, to detect (catch) un-handled system exceptions. The instrumentation analyzes the exception information in order to extract the file name, the line number, and the type of the error that an exception represents. Furthermore, the instrumentation uses the host's computer clock to generate timestamps describing the time of the exception occurrence.

By answering these questions designers collect requirements on the monitoring system, starting from high-level information presented by monitors, and ending at the concrete measurements performed by the instrumentation. We call these requirements *monitoring requirements*.

5.1.2 Building a monitoring system in stages

In order to reduce the complexity of the design process, we suggest designers to address monitoring requirements in a particular order, and possibly in groups based on some common characteristics. In this section, we define design stages for monitoring systems consisting of three tiers: a monitor, an MSS and an instrumentation. Each design stage addresses a subset of the monitoring requirements.

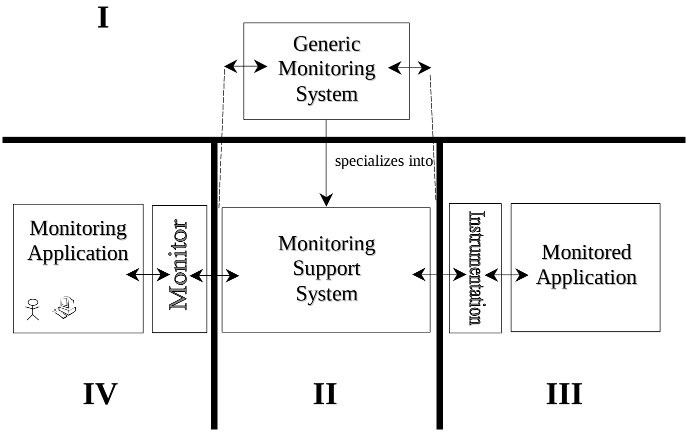


Figure 5-2
Decomposition of the design process

We propose the following decomposition of the design process into four stages (Figure 5-2):

I. GMS design;
II. GMS specialization;
III. Instrumentation design;
IV. Monitor design.

The first stage deals with the design of a Generic Monitoring System (GMS). We base the motivation for this design stage on our prior observation from Chapter 2 that the monitoring system performs common monitoring activities, some of which (e.g., dissemination) we consider independent from the domains of the monitored and the monitoring applications. In this stage, designers generalize the monitoring requirements collected with the help of Q3 and Q4 in order to develop a GMS that works with monitoring data in a domain independent manner. The GMS represents a generalization of an MSS. The GMS architecture provides, among others, the following benefits to designers of monitoring systems:

- Explicit definition of the service of the MSS to facilitate development of new monitors and instrumentations;
- Increased reusability and maintainability of core monitoring functionality in a distributed environment;
- Increased technological independence from the monitored and monitoring domains;
- Improved scalability for facilitating monitoring of large distributed systems.

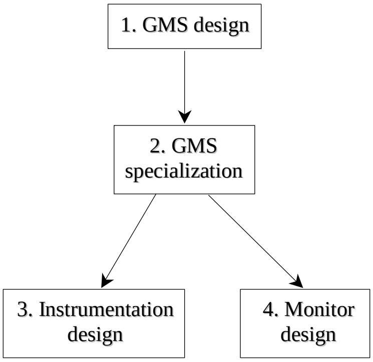
The second stage deals with the specialization of the GMS from the previous stage into an MSS suitable for monitoring of a particular monitored application for a particular purpose. In this stage, designers consider monitoring requirements collected with the help of Q2 and Q3 in order to define a monitoring model of the monitored application. Designers also define a monitoring data structure for the monitoring model. The monitoring data structure defines how one represents instances of the concepts from the monitoring model using concrete data types, such as strings and numbers. Furthermore, in this stage designers consider monitoring requirements from Q4 to identify any functionality of the MSS that requires processing of the monitoring data coming from the instrumentation.

The third stage deals with the instrumentation design for a monitored application according to the monitoring model defined in the previous stage. In this stage, instrumentation designers consider monitoring requirements from Q4 and Q5 in order to design proper measurement mechanisms that can provide the required monitoring data.

The fourth stage deals with the design of monitors that can analyze monitoring data using the monitoring model identified in stage two. In this stage, monitor designers consider monitoring requirements from Q1, Q2, and Q3 to produce a monitor that can provide the required information to the monitoring application.

Figure 5-3 illustrates the relations among the different stages. These relations define a design trajectory that leads designers through the process of designing monitoring systems.

Figure 5-3
Relations among the stages



Note that as a result of the definition of a monitoring model and a data structure in stage two, designers can perform the Monitor and

Instrumentation design stages independently. The independent design of the monitor and the instrumentation provides several benefits:

- The possibility for development of a monitor and an instrumentation by separate teams;
- Makes it easier to support many different monitoring applications that require similar monitoring data from an instrumentation;
- Makes it easier to support instrumentations of many different monitored applications that produce similar monitoring data required by a monitor;

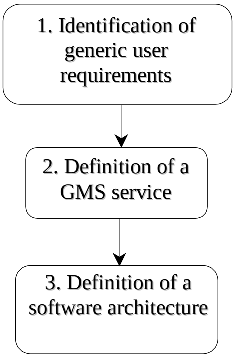
Although the proposed design approach divides the development of a monitoring system into four separate stages, designers can skip a stage, should they find existing software that meets the monitoring requirements. For example, designers can skip the first stage if there already exists a generic monitoring system with the necessary capabilities. In the second stage designers could then only create an MSS as a specialization (a monitoring model and additional processing components) of that monitoring system so that they can build instrumentations and monitors.

In the next sections we describe in more detail the steps in each of the stages. In presenting the stages of our design approach we limit the discussion to design guidelines only. At this point, we do not offer concrete designs nor do we address any implementation issues.

5.2 GMS design

We propose the following three steps during GMS design (Figure 5-4):

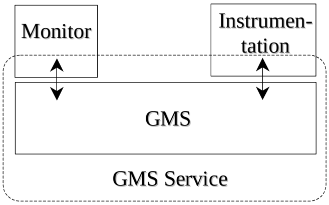
Figure 5-4 Steps in GMS design


1. Identification of generic user requirements
2. Definition of a GMS service
3. Definition of a software architecture

5.2.1 Identification of generic user requirements

In this step, designers start with a view on the GMS from the perspective of its users: the monitor and the instrumentation (Figure 5-5). We consider the GMS a black box characterized by its external behavior – the GMS service. In this context, user requirements represent the requirements of the monitor and the instrumentation on the service of the GMS. These requirements encompass both functional requirements and monitoring requirements (i.e., requirements on monitoring data).

Figure 5-5 The service of the GMS



The GMS represents a generalization of an MSS that deals only with domain independent monitoring activities. These activities include dissemination activities and generic processing activities on monitoring data. Dissemination activities include collecting and delivering monitoring data. Generic processing activities do not require interpretation of domain specific information about the monitored application by the GMS. Designers can define requirements on the dissemination and processing activities performed by the GMS.

The monitoring requirements resulting from Q3 and Q4 concern the monitoring data accessible through the MMS service. In this step, designers generalize these requirements to define requirements on the monitoring data that the GMS provides to its users in a generic way.

5.2.2 Definition of a GMS service

Based on the identified user requirements, designers can proceed with the GMS service definition. We define a service as a collection of service primitives, parameters for each primitive, and the relations between the primitives [ViPi + 00]. A service primitive represents one type of interaction between the system and its users. Designers can organize service primitives into groups called service elements, based on some common characteristics. The relations among service primitives also imply relations among their corresponding service elements.

In order to design the service for the GMS, designers need a starting point. In Figure 5-6 we present one possible GMS monitoring model that defines a general designer’s view on a monitoring system. The monitored application generates some monitoring reports that correspond to events occurring in the parts of the monitored application (producers). Each event has two attributes, *event (source_id, t)*, where *source_id* represents some unique identifier of the producer that generated the event, *t* represents a timestamp of the moment of the event detection by the monitoring system (and the instrumentation in particular). The level of generality and domain independence of the GMS monitoring model, determines the applicability of the GMS to different monitoring and monitored applications.

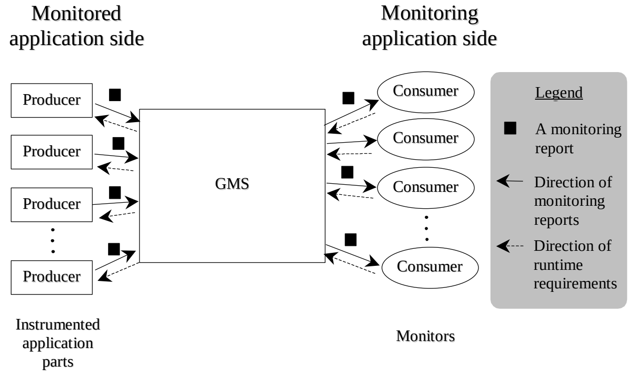

Monitored application side
Monitoring application side
Producer
Producer
Producer
Producer
GMS
Consumer
Consumer
Consumer
Consumer
Legend
■ A monitoring report
← Direction of monitoring reports
←--- Direction of runtime requirements
Instrumented application parts
Monitors

Figure 5-6 A possible GMS model

The monitoring application comprises several consumers (monitor instances), which require information about the behaviour of the monitored application. A consumer expresses its requirements for monitoring information to the GMS in some form. The GMS has the responsibility to supply consumers with monitoring reports according to their requirements.

The monitored application comprises several producers (instrumented application parts) that can generate monitoring reports. The GMS does not make any further assumptions about the behaviour of the producers or about any relations among different producers. Based on the consumer requirements, the GMS can instruct producers to generate monitoring reports containing the required data. The GMS then collects generated monitoring reports from the producers and delivers them to the appropriate consumers.

Monitoring data constitutes a flow of individual monitoring reports from the monitored application to the monitoring application. Designers need to define a generic monitoring data structure for the GMS monitoring model. Consider again Example 1, we define that a monitoring report has one field for every attribute in an event of the generic monitoring model we discussed above. The *source_id* attribute accepts string values generated using the GUID algorithm [RPC], the *t* attribute accepts 64-bit long integer values representing a timestamp in milliseconds since midnight, January 1, 1970 UTC. The generic data structure allows the GMS to process monitoring data independently of the monitored and the monitoring domains, for example, to make sure that the proper monitoring data goes to the correct consumers.

5.2.3 Definition of a software architecture

In the next step, designers open the black box in order to determine how internal GMS components realize the GMS service. The GMS performs two types of activities: generic processing and dissemination (e.g., collecting, delivery) of monitoring data. We propose two views on the internal structure of the GMS: a logical decomposition that defines the functional blocks that perform processing activities, and a physical decomposition that determines the GMS structure according to the distribution aspects of dissemination activities.

Logical decomposition

In a top down approach, a logical decomposition of a software system defines the internal components that implement the system's service. Each component implements some part of the service and collaborates with the other components to achieve the behavior of the system as a whole. We consider a logical decomposition similar to the computational viewpoint of RM-ODP [Put01]. As such, the logical decomposition does not consider any distribution issues.

The logical decomposition of the GMS defines the generic processing and the dissemination components of the GMS. The processing and dissemination activities described in Chapter 2, may give a hint to GMS designers how to structure the GMS functionality. Based on the separate activities, designers can identify components that deal, for example, with subscription, filtering and delivery of monitoring data, and the relations among these components.

Physical decomposition

In order to operate in a distributed environment, a monitoring system may have to satisfy certain distribution and scalability requirements. The physical decomposition of the GMS takes such requirements into account. We consider physical decomposition similar to the engineering viewpoint of RM-ODP.

Monitoring of a distributed application requires the use of a distributed monitoring system. A distributed monitoring system should employ an architecture that does not limit its use together with the monitored application. For example, in an environment of wide physical distribution, monitoring of application parts may require dealing with communication delays (due to use of various types of network), partial failure and temporary network unavailability, in order to deliver collected monitoring reports to interested monitors.

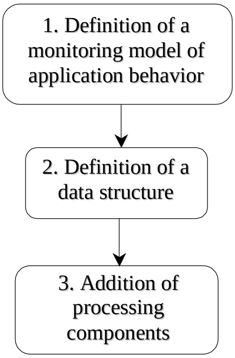
Monitoring of very large and dynamic (in terms of growth) applications require the monitoring system to deal with scalability issues. Large (and growing) amounts of monitoring data may result in exhausting communication and computation resources within the monitoring system. A GMS should employ an architecture that allows one to extend its capacity dynamically in terms of volumes of monitoring data, large amounts of producers producing data, and large amount of consumers requesting data. For example, designers can address scalability issues by allowing flexible deployment of GMS components. The scalable architecture proposed in [Shaer98] presents an example of a solution how to deal with scalability issues, in which designers can deploy the monitoring system using a hierarchical infrastructure of interconnected monitoring agents.

As part of the physical decomposition, designers should also consider how to map the logical structure defined in the previous step, onto the components of the physical decomposition. For example, to improve efficiency, designers may consider distributing a logical filtering component onto several interconnected monitoring agents each responsible for the delivery of monitoring reports to a subset of consumers.

5.3 GMS specialization

We propose the following three steps for the specialization of the GMS (Figure 5-7):

Figure 5-7 Steps in GMS specialization


1. Definition of a monitoring model of application behavior
2. Definition of a data structure
3. Addition of processing components

5.3.1 Definition of a monitoring model

In Chapter 2, we introduced the role of a monitoring model (MM). We encourage designers to define monitoring models for their monitoring systems, because an MM explicitly and systematically identifies and defines the monitoring aspects of application execution [Bates85][KQS92][Hof + 94][Rack99][BeAb02]. Instrumentation designers can use an MM to build an instrumentation for the modelled monitored application. This instrumentation allows one to monitor that application in the terminology of the monitoring model. Monitor designers can use the MM to build monitors that analyse the behaviour of the application. Using the same MM for the instrumentation and the monitor makes sure that they both share the same modelling concepts that allow them to work together.

In stage one, designers define a general model of the monitored application as part of the model of the monitoring system. In section 5.2.2, we represented the monitored application as a collection of producers of monitoring data. In this stage, designers define an MM that specializes that general model.

To develop an MM, designers need two kinds of information: (1) monitoring requirements from Q2 and Q3, and (2) detailed knowledge about the monitored application. A designer uses this information during MM development, to identify interesting entities, and their behavior, in the monitored application. Designers can use status-based or event-based modeling to express entity behavior (Chapter 2, section 2.2.1).

Consider again Example 1, in which the monitoring system should deliver to its users information about errors in running applications. The MM for this example can represent the monitored application as a collection of application part instances, each running on a separate host. Errors relate to the consequences of abnormal system activities (e.g., a sudden and unpredicted hardware failure). The model identifies that system exceptions represents such activities and defines an event called *Error* to represent an exception. An *Error* event has the following information attributes: *error (source_id, t, ln, fname, ecode)*, where *source_id* and *t* we explained in the generic monitoring model, *ln* represents the line number which generated the error, *fname* represents the source filename, and *ecode* represents the error code of the error.

We have to emphasize that an MM differs from a design model. Designers use a design model to model software applications, hence an application model constitutes an instance of the design model. In contrast, designers use a monitoring model to model individual runs of an application, including all aspects of these runs that monitors may potentially consider interesting. Hence, a single run model constitutes an instance of the monitoring model.

5.3.2 Definition of a data structure

An MM constitutes abstract modeling concepts. In order to allow one to generate monitoring data that the monitoring system can transmit to monitors, the instrumentation designers need to define a data structure for the MM. The data structure defines how the instrumentation describes the concepts of the MM using concrete data values, such as numbers and strings found in modern programming languages. Using the monitoring data structure, the instrumentation builds an instance of the MM that represents the execution of the monitored application, by generating and sending to the MSS properly structured monitoring reports. Using the same monitoring data structure, monitors can actually process the monitoring reports coming from the MSS, rebuild the instance of the MM, and using it analyze the application execution that the instrumentation has observed.

Figure 5-8 illustrates the relation between the monitoring models and corresponding data structures for a GMS and an MSS.

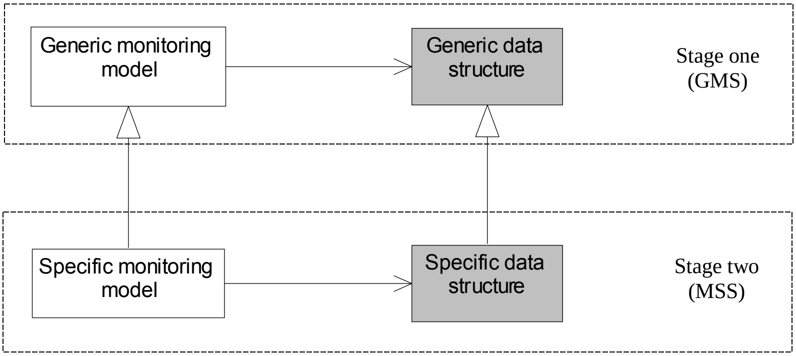


Figure 5-8 Relations among the generic and specific data structures

In this step, designers specialize the generic data structure of the GMS to make a data structure for the MSS. This would allow the GMS to work with the generic structure of monitoring reports (e.g., during generic processing and dissemination) in a domain independent way, while at the same time the instrumentation uses the specialized structure of the monitoring reports to communicate domain specific information to the monitor.

Consider again Example 1. We have identified the entities and represented their erroneous behavior using events. In this example, a monitoring report could represent a single *error* event type. For this simple example we can use some generic event distribution system as a GMS. We

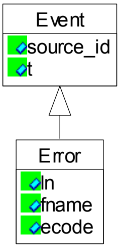


Figure 5-9 The Error class

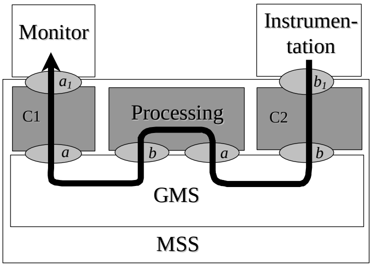
can represent the data structure of a monitoring report in Example 1 using a class as shown on Figure 5-9. The Error class has as attributes the attributes of the error event, however these attributes now can have concrete values. The *source_id* attribute and the *t* attribute we explained earlier, the *ln* attribute accepts 32-bit integers, the *fname* attribute accepts null-terminated strings, and the attribute *ecode* accepts 8-bit integers. Objects of the error class correspond to individual monitoring reports.

5.3.3 Addition of processing components

In some cases of monitoring, designers require the MSS to process monitoring data by interpreting the monitoring data generated by the instrumentation. Since the GMS deals with monitoring data in a generic way only, designers need to add in these cases processing components to produce an MSS that meets their requirements. Back to Example 1, the monitoring system needs to reorder monitoring reports before delivering them to the user. For this purpose, designers define a processing component that reorders the data according to the timestamp attribute *t* of a monitoring report.

When adding processing functionality, to use the benefit of having a generic monitoring system to a full extent, designers should avoid changing the GMS. Figure 5-10 shows how designers can add a processing component (e.g., for reordering) without changing the service of the GMS.

Figure 5-10
Specialization of the GMS into a concrete MSS



Monitors use the interaction point *a* to interact with the GMS. The instrumentation uses interaction point *b* to interact with the GMS. The processing component requires monitoring data in a similar way as the monitor and produces processed monitoring data in a similar way as the instrumentation. The processing component can use the same type of interaction points (and hence the same service primitives involved in the corresponding interactions) as the monitor and the instrumentation. Two new components, C1 and C2, and the processing component implement the service of the MSS (GMS plus the processing functionality). The MSS offers its service through two new interaction points a_1 and b_1 for the monitor and the instrumentation respectively.

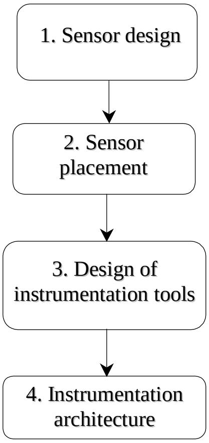
5.4 Instrumentation design

We consider instrumentation design explicitly in a separate step in our design approach, because the quality of the instrumentation may determine the performance of the whole monitoring system [KQS92] [LWSB97] [LDKK98]. The instrumentation design defines the concrete measurement mechanisms that the monitoring system uses to collect the required data.

We model the instrumentation as a collection of one or more *sensors*. In general, a sensor represents “a device that responds to a physical stimulus (as heat, light, sound, pressure, magnetism, or a particular motion) and transmits a resulting impulse (as for measurement or operating a control)” [M-W]. In our case, a *software sensor* represents a small computer program that generates some data output when the environment in which it operates meets some condition defined in the sensor program. When a software sensor produces an output we say that the sensor *triggers* the instrumentation to generate a monitoring report from its output. Further in this text we use the terms “sensor” and “software sensor” interchangeably.

We consider four steps during instrumentation design (Figure 5-11): sensor design, sensor placement, design of instrumentation tools, and definition of an instrumentation architecture.

Figure 5-11 Steps in instrumentation design


1. Sensor design
2. Sensor placement
3. Design of instrumentation tools
4. Instrumentation architecture

5.4.1 Sensor design

During sensor design, designers use the MM to identify individual sensors, what aspect of the monitored application these sensors measure exactly, when they trigger, and what output they produce. Designers may want a sensor to detect an event or to measure a status. For example, if a designer embeds a sensor in the source code of an application, right after the last statement of some complex activity, the triggering of the sensor may represent the completion of the execution of that activity (i.e., its last statement has completed) and hence represent an event. The sensor program may also read the current value of the system clock to generate a timestamp as parts of its output. This timestamp can represent the time attribute of the event. Alternatively, the monitoring system may execute a sensor embedded in the monitored application, in order to perform some measurements of status variable and to generate as output their values.

In this step, designers also use the data structure of the MM to design how the outputs of sensors relate to attributes in monitoring reports.

5.4.2 Sensor placement

During sensor placement, designers determine where and how to position sensors with respect to the monitored application. For example, to detect an event designers may position a sensor after some statement in the application's code. We distinguish several general sensor placement techniques that designers can use to implement their instrumentation in a structured way. These techniques vary by the level of dependence on a particular technology (e.g., programming language or operating system), the amount and type of work required from developers to produce the instrumentation, and the impact of change of the sensor placement on the monitored application:

- *Using available APIs*. Some applications or execution environments may provide Application Programming Interfaces (APIs) suitable for monitoring. Examples of such APIs constitute operating system level notification mechanisms, and middleware mechanisms such as CORBA interceptors [CORBA]. Using APIs potentially has the lowest impact of change and requires less instrumentation work compared to other techniques. This mechanism however, limits the type of things that one can monitor to those predefined in the APIs;
- *Source code modification*. Designers may modify source code to install sensors in it. Source code modification provides maximum opportunity for access to internal application information. This technique however may yield high impact of change. Furthermore, designers need to compile, re-package, and re-deploy the modified application;

- *Binary code modification*. Designers may modify binary code to install sensors in it. Binary modification enables monitoring in cases when designers cannot use source code modification, e.g. because they do not have access to the source code or the legal rights to modify the source code. In contrast to source code modification, binary code modification does not require recompilation of the monitored application. In general, we consider this instrumentation technique difficult to use, because it requires special knowledge about the binary machine code for a particular computer architecture, and in general makes the monitored application and its instrumentation harder to maintain;
- *Wrapping*. In case of wrapping, designers replace an application part by a new component that matches the service provided by the original part. When invoked, the new component may trigger any sensors that designers have embedded in it, and then delegates execution to the original component it replaces, in order to retain the original functionality of the monitored application part. To perform wrapping, designers may use software design patterns, such as Proxy [BuMe + 96] or variations of Wrapper Façade [SSRB00]. Designers may use wrapping instead of source code modification to reduce the impact of change, but they still need to re-deploy the new (wrapper) components;
- *Hardware*. A hardware sensor for digital computing devices represents a digital device that designers use to measure information about the application execution in a non-intrusive way. For example, in [SLC99] designers evaluate the performance of a particular CORBA ORB implementation using hardware sensor capabilities of a VMEbus compliant single-board computer running a real-time operating system. Another example constitutes network sniffing using an Ethernet card to read all traffic on the network.

Various circumstances can influence the designer’s choice for an instrumentation technique. We consider the best choice the use of APIs, as they save development time. Nevertheless, a predefined API may lack richness of the information it can offer to monitors. Source code modification gives greatest control over the monitored application – instrumentation designers can manipulate its source to measure anything they need. When designers cannot modify the source code for some reason, designers can resort to binary code modification. Because computers use binary code for efficient execution and not for human comprehension, binary code modification may turn out difficult to perform by the instrumentation designer. We consider wrapping useful if for some reason (e.g., legal issues) designers do not want to make source or binary modifications to the original application. Wrapping however cannot provide information about the internal structure and mechanisms of a monolithic application component. When monitoring based on software sensors produces too much overhead, designers may use dedicated monitoring hardware to provide information about certain execution aspects while maintaining low overhead. In this thesis we do not consider hardware sensors.

In a heterogeneous distributed environment, designers may use combinations of different sensor placement techniques to prepare an application for monitoring.

5.4.3 Development of instrumentation tools

In order to facilitate the development process for instrumentation that requires the placement of a large number of sensors, designers often use *instrumentation tools* to automate sensor placement. Typically, such tools process the application, either at API, source, or binary level, and insert, install or deploy software sensors, and subsequently (if necessary) compile, build, package and deploy the modified application. We advise designers to consider the additional effort of designing instrumentation tools because they help reduce the error-prone aspects of manual instrumentation. We distinguish two types of instrumentation tools:

- *Design-time*. A design-time instrumentation tool installs sensors during the instrumentation design. For example, a design-time instrumentation tool can automatically process the source code of the monitored application, identify the correct places for sensor embedding, embed sensors, and then compile the instrumented source code;
- *Runtime*. A runtime instrumentation tool installs sensors during the execution of the monitored application. For example, a custom ClassLoader [CLSLDR] can modify the byte code of classes of a monitored Java application during their initial loading in the Java virtual machine.

Designers can use design-time tools for sensor embedding using APIs, source code, and binary modification of the monitored application. In contrast, runtime tools provide in-memory and on-demand binary[7] instrumentation, leaving the original monitored application unchanged. We consider as a major drawback of runtime instrumentation tools the possible

[7] Runtime tools typically have access to the binary code that computers execute. An exception represents an application written in an interpreted scripting language, for which a runtime tool actually embeds code in the source code (since the application source code gets interpreted during runtime by the generic binary code of an interpreter).

difficulties for their development (as compared to design-time tools), because they rely heavily on binary modification.

We consider a principal drawback of all instrumentation tools the additional software development cycle necessary for their creation. In this context, designers of application instrumentation have to consider the usefulness of a tool to justify the effort for its development.

5.4.4 Instrumentation architecture

In this step, designers define the internal architecture of the instrumentation. This architecture defines functional blocks that manage all sensors, collect the data sensors generate and provide this data to the MSS using the GMS service.

In Chapter 7 we provide such an architecture for object and component middleware.

5.4.5 Discussion on instrumentation performance

In Chapter 2 we defined intrusion as the delay a sensor execution may introduce to the application behavior. This delay results from the computing power and memory it takes for a sensor to complete its measurements and generate its output. In general, the more time a sensor spends processing measurements, the more it deprives the monitored application of computing resources.

Designers should constantly keep in mind the intrusive aspect of sensors. A systematic way to assess the intrusion on the monitored application consists of comparing performance times of the monitored application with and without the presence and the operation of sensors. Users of the monitored application may tolerate a certain delay during its operation. As a method for assessing the intrusiveness from the perspective of the users of the monitored application, designers may repeat the acceptance tests[8] of the monitored application in the presence of a running instrumentation.

In Chapter 2 we also presented a method for quantification of the information consistency of a monitoring system, which we consider important for its users. Applying such a method would allow designers to assess the quality of their instrumentation from the perspective of the users of the monitoring system.

During instrumentation design, designers should consider ways of reducing intrusion. For example, the monitoring system can avoid

[8] The ISO9000 series of standards defines acceptance test as the test that determines whether the final product meets user expectations [ISO9000].

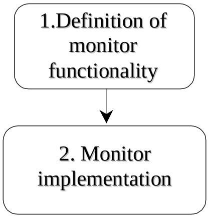
unnecessary sensor execution by switching sensors on and off during runtime, depending on the current demand for monitoring data. The monitoring system can even install or completely remove sensors on-demand during runtime with the help of runtime instrumentation tools. Other examples for reducing intrusion constitute efficient sensor implementation by experienced programmers, using automated code optimizations, and use of profiler tools to discover bottlenecks in sensor execution and replace them with more efficient implementations.

5.5 Monitor design

In this section we discuss monitor design. In many cases, designers need to develop several monitors (or monitoring tools) for several monitoring applications.

We propose a two-step monitor development process (Figure 5-12).

Figure 5-12 Steps in monitor design


1.Definition of monitor functionality
2. Monitor implementation

5.5.1 Definition of monitor functionality

In the first step, designers take into account monitoring requirements derived from the answers to Q1 and Q2 (see section 5.1.1). In this step, we keep the definition of the monitor coarse. For example, designers identify whether the monitor can take decisions and perform actions automatically, whether the monitor can interact with an operator, the type of analysis it performs, and the type of presentation to operators. We discussed issues about the presentation to human operators in Chapter 2.

When determining the aspects of the analysis performed by the monitor, designers use the MM developed in stage 2. Note that the type of (formal) analysis of monitoring data that designers can use in a monitor directly depends on the expressive power of the monitoring model developed for this monitoring system in stage two of our design approach. For example, verification of application execution such as presented in [Logean00] requires a monitoring model of distributed computation that allows verification of prototype behavior against predefined properties.

5.5.2 Monitor implementation

In this step designers implement the software of the monitor. The actual architecture of a monitor may depend on the functionality of the monitoring application as a whole. For example, designers typically implement a monitor that requires interaction with a human operator as a centralized component operating on a single host with some graphical user interface that presents to the operator information about the monitored application. In contrast, a monitor that represents an automated management system may consist of automated agents deployed throughout the distributed environment, performing some analysis on monitoring data.

Different monitors in a monitoring application may share functionality. In such cases, designers may consider to provide a monitor development framework that offers standard functional blocks to use in monitor development. For example, [Rack01] reports the MIVIS framework for Java-based online monitors. Provisioning of a monitor development framework (potentially) reduces the time for the production of monitors.

Designers may find an iterative and incremental approach to the development of complex monitors more manageable and realistic than a single development cycle. In an incremental development, designers gradually add new functionality to the monitor.

Various monitors may have to work cooperatively to achieve a common goal in a monitoring application. For example, in performance analysis a number of monitoring tools may have to analyze various aspects of the same monitoring data. For this reason, designers may consider to use a standard format for exchanging information among analysis tools. For example, in [CTRACE] designers use the XML Meta-data Interchange (XMI) format to represent monitoring information about the communication patterns among distributed objects.

5.6 Conclusions

In this chapter we presented a design approach for monitoring systems. This design approach focuses on the separation of domain-dependent from domain-independent functionality in a monitoring system.

Stage I deals with domain-independent functionality. In this stage, designers design a GMS that provides generic monitoring functionality to monitors and instrumentation.

Stage II deals with the specialization of a GMS into a concrete MSS. In this step designers develop a common monitoring model and a data structure, which allow the instrumentation to communicate monitoring data to a monitor. In this step designers also develop processing components for the MSS.

Stage III deals with the development of an instrumentation for a particular monitored application. This instrumentation effectively prepares the application for monitoring.

Stage IV deals with the development of a monitor. In this stage, designers develop a monitor that can provide the monitoring application with the required information.

In the subsequent Chapters 6, 7, and 8, we follow the stages of this design approach to provide solutions to problems associated with the monitoring of object and component communication in middleware-based distributed applications. We start each of these chapters by providing answers to the preparatory design questions (section 5.1.1) relevant to the design stage under discussion.

Chapter 6

An architecture for a generic monitoring system

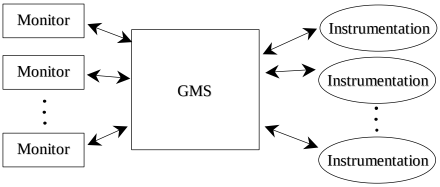
This chapter presents an architecture for a Generic Monitoring System (GMS). This architecture provides a basis for the development of monitoring systems for various monitoring and monitored applications. We define the architecture following stage one of the design approach presented in Chapter 5.

We start with the identification of user requirements. User requirements represent the requirements of the monitor and the instrumentation on the service of the GMS. Based on these requirements we define the GMS service. The GMS service defines the externally observable (from an user perspective) functionality of the GMS. We then define a software architecture for the GMS. The GMS software architecture presents a decomposition of the GMS functionality into components, which cooperatively implement the GMS service. At the end we provide a report about a prototype implementation of the presented architecture.

6.1 Identification of generic user requirements

We choose as a starting point the following high-level model of a GMS (Figure 6-1).

Figure 6-1 GMS model

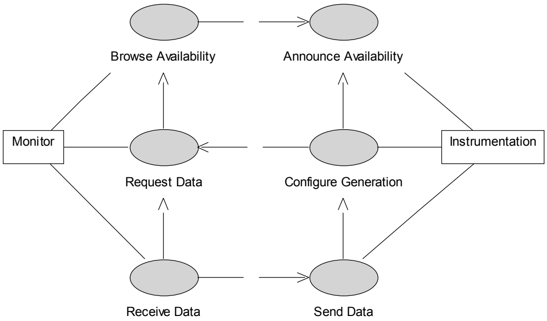


A GMS has two types of users: monitor instances and instrumentation instances. The GMS interacts with one or more monitor instances, which require information about a monitored application. The GMS also interacts with one or more instrumentation instances, which can provide monitoring data containing information about the monitored application. The information about the monitored application consists of statuses measured in an instrumentation instance and events representing activities happening in an instrumentation instance. The GMS model does not define concrete statuses and events. The monitoring data consists of individual monitoring reports (status reports representing statuses and event reports representing events) that an instrumentation instance generates, sends to the GMS, and the GMS delivers to the monitor instances.

We continue with the design of the GMS by focusing on the kind of functionality a monitor and an instrumentation require from the GMS. We capture user requirements with the help of the use case scenario “Performing online monitoring”. Figure 6-2 shows an UML use case diagram of the scenario. This scenario takes into account six cases of use of the GMS during online monitoring.

We focus on online monitoring, because in general, offline monitoring poses less restrictive constraints on the functionality of the GMS (see Chapter 2) than online monitoring. For example, in online monitoring users typically have requirements on the time it takes to deliver monitoring data, while in offline monitoring such a requirement does not exist. Therefore, designers can adapt a GMS service design that supports online monitoring requirements, to a design that support offline monitoring, by removing some of its functionality. For example, an offline monitoring system (one that collects monitoring data first, and then at an arbitrary time later makes it available to analysis tools) does not require subscription functionality that allows delivery of monitoring data to monitors as soon as the instrumentation generates it.

Figure 6-2
“Performing online monitoring” use case diagram



The diagram contains six use cases: Browse Availability, Request Data, Receive Data, Announce Availability, Configure Generation, and Send Data. The first three illustrate how the monitor interacts with (uses) the GMS, and the following three illustrate how the instrumentation interacts with (uses) the GMS. The use cases "depend" on each other in the sense that the execution of one use case enables the execution of another. In the following sections we describe each use case and the dependencies among the use cases.

6.1.1 Browse Availability

A (generic for the GMS) monitoring model determines all possible types of monitoring data available to monitors. Nevertheless, at some moment during runtime, the monitored application may not produce all possible types of monitoring data. For example, to monitor a certain type of monitoring data may depend on the instantiation of a specific part of the monitored application that produces this monitoring data. In this use case, a monitor finds about the available types of monitoring data during runtime. The GMS presents availability information to the monitor in the form of a *specification of availability*. Note that the specification of availability provided to monitors may change over time.

6.1.2 Request Data

In this use case a monitor requests monitoring data from the GMS. For example, a monitor may request monitoring data in two ways: *subscription-based* and *interrogation-based*. In case of a subscription-based request, the monitor announces to the GMS its presence and submits a specification of interest. At later moments, the GMS notifies the monitor about new monitoring data that matches the criteria from the monitor's specification of interest.

In case of an interrogation-based request, the monitor requests monitoring data from the GMS in a request/response style. It does this by sending a selection criteria along with the request and requires within some time limits, a response from the GMS with monitoring data that matches the selection criteria.

6.1.3 Receive Data

In this use case the monitor receives new monitoring data from the GMS. The method of requesting monitoring data determines the way the GMS delivers new monitoring data: one or more notifications in case of a subscription or a single response in case of an interrogation (see previous section).

6.1.4 Announce Availability

In this use case, an instrumentation announces (to the GMS) what kind of data it offers for monitoring. For this purpose, the instrumentation submits to the GMS a *specification of availability*. The specification of availability may change over time, for example, the instrumentation can withdraw its specification when an instrumented application part it monitors goes offline (i.e., terminates).

6.1.5 Configure Generation

In this use case the instrumentation receives configuration information from the GMS in order to generate monitoring data. For example, as a result of configuration, the instrumentation switches some of its sensors on or off, so that it only generates monitoring data relevant to monitors.

We consider this use case similar to the “request data” use case, in which the monitor “configures” the GMS to deliver (generate) monitoring data using the monitor’s subscription or request information. We use this observation later to identify dependencies among service primitives used by the monitor and the instrumentation.

6.1.6 Send Data

In this use case, the instrumentation sends monitoring data to the GMS. We consider this use case similar to the “receive data” use case, in which the GMS sends monitoring data to the monitor. We also use this observation later to identify dependencies among service primitives used by the monitor and the instrumentation.

6.1.7 Use case dependencies

“Browse availability” depends on “announce availability” because the instrumentation has to send availability information to the GMS before the GMS can send it to monitors. For example, the instrumented components of a distributed monitored application may individually announce availability information as they become online (i.e., get instantiated by the environment), and withdraw availability information as they go offline (i.e., get terminated by the environment). This way the GMS can maintain a view on what monitoring data it can provide at any moment.

“Request data” depends on “browse availability” because, for example, using the specification of availability a monitor can compose a specification of interest that precisely defines the monitor's requirements according to the current availability of monitoring data.

"Receive data" depends on "request data", because monitors receive notifications about new monitoring data as a consequence of establishing a subscription with the GMS, and receive a response as a consequence of an interrogation-based request.

"Receive data" also depends on "send data" because monitors cannot receive monitoring data that the instrumentation hasn't sent to the GMS.

"Send data" depends on "configure generation". For example, only sensors that the GMS has switched on can generate monitoring data.

"Configure generation" depends on "request data" because, for example, based on the monitor's subscription, the GMS configures the instrumentation so that it generates only data needed by monitors.

"Configure generation" also depends on "announce availability" because the GMS configures the generation only for data that the instrumentation can generate.

Alternatively, we can view a use case as a phase in the interaction of a GMS user with the GMS system and hence the relations between them as precedence in performing of each phase.

6.2 Definition of the GMS service

The GMS service defines the externally observable (by users) functionality of the GMS system. Among other things, the GMS service shields the monitor and the instrumentation from the details about dissemination and processing of monitoring data in a distributed environment.

6.2.1 Basic concepts

Before we proceed with the service definition we need to introduce some terminology.

We use the terms *service user* and *service provider* as defined in the OSI Service Conventions Technical Report [ISO87]. The GMS system represents the service provider for the GMS service. We call a *GMS user* the service user of the GMS service. A *monitor* represents a GMS user that requires monitoring data about the execution of a monitored application. An *instrumentation* represents a GMS user that provides monitoring data about the execution of a monitored application.

A *service access point* (SAP) represents an interaction point between a service provider and a service user. A *service primitive* (SP) represents an interaction that can occur at a SAP, where each interaction takes certain service parameters. For the GMS service, we distinguish two types of SAPs, one between a monitor and the GMS (M-SAP) and one between an instrumentation and the GMS (I-SAP).

The GMS service consists of service primitives, service parameters and relations between service primitives. A *service element* may group service primitives by some common function of the GMS service provider, effectively structuring the GMS service. Service elements can relate among each other based on the relations among participating service primitives.

Figure 6-3 shows a sequence diagram illustrating how we describe service primitives and the possible relations among them with respect to SAPs. We use a notation based on time sequence diagrams of ISO/OSI service conventions [ISO87] with some minor adjustments. The axis of a SAP represents time, an arrow represents a service primitive, the direction of an arrow indicates who has the initiative for the interaction, and an arrow higher than others on the axis means the corresponding service primitive occurs prior to the ones below. We also allow depicting two independent time sequence diagram if they share the same SAP(s) by connecting the SAP axis with a “spring”-like line. With a dashed straight or arc line we express a causal relationship (with the semantics of realized causality) between two service primitives at different or the same SAP. We express explicitly that service primitives belonging to two distinct SAPs can occur in any time order (hence we consider them causally independent) by the absence of a dashed line between them and using arrows at approximately the same vertical position on the time axes. A fully qualified name of a service primitive consists of the name of its service element in “bold” script (S1 to S4 on the diagram) concatenated with one “space” and with the short name of the service primitive in “normal” script.

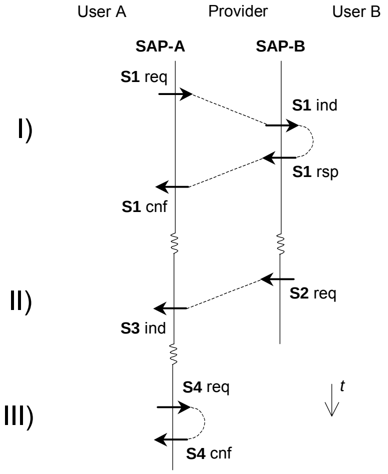


Figure 6-3 Example service primitives and relations among them.

We use four basic types of service primitives : “req”, “ind”, “rsp”, and “cnf”. A service user initiates *req* and *rsp* primitives on the service provider, where *req* represents a request and *rsp* represents a response to a previous request. The service provider initiates *ind* and *cnf* primitives, where *ind* represents an indication of a request at another SAP and *cnf* represents a confirmation of a response at another SAP. In the diagram, we show three cases. The first case shows an RPC style method invocation as a sequence of *req*, *ind*, *rsp*, and *cnf* primitives. The second case shows that a service user can make a request without expecting a result, for example, to notify the service provider (and another user) about some change in its state. The third case shows that interaction does not necessarily involve several users. In this case, the provider initiates by himself a response to the user’s request. During service design, we use the service primitive types to define the concrete service primitives for the GMS system. We indicate the type of a service primitive by concatenating its name with the type name.

In the following sections, we derive service elements from the use cases we identified earlier, and then we refine the service elements into constituent service primitives, their parameters and the relations among the service primitives.

6.2.2 GMS service elements

We define the GMS service elements using the user requirements that we have captured with the use case scenario presented in section 6.1.

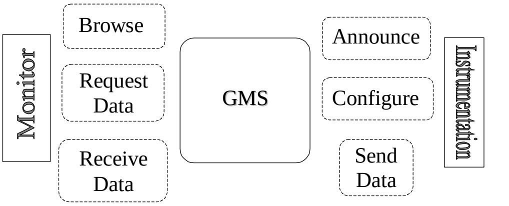


Figure 6-4 Elements of the GMS service

At an M-SAP, we define the following three service elements (Figure 6-4):

- Browse – service primitives that monitors use to browse the monitoring data types available for monitoring;
- Request Data – service primitives that the monitor uses to request monitoring data from the GMS;
- Receive Data – service primitives that the monitor uses to receive monitoring data.

At an I-SAP, we define the following three service elements:

- Announce – service primitives that the instrumentation uses to announce the monitoring data types available for monitoring;
- Configure – service primitives that the instrumentation uses to receive configuration information from the GMS;
- Send Data – service primitives that the instrumentation uses to send monitoring data to the GMS.

Note that each use case represents a block of functionality that GMS users need to use. Since a service element represents a group of common functions, we choose to represent the functionality in each use case with a corresponding service element that groups the service primitives that provide that functionality.

6.2.3 Relations between GMS service elements

We associate each service element with a specific case of use of the GMS service. A service element relates to other service elements by enabling the (interaction) phases that correspond to the use cases associated with the other service elements. Figure 6-5 shows an example of these enabling relations for a single monitoring report.

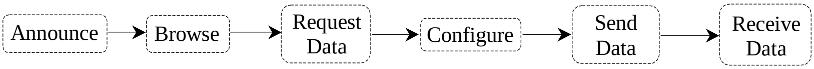


Figure 6-5 Enabling relations between the six service elements

In a typical scenario, an instrumentation announces that it can produce certain types of monitoring reports. This allows a monitor to browse through the availability information and make a selection. Based on the selection, a monitor can compose a specification of interest and use it to request monitoring reports. As a result of the request, the GMS configures the instrumentation to generate relevant monitoring reports. The instrumentation then sends to the GMS monitoring reports that it generates, e.g., when interesting events occur, or when the time comes to perform a measurement. The monitor can receive the new monitoring reports from the GMS.

We refine this informal description of dependencies among service elements later when we describe the relations among their corresponding service primitives.

6.2.4 Service primitives

Table 6-1 lists the primitives of the GMS service and their parameters.

Table 6-1 GMS service primitives

User SAP	Element	Primitive name	Parameters
Monitor	Browse	interrogate req	search criteria
		interrogate cnf	specification of availability
		subscribe req	monitor id, types notification reference
		subscribe cnf	types subscription status
		unsubscribe req	monitor id
		update ind	update status
	Request Data	subscribe req	monitor id, specification of interest, data notification reference
		subscribe cnf	data subscription status
		unsubscribe req	monitor id
		interrogate req	data selection criteria, response notification reference
	Receive Data	notify ind	monitoring data
		interrogate cnf	monitoring data
Instrumen-tation	Announce	register req	instrumentation id, specification of availability
		register cnf	registration status
		unregister req	instrumentation id
	Configure	configure ind	configuration specification
		configure rsp	configuration status
	Send Data	notify req	monitoring data
		interrogate ind	data selection criteria
		interrogate rsp	monitoring data

In the following sections we describe each service primitive grouped by service elements. We discuss the relations between service primitives belonging to the same service element, and we discuss the roles of the parameters of the service primitives. In section 6.2.5 we discuss the relations between service primitives belonging to different service elements. In section 6.2.6 we discuss the service primitive parameters.

Browse

A monitor issues an “interrogate req” to request information about the availability of monitoring data. This information has to match the *search criteria* parameter.

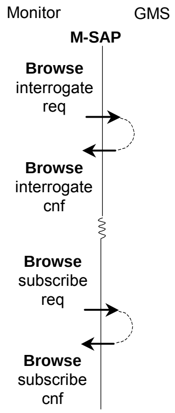


Figure 6-6 Browse service primitive relations

The GMS issues an “interrogate cnf” to send a *specification of availability* parameter to a monitor. This parameter contains a description of the monitoring data types matching the search criteria passed in a previous “interrogate req” from the monitor (Figure 6-6).

A monitor issues a “subscribe req” to initiate a subscription for updates on changes in the availability of monitoring data. The *monitor id* parameter contains an unique identifier of the monitor. The GMS uses this identifier to distinguish among different monitors. The *types notification reference* parameter contains the monitor address that the GMS uses to send change notifications.

The GMS issues a “subscribe cnf” to report the result of a previous “subscribe req” (Figure 6-6). The *types subscription status* parameter indicates a successful subscription or an error.

A monitor issues an “unsubscribe req” to notify the GMS that it withdraws from a subscription for type change notifications. The monitor sends its *monitor id* to identify the subscription that the GMS has to terminate.

The GMS issues an “update ind” to notify a subscribed monitor about changes in the availability of information. The *update status* indicates the kind of update: removing, adding or changing the information about data types.

Request Data

A monitor issues a “subscribe req” to subscribe with the GMS for new monitoring data. The *monitor id* parameter contains an unique identifier of the monitor. The GMS uses this identifier to distinguish the subscription of this monitor from subscriptions of other monitors. The *specification of interest* parameter contains the runtime requirements of the monitor for monitoring data. Runtime requirements represent some criteria according to which the GMS can decide whether any new monitoring data matches the interest of a monitor. The *data notification reference* parameter contains the monitor’s address that the GMS will use to send relevant monitoring data to the monitor. We assume that a monitor can have only one subscription. If a monitor requires additional monitoring data, it can always unsubscribe first, and subscribe again by specifying a new *specification of interest* reflecting its new requirements for monitoring data.

The GMS issues a “subscribe cnf” to report the result of a previous “subscribe req”. This relation however involves interactions at the I-SAP too and we discuss it in section 6.2.5. The *data subscription status* parameter indicates a successful subscription or an error.

A monitor issues an “unsubscribe req” to notify the GMS that it withdraws from a subscription. The monitor sends its *monitor id* to identify the subscription that the GMS has to terminate.

A monitor issues an “interrogate req” to instruct the GMS to make a measurement and return the resulting monitoring data. This also means that effectively the monitor obtains monitoring data in a request/response style. The *data selection criteria* parameter contains the monitor’s runtime requirements for monitoring data. The runtime requirements here describe instructions for performing measurements and returning a response. The *response notification reference* parameter contains the address that the GMS will use to send the response of a request.

Receive Data

The GMS issues a “notify ind” when it has obtained new monitoring data that matches the specification of interest of a subscribed monitor. The *monitoring data* parameter contains the data matching the specification of interest.

The GMS issues a “interrogate cnf” to deliver a response to a previous monitor “interrogate req”. This relation however involves interactions at the I-SAP too and we discuss it in section 6.2.5. The *monitoring data* parameter contains the data matching the monitor requirements.

Announce

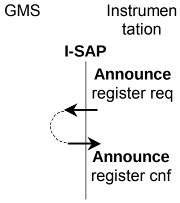


Figure 6-7
Announce service primitive relations

The instrumentation issues a “register req” to register with the GMS the types of monitoring data it can provide. The *instrumentation id* parameter contains the unique identifier of the instrumentation. The *specification of availability* parameter contains a description of the data types this instrumentation can provide.

The GMS issues a “register cnf” to report the result of a “register req” (Figure 6-7). The *registration status* parameter indicates a successful registration or an error.

The instrumentation issues an “unregister req” to un-register with the GMS all monitoring data types this instrumentation has previously registered. The instrumentation sends along its *instrumentation id* to identify the registration that the GMS has to terminate.

Configure

The GMS issues a “configure ind” to configure an instrumentation for generation of monitoring data as a result of a subscription request (see section 6.2.5 for the relations among SAPs). The *configuration specification* parameter contains the configuration information.

The instrumentation issues a “configure rsp” to report the result of a previous “configure ind” (Figure 6-8). The *configuration status* parameter indicates a successful configuration or an error.

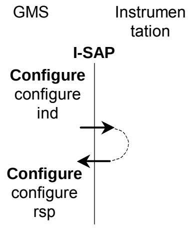


Figure 6-8 Configure service primitive relations

Send Data

The instrumentation issues a “notify req” to notify the GMS about new monitoring data. Note that notifications allow (potentially) low delivery times for event reports (or status reports generated in an event-driven manner), because the instrumentation can notify the GMS as soon as it detects an event and packages an event report. The *monitoring data* parameter contains the new event report.

The GMS issues an “interrogate ind” to instruct the instrumentation to take measurements on behalf of some monitor. This request effectively means that the instrumentation generates monitoring data in an on-demand or time-driven way. The *data selection criteria* parameter contains the runtime requirements from some monitor for monitoring data.

The instrumentation issues a “interrogate rsp” in response to a previous “interrogate ind” (Figure 6-9). The *monitoring data* parameter contains the data matching a selection criteria previously specified by some monitor.

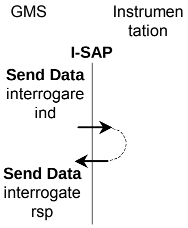


Figure 6-9 Send Data service primitive relations

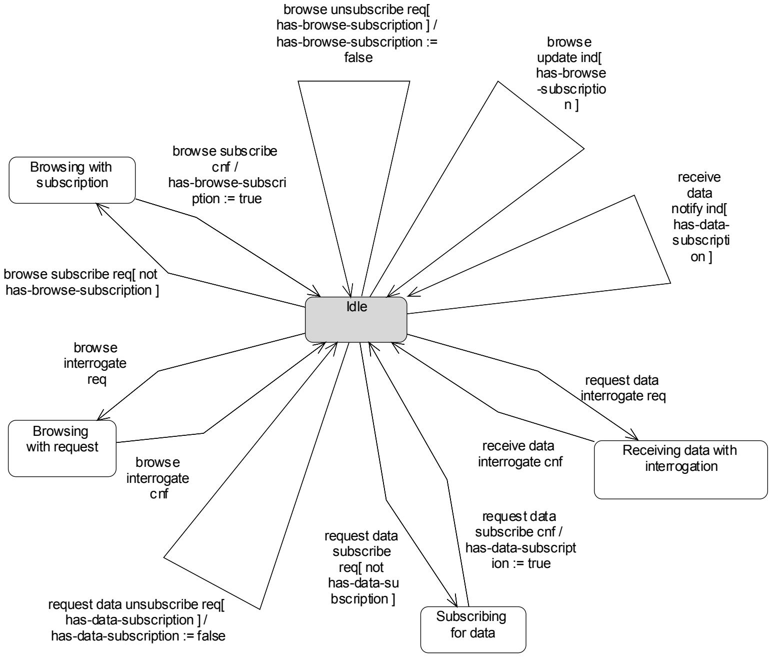
6.2.5 Relations among service primitives

Relations among service primitives fall into two main categories: local and remote. Local (to a service user) relations define possible sequence of service primitives performed by a single service user. The service primitives and their local relations define the service behavior at a single SAP. Remote relations define possible sequence of service primitives performed at different SAPs. The service primitives and their remote relations define the complete service behavior. In the next sections, we discuss relations in the following order: local relations for the monitor (M-SAP), local relations for the instrumentation (I-SAP), and remote relations.

Relations local to the monitor

Figure 6-10 describes all possible sequences of service primitives at the M-SAP using an UML state-chart diagram.

Figure 6-10 M-SAP state chart diagram


browse unsubscribe req[has-browse-subscription] / has-browse-subscription := false
browse update ind[has-browse -subscriptio n]
Browsing with subscription
browse subscribe cnf / has-browse-subscri ption := true
receive data notify ind[has-data- subscripti on]
browse subscribe req[not has-browse-subscription]
Idle
browse interrogate req
request data interrogate req
Browsing with request
browse interrogate cnf
receive data interrogate cnf
Receiving data with interrogation
request data subscribe cnf / has-data-subscript ion := true
request data subscribe req[not has-data-su bscription]
request data unsubscribe req[has-data-subscription] / has-data-subscription := false
Subscribing for data

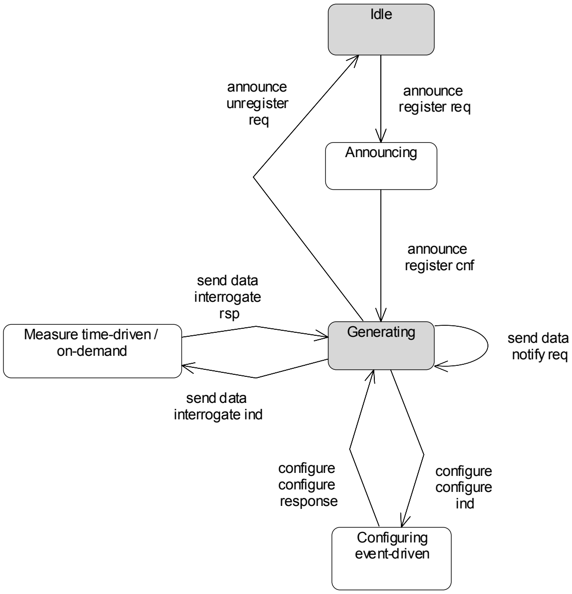
To reduce the number of states, we introduce the “has-browse-subscription” and “has-data-subscription” global state variables, which indicate whether a monitor has established a browsing subscription and a subscription for monitoring data respectively. Note that we permit only one browsing subscription and one data subscription per monitor.

From the “Idle” state, a monitor can go to four other states and back to “Idle”: subscribe for availability information (“Browsing with subscription” state), interrogate for availability information in a request/response style (“Browsing with request” state), subscribe for monitoring data (Subscribing for data” state), and request monitoring data in a request/response style (“Receiving data with interrogation” state).

Relations local to the instrumentation

Figure 6-11 shows all possible sequences between service primitives at the I-SAP using an UML state-chart diagram.

Figure 6-11 I-SAP state chart diagram



An instrumentation starts in the "Idle" state, in which it cannot generate monitoring data and the GMS does not "know" about this instrumentation yet. From the "Idle" state the instrumentation can move to the "Announcing" state, in order to register with the GMS and provide availability information about the monitoring data it can produce. From the "Announcing" state the instrumentation can move to the "Generating" state in which it can actually generate monitoring data in two ways: by measuring values and packaging monitoring data upon an request from the GMS ("Measure time-driven/on-demand" state), or by configuring its sensors to detect particular events in the monitored application ("Configure event-driven"). In the case of configured sensors, the instrumentation can make a transition to the same state "Generating" by sending data using notification. From the "Generating" state an instrumentation can move to "Idle" again by unregistering from the GMS.

Remote relations

In this section we define the remote relations in the service behavior of the GMS. The behavior of the GMS concerns relations among primitives both of the M-SAP and the I-SAP. We use time sequence diagrams to model the remote relations. We prefer using time sequence diagrams instead of a state because, this way we avoid dealing with too many states. The following Figure 6-12 shows all possible sequences among primitives.

Figure 6-12 Time sequence diagrams

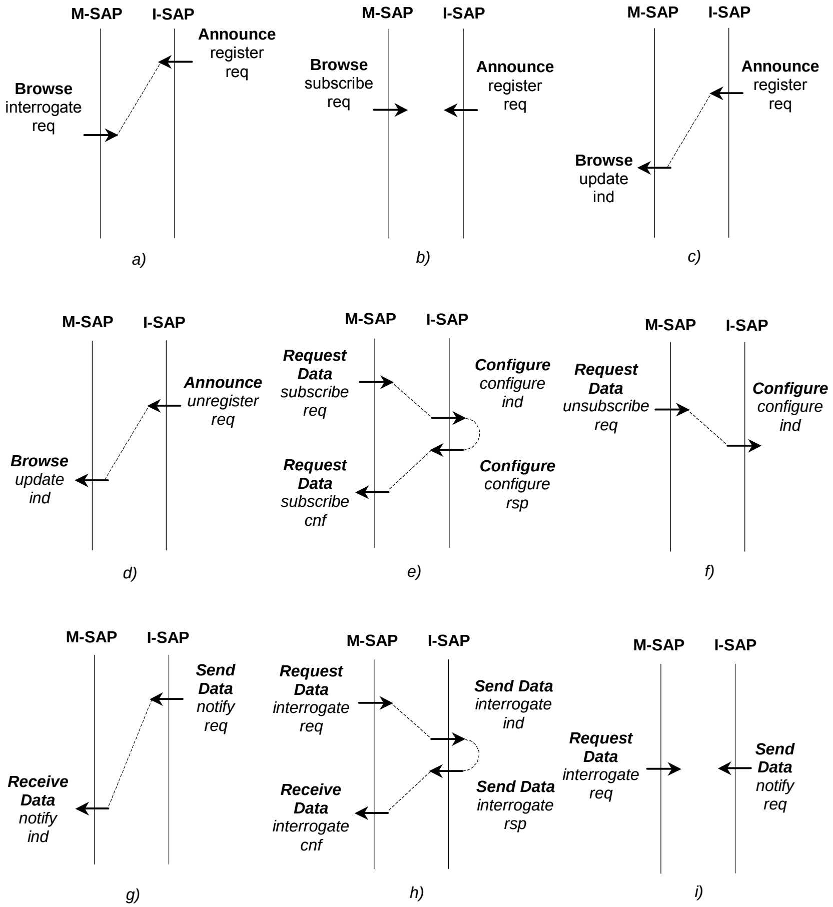


Diagram *a)* states that a monitor can interrogate the GMS for availability information only after the instrumentation has registered with the GMS. This allows a monitor to request availability information whenever necessary. In contrast, in diagram *b)* we define that subscribing for availability notifications may occur independently to any registering of availability information. This allows monitors to receive timely notifications about the registration of new availability information (diagram *c*) or about the un-registration of availability information (diagram *d*) by the instrumentation. Diagram *e)* shows that a monitor can subscribe for monitoring data and that a successful subscription results in configuring the instrumentation so that it generates only information that the monitor requires. Similarly, when a monitor unsubscribes, the GMS reconfigures the instrumentation so that it does not generate unnecessary monitoring data (diagram *f*). When the instrumentation (previously configured) sends data to the GMS, the GMS notifies the monitor (diagram *g*). Diagram *h)* describes that a monitor can interrogate the GMS for monitoring data whenever necessary. An interrogation results in the instrumentation making measurements, packaging and returning monitoring data, which the GMS sends to the monitor. Diagram *i)* shows that interrogation and notification occur independently, so that monitors can both interrogate for monitoring data and subscribe for notifications.

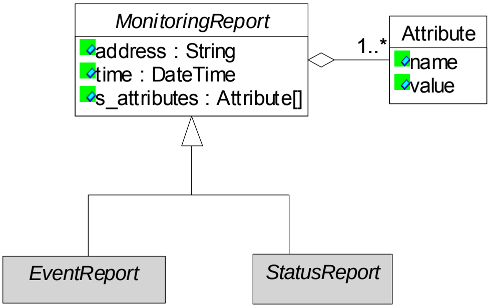
6.2.6 Service primitive parameters

In the previous section we have discussed only the role of service primitive parameters. Nevertheless, some of these parameters require a more detailed definition of their structure. In this section we discuss the structure of the following parameters: *monitoring data*, *specification of availability*, *search criteria*, *specification of interest*, *data selection criteria*, *configuration specification*.

Monitoring data

The GMS model we defined in section 6.1 represents the monitored application as a collection of instrumentation instances, each producing monitoring data as a sequence of individual monitoring reports. The "monitoring data" parameter of GMS service primitives contains as a value a list of monitoring reports. Figure 6-13 shows the structure of a monitoring report using an UML class diagram.

Figure 6-13
MonitoringReport class



The MonitoringReport has three attributes: *time*, *address* and *s_attributes*. The address attribute uniquely identifies the instrumentation instance that produced the monitoring report. For example, in a particular GMS implementation, the address attribute may contain the IP address of the network host on which the instrumentation generated the monitoring report. The time attribute represents the time of generation of a monitoring report. For example, the time attribute may contain a timestamp representing the moment of generation, measured using the host computer clock. The time and address attributes allow the GMS to filter out monitoring reports for delivery to monitors. We discuss filtering later when we define the *specification of availability* and *search criteria* parameters. The s_attributes attribute contains a list of one or more specific attributes. Subclasses of the MonitoringReport class may define different sets of specific attributes. The GMS models a specific attribute in a generic way using the Attribute class. An attribute has a *name* and a *value*.

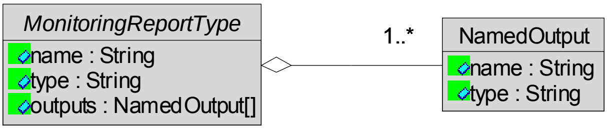
A MonitoringReport has two subclasses: EventReport and StatusReport. The EventReport class represents event reports and the StatusReport class represent status reports. We explained the differences between event and status reports in Chapter 2. The GMS uses the MonitoringReport and its two subclasses EventReport and StatusReport to disseminate monitoring data to monitors in a generic way.

In order to build specializations of the GMS for particular monitoring and monitored applications (stage two of the methodology) designers would have to subclass EventReport or StatusReport to define event and status reports for the particular specialization. These new event and status classes share the time and address attributes, but define their own specific attributes.

Specification of availability

The "specification of availability" parameter contains as a value a list of monitoring report types. Figure 6-14 shows a UML class diagram for a monitoring report type.

Figure 6-14 MonitoringReport-Type class



The instrumentation uses one MonitoringReportType object to describe one particular MonitoringReport class that it can generate. The MonitoringReportType class has three attributes: *name*, *type* and *outputs*. The *name* attribute represents the name of the MonitoringReport subclass that the instrumentation wants to describe. The *type* attribute holds as a value either the string "event" for an EventReport subclass or the string "status" for a StatusReport subclass. The *outputs* attribute contains a list of NamedOutputs. A NamedOutput models in a generic way the output of a sensor in the instrumentation. A NamedOutput has a *name* and *type* attribute, where type represents the data type of the output value. For each

NamedOutput we establish a one-to-one correspondence with a specific Attribute of the MonitoringReport subclass, that the instrumentation can generate.

At the I-SAP, the instrumentation uses instances of the MonitoringReportType class to compose a “specification of availability” list describing the concrete event and/or status types it can generate.

At the M-SAP, the GMS composes a “specification of availability” list of MonitoringReportType objects previously registered with the GMS to satisfy “search criteria” requested by monitors.

Search criteria

The “search criteria” parameter contains a search expression composed by a monitor that interrogates the GMS about availability information. The GMS uses this search expression to select MonitoringReportType objects that represent event and status types relevant to the interrogating monitor. In Figure 6-15 we define a Simple Type Search Language (STSL) using the Extended Backus-Naur Form (EBNF) [Marc86]. Monitors use the STSL to compose search expressions. Note that we enclose one-symbol terminals between “”, and we depict other terminals in bold font. Each statement ends with “;”.

Figure 6-15 EBNF for the STSL language

search_criteria_expression ::= **selecttype** select_expression [**where** attribute_predicate];
select_expression ::= **event** | **status** | “*” ;
attribute_predicate ::= [unary_predicate_op] attribute_expression { binary_predicate_op attribute_expression };
unary_predicate_op ::= **not** ;
binary_predicate_op ::= **and** | **or** ;
attribute_expression ::= attribute **match** pattern_value ;
pattern_value ::= string | regular_expression ;
attribute ::= **name** | **output “.” name** | **output “.”** output_name **“.” type** ;
output_name ::= string;

The STSL distinguishes between events, statuses, or any reports (using the “*” to indicate “any”). The *attribute_predicate* consists of attribute expressions using the logical AND, OR and NOT operators. An *attribute_expression* consists of an *attribute* and a *pattern_value* that this attribute needs to match. An *attribute* can refer to three different things: the name of the MonitoringReportType, a NamedOutput name or a NamedOutput type. The *pattern_value* can represent a string value or, for example, an UNIX style regular expression. The system selects a MonitoringReportType object when its attributes match the pattern. In the following lines we present two examples of a search expression:

selecttype event where name match “ObjectCommunicationEvent”
selecttype status where output.“CPU load”.**type match** “integer”

The first example selects the MonitoringReportType object that represents an event type with name “ObjectCommunicationEvent”. The second example selects all MonitoringReportType objects that represent statuses and have a specific attribute called “CPU load” of type “integer”. We consider the definition of a more complex search criteria language out of the scope of this thesis.

Specification of interest

The “specification of interest” parameter contains a description of a monitor’s runtime requirements for monitoring data. Since subscriptions allow timely delivery through notifications when the instrumentation generates new event reports, we define that monitors can use this specification only to receive event reports. We allow monitors to request status reports by explicit interrogation using a “data selection criteria” (see next section). In Figure 6-16 we define a Simple Specification of Interest Language (SSIL) using EBNF. Monitors compose specifications of interest using the SSIL.

Figure 6-16 EBNF for the SSIL language

specification_of_interest ::= **subscribe** event_expression { “;” event_expression } ;
event_expression ::= name [**filter** filter_expression] ;
name ::= string | pattern ;
filter_expression ::= [unary_predicate_op] attribute_expression { binary_predicate_op attribute_expression } ;
attribute_expression ::= time_expression | address_expression | specific_attribute_expression ;
binary_predicate_op ::= **and** | **or** ;
unary_predicate_op ::= **not** ;
time_expression ::= **time** time_op time_value ;
time_op ::= **before** | **after** | **exactly** ;
address_expression ::= **address match** (string | pattern) ;
specific_attribute_expression ::= specific_attribute { relation_op (specific_attribute | value) } ;
specific_attribute ::= **s_attribute** “.” specific_attribute_name ;
relation_op ::= “>” | “<” | “=” | **match**;
specific_attribute_name ::= string ;

A specification in SSIL contains a list of event expressions. An event expression may contain a filter part, which defines constraints on the values of the event’s time, address, and specific attributes. Monitors can combine different constraint expressions in a filter using the logical AND, OR and NOT operators. We leave the actual date format represented with the time_value non-terminal, deliberately undefined by this service definition. We leave to designers to determine the date format (and hence the “ < ”, “ > ”, “ = ”, and match operators on dates) at implementation time. Using expressions on specific attributes, monitors can define constraints on the instances of concrete event types in a generic way. In the following paragraph we present an example specification of interest:

subscribe "CommunicationEvent" **filter time after** "20:00PM 20-05-2003" **and time before** "05:00AM 21-05-2003" **and address match** "130.89.*" **and s_attribute**."CPU load" < 50

In this example a monitor instructs the GMS to subscribe that monitor to events of type "CommunicationEvent", occurring between 20:00PM on date 20-05-2003 and 5:00AM on the following day, originating from a subnet starting with "130.89.", with a specific attribute "CPU load" having value less that 50. One can find examples of more complex languages for specifying monitor interests in [Samani95] and [Shaer98].

Data selection criteria

We use this parameter in two primitives, at the M-SAP and at the I-SAP.

At the M-SAP, the monitor uses the data selection criteria to instruct the GMS to perform measurements, which result in status reports. The "data selection criteria" parameter contains a list of names of StatusReport subclasses that the GMS can currently generate. The monitor can obtain these names from the availability information in the GMS. The GMS takes the list of StatusReport names and forwards it to the instrumentations, which can generate instances of these StatusReport types. A monitor can include in the data selection criteria any name of a StatusReport subclass available for measurement with the GMS (i.e., StatusReports described in the specification of availability). Hence, a monitor can request a measurement from any instrumentation instance registered with the GMS.

The monitor can also use the data selection criteria to select matching monitoring data (including both events and statuses) accumulated so far in the GMS. We consider languages such as SQL [SQL99] suitable for selecting large amounts of data from a storage within the GMS. Although our design does not prohibit this use of the "data selection criteria" parameter, in this service definition we do not use it.

At the I-SAP, the GMS uses the data selection criteria to instruct measurement from the instrumentation. The parameter has the same structure as when used at the M-SAP. In contrast to the M-SAP, the data selection criteria at the I-SAP can contain only names of StatusReport subclasses that the particular instrumentation can generate.

Configuration specification

The “configuration specification” parameter contains a list of names of EventReport subclasses. These names represent event reports that a monitor has previously subscribed for and has included their names in its specification of interest. The GMS uses this parameter to tell the instrumentation to start producing the particular type of events (in the implementation, this may involve switching of certain sensors in the instrumentation on or off).

6.3 Definition of the GMS software architecture

This section presents a software architecture for a GMS that implements the service defined in the previous section. The architecture consists of a logical decomposition and a physical decomposition.

6.3.1 Logical decomposition

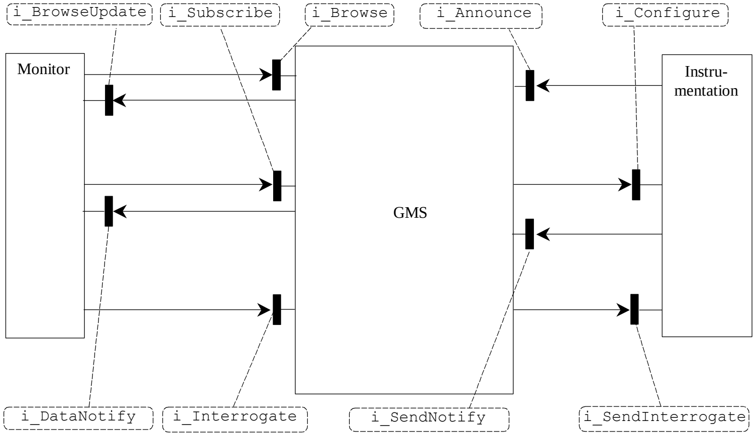
We decompose the GMS in four steps. In the first step we refine the GMS service definition into service-level interfaces that the GMS, the monitor and the instrumentation can offer to each other. We use this refinement in the subsequent steps to define which logical components of the GMS provide which interfaces. In the second step we decompose the functionality of the GMS into three components: Repository, Dissemination and Filtering. In the third step we further decompose the Dissemination component into four subcomponents. In the last step we specify the cooperative behavior of the logical components of the GMS using UML message sequence diagrams.

Service-level interface refinement

The service-level refinement consists of defining interfaces using the OMG Interface Definition Language (IDL) as a specification language. Each interface defines a number of operation signatures that correspond to one or more service primitives. IDL allows the use of two types of operations: synchronous and oneway. Invocation of a synchronous operation has the semantic of an RPC call, which consists of a request part (the caller requests some service) and a response part (the callee returns a result of that service). In contrast, an invocation on an oneway operation only has a request part and does not require any response. In order to define interfaces from the GMS service, we need to determine how service primitives map to operation invocations. We map causally related *req* and *cnf* service primitives to a single synchronous operation. We also map causally related *ind* and *rsp* service primitives to a single synchronous operation invocation. We map a single *req* or a single *ind* service primitive to a single oneway operation.

We use the following criteria to group operations into interfaces. Firstly, we reuse the structure of service elements introduced in the GMS service. Hence, an interface contains operations that map to service primitives from the same service element (with one exception – see below). Secondly, we group primitives based on the initiative for their provisioning. A single interface contains operations provided only by a single entity, such as the GMS, the instrumentation or the monitor. Figure 6-17 shows the interface refinement.

Figure 6-17 Interface refinement for the GMS service



For the “Browse” service element we define two interfaces: i_Browse and i_BrowseUpdate. The i_Browse interface defines the operations that a monitor calls on the GMS. The i_BrowseUpdate interface defines the operations that the GMS uses to notify a monitor about changes in the availability information. For the “Request Data” service element we define two interfaces: i_Subscribe and i_Interrogate. The i_Subscribe interface defines the operations that a monitor uses to subscribe for monitoring data. The i_Interrogate interface defines the operations that a monitor can use to interrogate the GMS for monitoring data. This interface contains the *interrogate* operation, which maps to two service primitives: the “Request Data interrogate req” represented by the invocation of the interrogate operation and the “Receive Data interrogate cnf” represented by the response of the interrogation operation. Although these two service primitives belong to different service elements we map them to one operation because we find this a convenient way to implement a request/response style of interaction. For the “Receive Data” service element we define the i_DataNotify interface. The i_DataNotify interface defines operations that the GMS uses to notify a monitor about new monitoring data. For the “Announce” service element we define i_Announce interface, therefore the i_Announce interface defines operations that the instrumentation uses to announce the available types of monitoring data. For the “Configure” service element we define the i_Configure interface. The i_Configure interface defines operations that the GMS uses to configure the instrumentation. For the “Send Data” service element we define two interfaces: i_SendInterrogate and i_SendNotify. The i_SendInterrogate interface defines operations that allow the GMS to interrogate an instrumentation for monitoring data. The i_SendNotify interface allows the instrumentation to notify the GMS about newly generated monitoring data.

In Appendix A, we present the detailed IDL specifications of these interfaces, along with comments explaining which service primitives correspond to which operations.

Functional decomposition of the GMS

We observe that the GMS service defines two basic types of GMS functionality: functionality related to the management of availability information, and functionality related to the dissemination of the actual monitoring data. Management of availability information includes on one hand the registering (and unregistering) of availability information by the instrumentation (the “Announce” service element) and on the other hand browsing of availability information by monitors (the “Browse” service element). Based on this observation we identify a Repository component that manages availability information in the GMS. The Repository component has the responsibility for storing and managing the access to the availability information in the GMS. The Repository component implements the i_Browse and i_Announce service-level interfaces and an internal (not part of the GMS service) interface ii_Repository (see Appendix A for the detailed specification of this interface).

From Chapter 2 we know that dissemination activities comprise the collection and delivery of monitoring data. Based on this we identify a Dissemination component. The Dissemination component has the responsibility to disseminate monitoring data. The Dissemination component uses the ii_Repository interface to validate specifications of interest and data selection criteria with the current availability information. The Dissemination component implements the i_Subscribe and i_Interrogate service-level interfaces and an internal interface ii_DeliverEvent.

Processing activities involve among others filtering activities. Since the GMS allows a monitor to subscribe for monitoring data based on a specification of interest, the GMS needs to perform filtering of the monitoring data to make sure that monitors receive the requested monitoring data. Based on this we identify a Filtering component, which filters monitoring data. Note that filtering in the GMS represents a generic processing activity; by using the SSIL language the GMS does not need to interpret any (application domain) specific information in order to filter out monitoring data. The Filtering component has the responsibility to determine the relevancy of new event reports that the instrumentation sends to the GMS. The Filtering component uses the ii_DeliverEvent to pass relevant filtered monitoring data to the Dissemination component. The Filtering component implements the i_SendNotify service-level interface and the internal ii_ConfigureFiltering interface. The Dissemination component uses the latter interface to configure the filtering mechanisms when monitors make or remove subscriptions.

Figure 6-18 depicts the relations between the GMS interfaces and the three components we introduced.

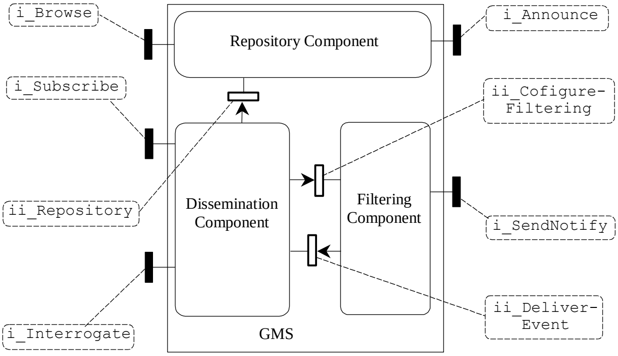
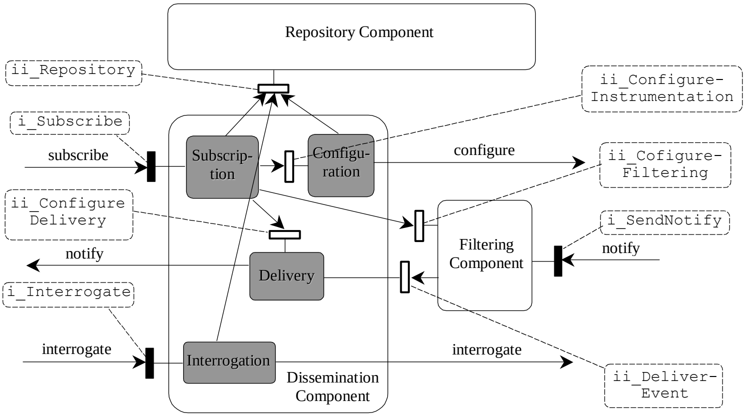


Figure 6-18 Functional decomposition of the GMS

Functional decomposition of the Dissemination component

The Dissemination component still provides relatively complex functionality, which it exposes through two service-level interfaces and one internal interface. Figure 6-19 shows how we further decompose the Dissemination component.

Figure 6-19
Structure of the dissemination component



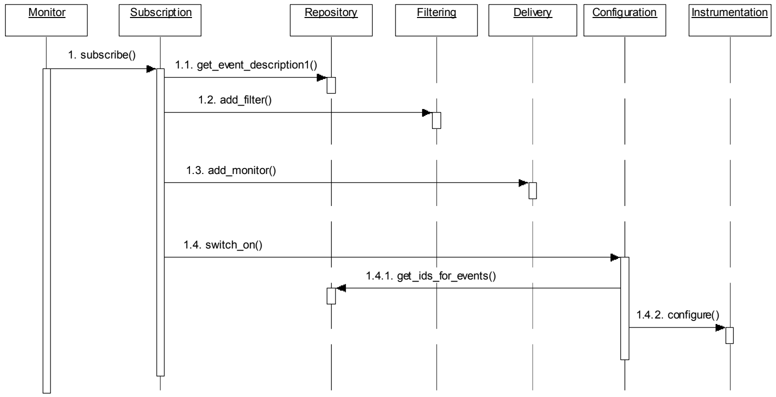
The Dissemination component consists of four sub-components: Subscription, Configuration, Delivery and Interrogation. The Subscription component handles all monitor subscriptions. The Configuration component has the responsibility for configuring the event report generation in the instrumentation. The Delivery component holds the responsibility for the delivery of (filtered) event reports to the appropriate monitors. The Interrogation component holds the responsibility for handling monitor interrogations for status reports. In Appendix A, we present the internal interfaces that specify the operations offered by the different sub-components of the Dissemination component.

Cooperative behavior of the GMS logical components

We define the cooperative behavior of the GMS logical components using four scenarios of use: “monitor subscription”, “monitor un-subscription”, “notification about new monitoring data” and “interrogation for monitoring data”. For each scenario we provide a UML sequence diagram that defines the sequences of operation invocations among components interacting in this scenario. For simplicity we do not discuss error-handling scenarios. We also do not consider scenarios regarding registering and browsing of availability information, because they do not require interaction among the GMS components.

Figure 6-20 shows the sequence of operation invocations among GMS components and the instrumentation that occur when a monitor issues a subscription request.

Figure 6-20 Monitor subscription scenario



When a monitor subscribes for monitoring data, it submits a specification of interest written in SSIL to the Subscription component. The Subscription component processes the specification and contacts the Repository component to extract the event report type availability information indicated in the specification. Then the Subscription component passes the filter expressions from the specification to the Filtering component. The Filtering component adds the filter expression to its filtering mechanism. Then the Subscription component contacts the Delivery component to add the subscribed monitor to the event delivery mechanism. The Subscription component then instructs the Configuration component to switch on the production of relevant events in all instrumentation instances that support the event report types indicated in the monitor's specification of interest. The Configuration component contacts the Repository component to obtain the identifiers (e.g. object references) of instrumentations that support the required events. The Configuration component uses these identifiers to configure the instrumentation to emit events of the desired types.

Figure 6-21 shows that GMS handles un-subscribing in a similar to subscribing way.

Figure 6-21 Monitor un-subscription scenario

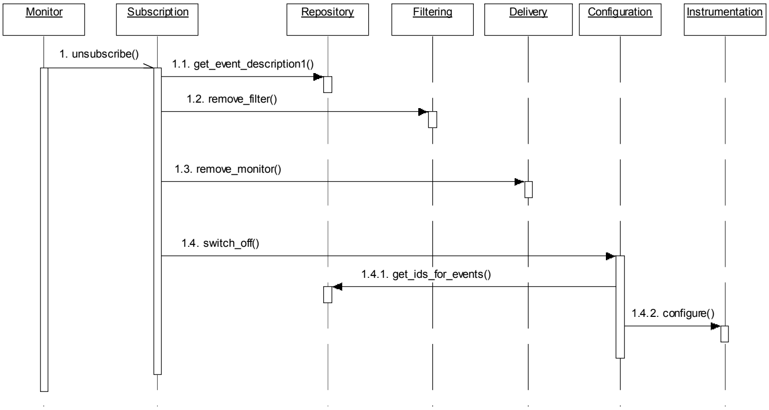
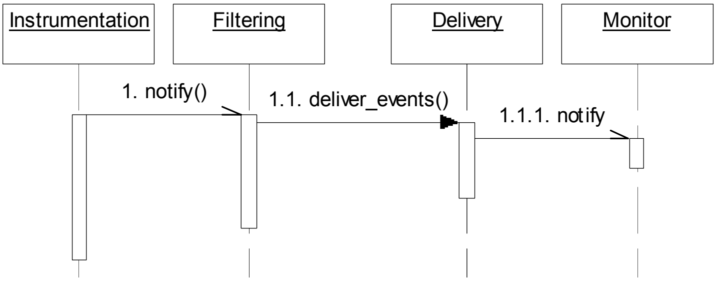


Figure 6-22 shows the sequence of operation invocations among GMS components and monitors when a notification about new event report(s) comes from the instrumentation.

Figure 6-22 Notification about new monitoring data



The instrumentation sends an event report to the Filtering component. The Filtering component determines the relevancy of the event to any monitor by evaluating the accumulated filter expressions for this event report type. The Filtering component also determines which monitors have interest in this event report. In the case the Filtering component finds the event report relevant, it sends the event to the Delivery component, which notifies appropriate monitors about the event.

Figure 6-23 shows the sequence of operation invocations among the GMS components and the instrumentation when a monitor interrogates the GMS for measuring statuses.

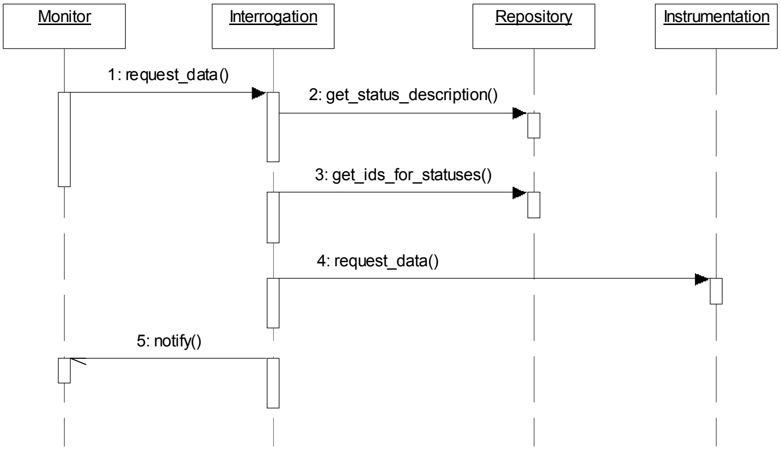


Figure 6-23
Interrogation for monitoring data

A monitor makes a request to the Interrogation component by supplying a list of status report type names. The Interrogation component first contacts the Repository component to validate the names. Then the Interrogation component gets from the Repository component the identifiers (e.g. object references) of all instrumentation instances that support the requested status report types. It then makes requests for measurements to the instrumentation, composes a list of results and sends it to the requesting monitor.

6.3.2 Physical decomposition

The GMS logical decomposition does not explicitly take into account distribution aspects. The way the GMS handles distribution aspects however, can have a great impact on the system scalability, performance and overhead. The GMS should efficiently utilize the communication and computing resources needed for collecting, filtering and delivering monitoring data to monitors.

In [Shaer98], Al Shaer argues that a hierarchical architecture allows the users to configure the monitoring system optimally with respect to unwanted communication overhead. Furthermore, designers can deploy a monitoring system with a hierarchical architecture consistently with the hierarchically organized administrative policies that we often find in large distributed environments. These policies define how a distributed system should deal with issues such as available resources (e.g. bandwidth), heterogeneity, interoperability, security and failure. For these reasons we choose a hierarchical architecture as a basis for the physical decomposition of the GMS.

We develop the physical decomposition in three steps. In the first step we define a hierarchical distribution model that we use to structure the distribution aspects of a distributed environment, in which we want to deploy the components of a monitoring system. In the second step we use the hierarchical distribution model to define a hierarchical agent-based architecture for the GMS. In the third step we map the agent-based architecture onto the logical decomposition of the GMS.

Hierarchical distribution model

The hierarchical distribution model (Figure 6-24) models a distributed environment using three types of regions: *co-location regions*, *locality regions*, and *administrative domain regions*.

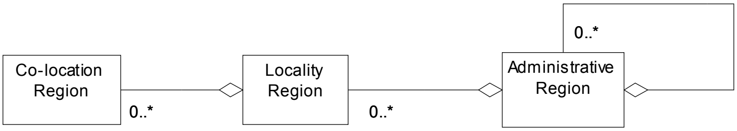


Figure 6-24 The hierarchical distribution model

A co-location region represents a single operating system process in the execution environment. We define the co-location region as the minimal set of concerns with respect to complexity of distribution aspects. A co-location region may contain one or more components of a monitoring system; we call these components *co-located*. Co-located components share the same operating system process, and hence the same execution environment and the same host. Co-located components may also share the same development technology, including runtime libraries, versions of APIs that the components use, etc. We assume that a physical component belongs to precisely one co-location region at any moment in time. In general the communication infrastructure between co-located components has a high degree of reliability (e.g. in-memory communication), security (the same host, the same process, the same access rights), performance (e.g., the internal architecture of the OS and the hardware platform determines the communication speed), and has low economical cost (as compared to, e.g., the cost for bandwidth usage in networks).

A locality region represents a single physical host. A locality region contains (aggregation relation) one or more co-location regions. Hence, non co-located components may still share the same locality region if they reside on the same host. *Local* components that share the same host. Non co-located but local components do not necessarily share the same development technology, including runtime libraries, versions of APIs that the components use, etc. For this reason, communication between local non co-located components may experience synchronization, interoperability, performance and security issues. Consider as an example, the communication between different processes running with different access rights to host's resources.

An administrative domain region represents a collection of hosts and a network that connects them within a single organization. An administrative domain region contains (aggregation relation) one or more locality regions corresponding to the hosts in the organization. Hence, non local components may still share the same administrative domain region if they reside on hosts in that region. An administrative domain region may also contain (aggregation relation) other administrative domain regions. Within an administrative domain region, management policies regulate the type of hardware, software, communication devices, security and fault issues. Non local components within the same administrative domain region run under the same policy but still may have to deal with issues such as communication delay and partial communication failures between the hosts they reside on. Components belonging to different administrative domain regions however do not share common administrative policies.

The so defined hierarchical distribution model allows us to categorize the communication in a particular deployment among the physical components of a monitoring system from the perspectives of reliability, security, and performance requirements. For example, we generally consider communication between co-located components secure, reliable and efficient, whereas communication between components belonging to different administrative domains may require the additional use of encryption, reliable communication protocols, and special bandwidth reservation protocols to guarantee communication performance. We define a hierarchical agent-based architecture that reflects this categorization in the way it handles communication within and across the different regions.

Hierarchical agent-based architecture

In the hierarchical agent-based architecture the GMS consists of *monitoring agents* that can communicate among each other. A monitoring agent represents a non-distributed (hence monolithic) physical component of the GMS. We use the hierarchical distribution model to define three types of monitoring agents: Co-located Monitoring Agent (CMA), Local Monitoring Agent (LMA), and Domain Monitoring Agent (DMA). Figure 6-25 shows the relationships between these types of agents.

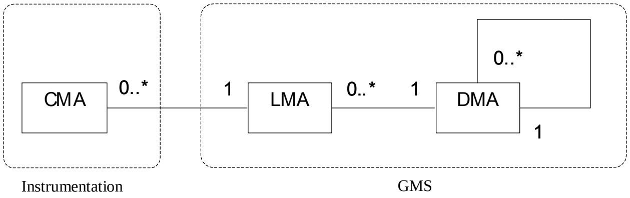


Figure 6-25
Relations among monitoring agents

A CMA represents a region of co-location. The CMA co-locates with a monolithic (i.e. non-distributed) part of the monitored application. During monitoring, the CMA interacts with the monitored application part in an efficient, but possibly technologically proprietary and domain dependent manner. We consider the CMA the part of the instrumentation that communicates with the GMS. As such, the CMA has the responsibility to implement the GMS service-level interfaces of the instrumentation i_SendInterrogate and i_Configure. We consider the internal structure of the CMA outside the scope of the GMS physical decomposition.

An LMA represents a region of locality. An LMA manages all CMAs on the same host. The LMA implements the service-level interfaces of the GMS involved in the interaction with the instrumentation: i_Announce and i_SendNotify. The interactions between CMAs and a LMA cross the boundaries of regions of co-location but do not leave the region of locality (the host). An LMA interacts with one DMA.

A DMA represents an administrative domain region. A DMA manages all LMAs in its region. The interactions between a DMA and its LMAs cross the boundaries of regions of locality but do not leave the administrative domain region. A DMA also interacts with DMAs of other administrative domain regions. The relations between DMAs cross the boundaries of administrative domain regions. Furthermore, the DMA implements the service-level interfaces of the GMS involved in the GMS interactions with monitors: i_Browse, i_Subscribe and i_Interrogate. Effectively, a DMA represents the GMS to all monitors deployed in the same administrative domain region.

Figure 6-26 illustrates an example deployment of a monitoring system according to the hierarchical agent-based architecture. In this example, the GMS comprises a collection of LMA and DMA instances, and the instrumentation comprises a collection of CMA instances.

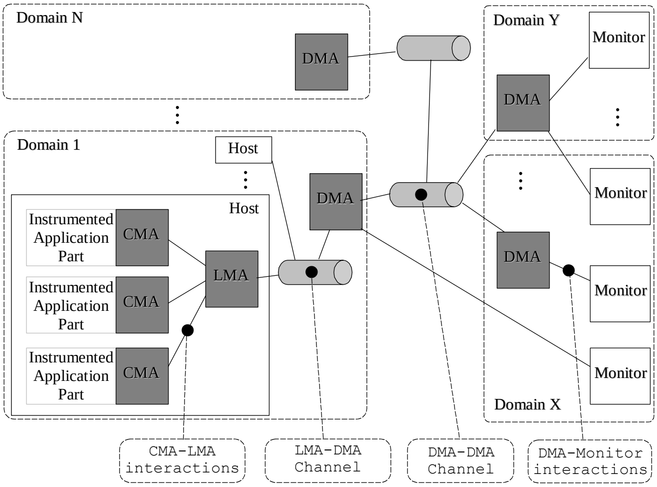


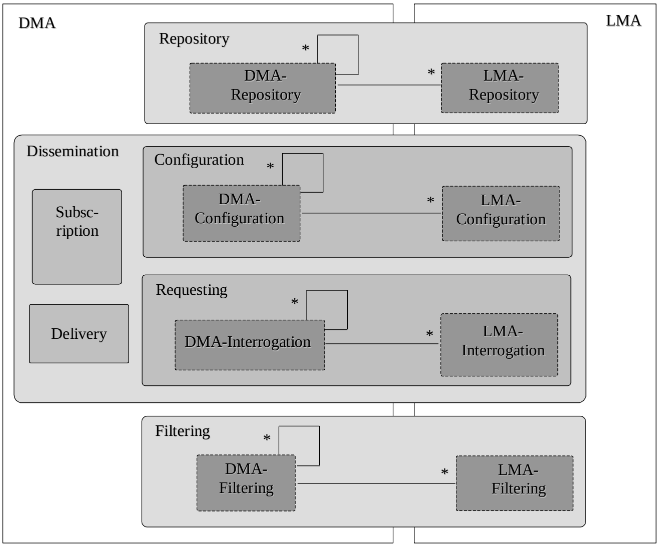
Figure 6-26
Physical decomposition of a monitoring system.

We consider an important aspect of the physical decomposition, how monitoring agents interact with each other and the environment to fulfill the functionality of the GMS. We have defined the CMA-LMA and the DMA-Monitor interactions as part of the GMS service definition. We mapped these interactions using synchronous and oneway operation invocations on the corresponding interfaces. LMAs communicate asynchronously with their DMA. DMAs also communicate asynchronously with other DMAs. We realize asynchronous communication using message exchange via communication channels. For example, the CORBA Notification Service [CNS] represents a technology that provides such channels. We choose channel-based communication because it promotes loose dependencies between communicating parties, and allows us to move the responsibilities for reliable and secure communication to the logic of the channel. Furthermore, channel-based communication supports multiple senders and multiple receivers of messages, where the channel administrators can take care of scalability issues of a growing monitoring system by reconfiguring the channels properly. In the GMS architecture, we consider two types of channels: LMA-DMA and DMA-DMA. An LMA-DMA channel connects the LMAs and the DMA within one administrative domain region and therefore does not need to deal with cross-domain administrative issues, such as security. A DMA-DMA channel connects the DMAs of different administrative domain regions and therefore may require additional capabilities, such as encryption.

Mapping functional components onto monitoring agents

In this section we map the GMS logical decomposition onto monitoring agents. Considering the structure of the logical decomposition, we further decompose the logical components into non-distributed physical components and we assign these components to the DMA and LMA monitoring agents (Figure 6-27). This effectively means that we define the internal structure of the monitoring agents by specifying their constituent components. At the level of the GMS hierarchical architecture we consider the CMA as a black box, because the CMA co-locates with the instrumentation and its internal structure typically reflects the structure of the instrumentation. Considering the behavior of the logical decomposition, with UML sequence diagrams we specify the interactions among its subcomponents. These sequence diagrams together with the sequence diagrams of the logical decomposition (see section 6.3.1) define the behavior of the GMS at the level of the physical decomposition.

Figure 6-27
Structural mapping of logical to physical components



We assign an instance of the Subscription and an instance of the Delivery logical component to each instance of a DMA. Hence a DMA can subscribe monitors and can deliver to them monitoring data. We decompose the Repository component into two types of non-distributed subcomponents: DMA-Repositories and LMA-Repositories. We assign to each DMA instance one instance of a DMA-Repository component, and to each LMA one instance of an LMA-Repository. A DMA can communicate with many LMAs, therefore a DMA's DMA-Repository component instance can communicate with all LMA-Repository component instances of the LMA's managed by this DMA. Furthermore, a DMA can also communicate with other DMAs, therefore a DMA's DMA-Repository can communicate with the DMA-Repositories of the other DMAs. We make the decomposition for the Configuration, Filtering and Requesting logical components in a similar way. This effectively means that we distribute the logic of the Repository, Configuration, Filtering and Interrogation components among instances of monitoring agents.

In the next sections we describe the internal behaviour of each of the logical components in terms of the external behaviour of its sub-components and the interactions among these sub-components.

The Repository component

The Repository component manages availability information in the GMS. Its function consists of allowing instrumentation instances to register/unregister availability information with the GMS, and monitors to browse and search the availability information. The following three use scenarios define the internal behavior of the Repository component: a CMA (representing an instrumentation instance) registers with the GMS, a CMA unregisters with the GMS, and a monitor browses the availability information.

In the first scenario (Figure 6-28), a CMA registers with its LMA by passing to it a specification of availability.

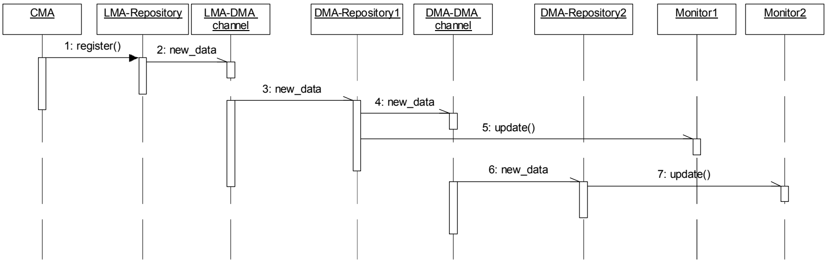


Figure 6-28 A CMA registers with the GMS

An LMA has the responsibility for passing this specification to its DMA. The LMA-Repository in a LMA keeps a view of the availability information of all CMAs on the same host. The DMA-Repository sends new availability information to all other DMAs so they can update their DMA-Repositories. Every DMA-Repository receives information from other DMAs, this way maintaining a global picture of the monitoring data available in the whole system. Each DMA sends notifications to subscribed monitors (types subscriptions) about changes in the availability information.

The GMS processes the unregistration of a CMA and the removal of the associated availability information in a similar manner as registration (Figure 6-29).

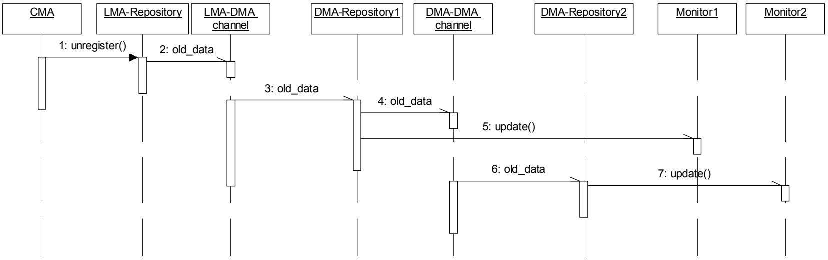


Figure 6-29 A CMA unregisters with the GMS

In the second scenario, a monitor contacts a DMA in order to browse the availability information. The corresponding DMA-Repository component handles the browsing request locally and returns a result according to its current global view on availability information, hence it does not need to interact with other components in this case.

The Configuration component

The Configuration component has the responsibility to configure event report generation in the instrumentation as a result of subscription/unsubscription of monitors with the GMS. The following two scenarios define the internal behavior fo the Configuration component: the GMS handles a subscription, and the GMS handles an unsubscription.

In Figure 6-30, the Subscription component contacts the DMA-Configuration component with a “switch on” request as a result (we do not depict this part) of a subscription initiated by a monitor with the GMS.

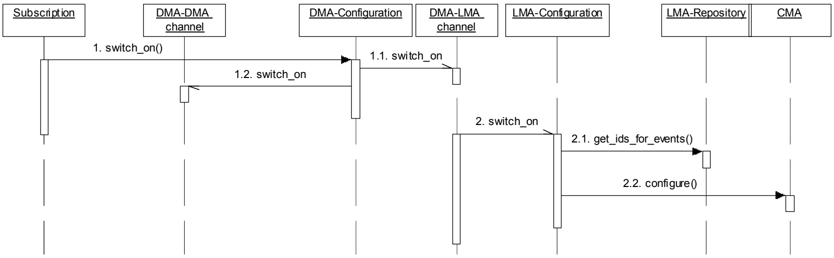


Figure 6-30 The GMS handles a subscription

The DMA forwards the "switch on" request to its LMAs by sending a message to its DMA-LMA channel, and to the DMA-Configuration components of other DMAs by sending a message to the DMA-DMA channel. A LMA processes the request by contacting the LMA-Repository to determine the relevant CMAs and then configures each CMA for event generation. A DMA-configuration component handles configuration messages from other DMAs in a similar way.

The system handles "switch off" requests similarly to "switch on" requests (Figure 6-31).

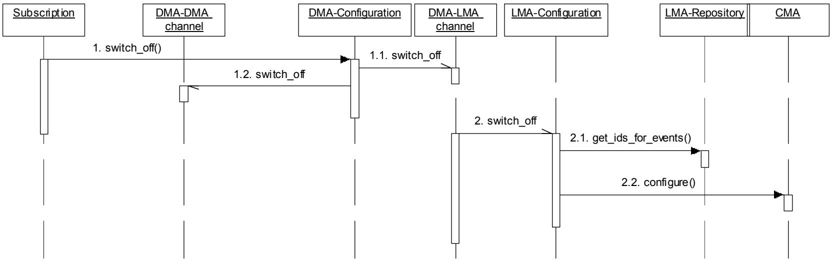


Figure 6-31 The GMS handles an un-subscription

The Interrogation component

The Interrogation component allows monitors to interrogate the GMS for status reports. An interrogation results in the measurement of values by the instrumentation and the generation of status reports. The following scenario defines the internal behavior of the Interrogation component: the GMS handles a request for status reports.

In Figure 6-32 a monitor makes a request for status reports.

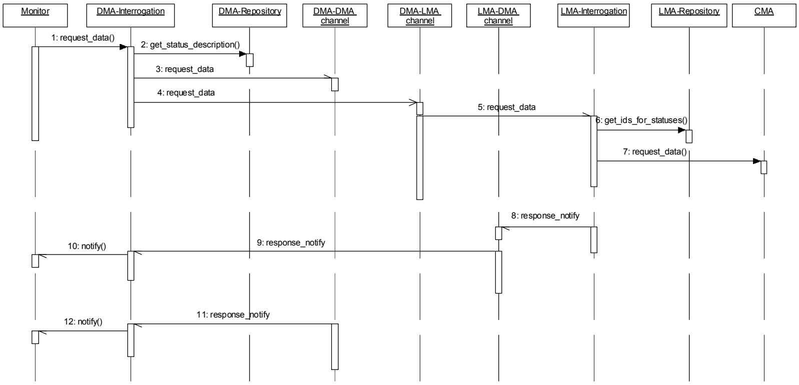


Figure 6-32 The GMS handles a request for status reports

The DMA-Interrogation component first calls the DMA-Repository to validate the request and then forwards it to other DMAs and to its LMAs by sending a “request data” message to the corresponding channels. When an LMA receives a “request data” message, it contacts its LMA-repository to get the ids for those CMAs registered with this LMA that can produce the status reports requested in that message, and then the LMA requests these status reports from CMAs that can provide them. When a CMA receives a request for a status report it performs the necessary measurements and returns the resulting status report to the LMA. The LMA sends the combined response containing the statuses to its DMA via the LMA-DMA channel. When a response comes from the LMA-DMA channel, the DMA checks whether this response corresponds to a request from any of the monitors subscribed to it. If the response does correspond, the DMA sends the response to the corresponding monitor. If the response does not correspond, the DMA sends the response to the DMA-DMA channel. When a response comes from the DMA-DMA channel, a DMA-Interrogation component determines whether it matches any pending request from its subscribed monitors, if yes, it sends the response back to the requesting monitor, if no, it ignores the response message. When a DMA receives a “request data” message from the DMA-channel, it processes it in a way similar to a request coming from a monitor subscribed with this DMA, but when a response comes back, the DMA sends the response directly to the DMA-DMA channel.

The Filtering component

The Filtering component has the responsibility to filter event reports coming from the CMAs. The following three scenarios define the internal behavior of the Filtering component: the GMS adds a filter (new subscription), the GMS removes a filter (un-subscribing), and the GMS processes an event report.

In Figure 6-33, the Subscription component contacts the DMA-Filtering component to add a new filter as a result of a new subscription.

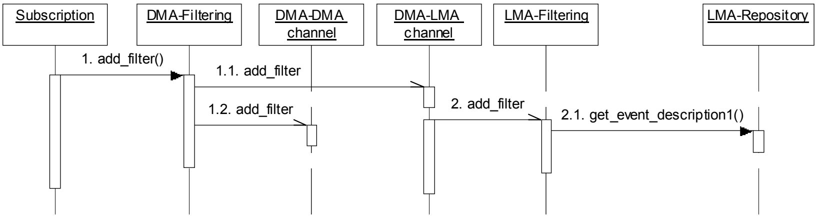


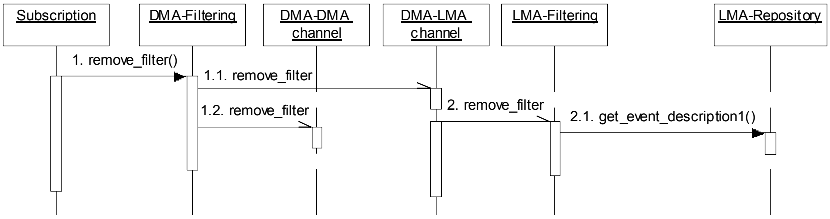
Figure 6-33 The GMS adds a filter (new subscription)

The Subscription component passes new filter to the DMA-Filtering component. The DMA-Filtering component maintains a view on all filters for monitors subscribed with this DMA. When it receives a new filter, the

DMA-Filtering component adds it to that view and sends it to the DMA-LMA and the DMA-DMA channels. When an LMA receives a new filter, its LMA-Filtering component processes the filter in order to extract from it those parts that this LMA can satisfy (i.e., it can produce events matching that filter). It does this by contacting the LMA-repository to obtain information about event types supported by the CMAs currently registered with the LMA. The LMA-Filtering component then saves the processed filter in order to maintain a local (to that host) view of monitor demands. The DMA-Filtering component handles “add filter” messages from other DMAs in a way similar to the way it handles “add filter” messages from a Subscription component, except that the DMA-Filtering maintains a separate view (separate of the view of all filters coming from monitors directly subscribed to the DMA) of all filters that have arrived to this DMA-Filtering component via the DMA-DMA channel). This separation allows for efficient processing of new event reports arriving at a DMA-Filtering component, depending on where it comes from: the LMAs or other DMAs.

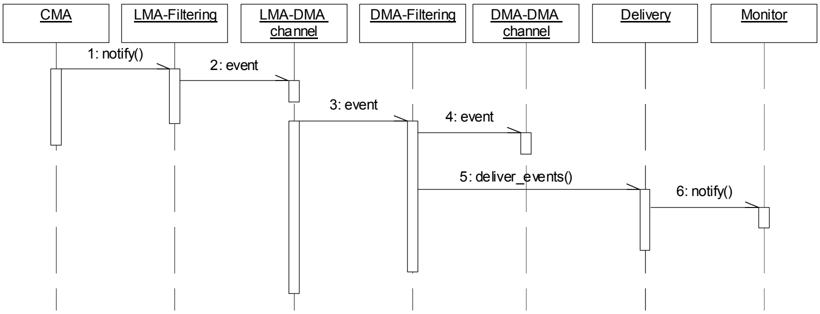
The DMA-Filtering component handles “remove_filter” requests from the Subscription component in a way similar to the previous case (Figure 6-34).

Figure 6-34 The GMS removes a filter (un-subscribing)



In Figure 6-35, the DMA-Filtering component processes a new event report. The CMA sends an event report to the LMA-Filtering component.

Figure 6-35 The GMS processes an event



The LMA-Filtering determines the relevancy of an event report to any monitor in the system according to its local filter view. In case relevant, it then sends the event report to its DMA-Filtering component via the LMA-DMA channel. The DMA-Filtering component determines the relevancy of the new event report to any of its subscribed monitors using its current view on their filters and sends the event report to the Delivery component which delivers the event report to those monitors. The DMA-Filtering component also determines the relevancy of an event report to any of the monitors subscribed to other DMAs in the system using its view on the rest of the system's filters, and sends a relevant event report to the DMA-DMA channel. When a new event comes to a DMA-Filtering component from the DMA-DMA channel, it determines its relevancy according to its view on its subscribed monitors and sends a relevant event to the Delivery component, which delivers it to its monitors. It does not have to check whether to send the even to the DMA-DMA channel since it got the event from that channel (and hence the other DMAs got it too).

6.3.3 Quality of service

We define the Quality of Service (QoS) of a system as a set of qualities regarding the cooperative behavior of system components that realize the service of the system [ISO98]. In order to satisfy its users, a system has to meet some QoS requirements.

In order to meet the QoS requirements of its users, an application may require certain guarantees on the resources it needs from its environment. In the case of a distributed environment, we distinguish two types of resources: *computational* and *communication*. Computational resources include processing power, memory, OS handles and any other resources that relate to the execution of monolithic (non-distributed) application parts. Communication resources include communication bandwidth and communication delay.

In terms of computational resources, an application may require guarantees on available processing time. In terms of communication resources, an application may require guarantees on available bandwidth for communication among its components.

In Chapter 1, we have identified a class of problems related to the monitoring system overhead, resulting from resource sharing in low-cost distributed environments. One possible solution to these problems requires adding QoS mechanisms to the distributed environment, where we consider a QoS mechanism, one that gives users guarantees for meeting their QoS requirements. For middleware-based systems, the middleware handles most (or all) resource allocations in the layers below the middleware layer. Hence, in middleware-based systems, the middleware can provide to the application the benefits of QoS mechanisms transparently (i.e., hidden by the middleware). We call these mechanisms, middleware QoS mechanisms. The work described in [Weg03] constitutes an example of middleware QoS mechanisms for computational resources. This work presents middleware-based solutions for load balancing. The work described in [Halt03] and carried out in [AMIDST] constitutes an example of middleware QoS mechanisms for communication resources. This work presents the Quality Provisioning Service (QPS) developed for CORBA middleware. The QPS allows designers that use the CORBA middleware as a distributed processing environment for their applications to provide QoS guarantees on the response time of remote operation invocations.

In the GMS we do not deal with QoS requirements. Since we use middleware technology to develop the GMS, designers who want to use our GMS have the possibility to add middleware QoS capabilities if they need them.

6.4 Implementation report

In this section we describe the effort we have made to implement a middleware-based proof-of-concept GMS prototype. This report discusses supported functionality, technological decisions, security, and reliability issues. In Appendix B we describe how to compile, configure, deploy and use the prototype.

6.4.1 Supported functionality

Within the FRIENDS and the AMIDST research projects [FRIENDS, AMIDST], we have created a prototype implementation of a GMS. We incrementally introduced the functionality described by the GMS architecture, in order to meet the resource limitations of the projects that supported our work. As a result of these limitations, the recent prototype of the GMS still does not support some of the functionality:

- The GMS prototype has support for event reports only. It does not support service primitives that relate to status reports. This limits the use of the GMS prototype to applications with an event-based monitoring model;
- The GMS prototype does not support the filter expressions in a specification of interest. This removes the ability of the GMS prototype to filter relevant monitoring data; instead it delivers to all monitors all event reports that the system generates;

- The GMS prototype uses communication channels only for the DMA-DMA communication. We do not use channels for the LMA-DMA and the DMA-LMA communication. Instead, we use CORBA synchronous and oneway operation invocations to implement the message exchange between LMAs and DMAs.

6.4.2 Technological decisions

In order to build a GMS prototype we have made the following decisions regarding implementation technologies:

- The Java technology as a programming platform. Java provides unprecedented *portability* among software and hardware platforms. At present, the Java platform supports a wide variety of server, desktop and embedded operating systems, and the standardization process within the Java community allows third party vendors to provide Java support for new operating systems and hardware platforms. Furthermore, the current state-of-the-art in compiler technologies has reduced the *performance gap* between Java-based solutions and other solutions based on compilation to native machine language;
- The CORBA object middleware as a distributed object computing environment. Some components of the GMS prototype may have to operate on systems where we do not have a JVM implementation or we cannot use Java for performance reasons. Re-implementing these components in a native environment may pose the problem of integrating this new implementation with the already existing Java-based components of the GMS. CORBA solves this because it allows portability across programming languages by providing to designers the *platform-independent* Interface Definition Language (IDL).

We implemented the GMS prototype with the Java 2 Development Kit version 1.3.x and above. We have used the JacORB [JACORB] implementation of the CORBA standard for Java. We chose JacORB, because its authors have made it available free for non-commercial use, and they consistently update it to conform to the latest CORBA specification updates. Note that because the CORBA standard supports interoperability among ORB implementations using the IIOP wire protocol, we can in principle, implement the various components of the GMS using any other ORB product, e.g., a C++ or a C ORB implementation to gain additional performance from natively compiled (i.e., optimized for a particular computer architecture) programs or an ORB implementation with real-time features.

CORBA standardizes several object services. Object services represent facilities that automate some of the most commonly performed tasks in a distributed environment. We use two of them: the naming and notification services. The Naming Service provides means for obtaining access (an object reference) to remote objects using explicit platform independent naming conventions. We resolve all explicit dependencies among remote objects of the GMS monitoring agents using the JacORB Naming Service implementation to obtain the necessary object references. The Notification Service allows for exchange of structured and unstructured data in a generic, asynchronous and scalable manner. We use the ORBacus [ORBacus] Notification Service to setup DMA-DMA channels.

The computer science research community has recognized open source and free software as a powerful tool for supporting scientific research. *Open source* software (with flexible licensing) encourages the research and development by allowing access to source code. *Free* software encourages the development of various applications (of that software) based on sharing and reuse of its source code. We built the GMS using various free (for non-commercial use) open source technologies. We believe that a basic enabling technology such as a GMS, should support the free and open source movement in order to promote the process of its refinement and evolution by letting the community contribute to its development. For this reason, we have released the prototype of the GMS under the LGPL license agreement [LGPL].

6.4.3 Security

The GMS prototype does not provide functionality explicitly related to security, such as built-in authentication or encryption. Using the GMS in a (secure) testing environment to monitor for locating and removing errors, does not require security features in the GMS. Therefore, it makes sense to consider security aspects only for uses of the GMS during the normal operation phase of the monitored application.

In general, we consider communication within administrative domain regions secure, and thus delegate this responsibility to domain administrators. We however consider communication between domains insecure in the general case. For DMA-DMA communication between DMAs belonging to different domains, designers need to consider providing additional security, e.g., encryption of communication to prevent eavesdropping.

6.4.4 Reliability

We consider the following reliability issues in the GMS monitoring system: reliable communication, tolerance to partial failure, and potential problems with the ordering of monitoring reports.

Reliable communication

The technologies used in the GMS prototype provide reliable communication among the physical components of the GMS – CORBA uses the reliable transport protocol of the Internet. This means that the GMS components eventually detect any failure in the communication. The GMS uses a fail-fast strategy: when one interaction between GMS components fails, the GMS assumes the network has become unavailable and terminates the components. Designers who need more elaborate behavior may extend our implementation appropriately.

Partial failure

A partial failure in the GMS we consider when a monitoring agent becomes inaccessible for the rest of the system, e.g., it either terminated abnormally or the communication network became unavailable.

We consider failure in a CMA equivalent to a failure of the instrumented application part, which results in loosing subsequent monitoring data from this part.

Failure in an LMA would result in subsequent loosing of monitoring data from a whole host, however the rest of the monitoring system continues to function.

Failure in a DMA would result in loosing monitoring data about a whole administrative domain region. The other domains (if any) however, would continue to provide monitoring data to their monitors. Designers can try to prevent loosing a whole domain because of a failed DMA, for example, by upgrading LMAs with domain consciousness, i.e., allow them to create new DMA instances that in turn will seek out connection to the other DMAs of the GMS.

Order of monitoring reports

Some monitoring applications may consider the delivery order of monitoring reports important. The Inter-ORB protocol that CORBA uses, takes care of the correct byte order during the communicating of operation invocations. CORBA however, does not guarantee that two separate monitoring reports sent with two separate operation invocations would always end up at a monitor in the same order (of sending). This problem results from a combination of various issues such as non-synchronized (and/or imprecise) computer clocks, communication delays, loaded networks, partial communication failures, and so on.

The GMS prototype does not provide any guarantees about the ordering of monitoring reports. By definition, the GMS service requires a timestamp from a physical clock present in each report, to use for filtering based on time constraints. Nevertheless, this physical clock has accuracy as good as the accuracy allowed by the protocol used for synchronizing computer clocks (e.g., NTP [NTP]). We choose to leave reordering functionality out of the GMS because of its processing cost in terms of overhead. Designers may choose to extend the GMS prototype with a logical clock system. We add a logical clocks system to the monitoring system during the specialization of the GMS for monitoring object and component communication.

Chapter 7

A system for monitoring distributed object and component communication

This chapter presents the design of a system for MOnitoring Distributed Object and Component Communication (MODOCC). The MODOCC system establishes the basis for building monitoring applications that analyze object and component communication behavior in CORBA-based Java applications. We design the MODOCC system following the design approach presented in Chapter 5.

The MODOCC systems consists of two parts: (a) GMS specialization for monitoring of object and component communication behavior and (b) instrumentation that allows monitoring of Java objects, CORBA objects, and the instances of components built with the Distributed Software Components (DSC) framework.

First, we prepare for the design by further refining the requirements that we have identified in Chapter 4. We use the design questions that we formulated in Chapter 5 to structure the refinement.

We specialize the GMS presented in Chapter 6 by following stage two of our design approach. The GMS specialization consists of two steps: (a) definition of a MODOCC Monitoring Model (MM) that helps us to specify explicitly what aspects of object and component communication we want to monitor, and (b) definition of a MODOCC data structure for the MODOCC MM. The MODOCC MM specializes the generic monitoring model of the GMS (Chapter 6, section 6.1.1) by defining types of events and their specific attributes. The MODOCC data structure specializes the GMS data structure by defining how a MODOCC monitoring report represents the event types and their specific attributes defined in the MODOCC MM.

We design the MODOCC instrumentation by following stage three of our design approach. The MODOCC instrumentation design for monitoring object and component communication consists of four steps: (a) design of sensors for Java objects, CORBA objects, and DSC component instances, (b) sensor placement, (c) design of instrumentation tools for Java, CORBA and DSC, and (d) a CMA (instrumentation) architecture.

In the end we discuss the performance of the MODOCC system prototype that we have built.

7.1 Requirement refinement

In this section, we refine and structure the requirements from Chapter 4 relevant to GMS specialization, by answering the design questions from Chapter 5.

Q1: *Why do we want to monitor object and component communication?*

In Chapter 3, we introduced object and component middleware. Object and component communication plays a central role in defining the behavior of a middleware-based application that consists of cooperating objects and/or component instances. We require monitoring of object and component communication for two reasons. Firstly, to allow the application designer to *visually* examine an execution in order to determine whether an application prototype *faithfully* implements its intended behavior specified at the level of communicating object and component instances. Secondly, to allow the application designer to *locate* and possibly remove implementation errors. We restrict the communication that we want to monitor to method calls and operation invocations.

For example, a monitoring system can visualize application behavior using a message sequence diagram showing the sequence of operation invocations among application objects during one particular execution of the monitored application. Using this diagram, a tester can better understand how the monitored application operates, can locate an error by associating it with a particular invocation, can detect undesirable sequences of invocations, and in general check the conformance of the monitored application behavior to its object-oriented design.

Q2: *What information do we need from a visualization of object and component communication?*

The monitoring system should uniquely identify *individual invocations*, provide information about the *order* among invocations, should uniquely identify the *objects* or *component instances* that participate in an invocation, and should allow inspection of *invocation parameters*. The visualization should inform about order in the terms of *causal precedence* among operation invocations (we introduced these terms in Chapter 2).

Furthermore, the monitoring system should have the capabilities to provide information *online*, so that designers can (potentially) observe operation invocations while the monitored application executes. Based on this online information, for example, designers can steer the monitored application in order to better expose some particular erroneous behavior. Note that we do not require application steering capabilities from the monitoring system.

Q3: *What monitoring data does a monitor require from the MSS?*

Visualization represents a presentation activity performed at a monitor (see the discussion on monitoring activities in Chapter 2). A monitor analyzes and visualizes the monitoring data coming from the MSS. Hence, the MSS should provide sufficient monitoring data to support visualization as discussed in the previous two questions.

An operation invocation represents a complex (potentially) distributed activity. We can view an operation invocation as a sequence of several non-distributed activities. This way of modeling an operation invocation allows us to deal with distribution, by measuring the progress of related non-distributed activities involved in an operation invocation. We consider the following non-distributed activities: initiation of an invocation at the location of the caller (object/component instance), receiving of an invocation at the location of the called, returning of a result at the location of the called, and the receiving of the result at the location of the caller (see Chapter 3 for details on the middleware mechanism that handles operation invocations). The successful completion of each one of these activities represents an event (see the modeling concepts defined in Chapter 2). The monitor requires to receive from the MSS *event reports* that represent these events.

The monitor needs *timestamps* associated with each event report so that it can restore the order (of occurrence) among the events belonging to a single invocation. This way the monitor can determine whether an invocation has completed (successful invocation) or not (which can mean an error or some special condition that designers have overlooked). A vector logical clock system allows keeping track of the progress of sequential *processes* in the monitored application by generating *vector timestamps* (described in Chapter 2). Vector timestamps allow restoring of the causal precedence relation among events. Timestamps generated with the help of an imprecise computer physical clock allow restoring of the temporal order among the events.

For the proper visualization of object and component instances and their involvement in communication, the monitor not only requires unique identifiers (to distinguish among them) but also requires information about their *lifecycle* and in particular their lifecycle activities. We consider two types of lifecycle activities - creation and destruction of an object or component instance. The monitor requires from the MSS event reports representing the events of successful completion of lifecycle activities. For example, a monitor can use the knowledge that objects can only communicate after creation and before destruction to detect attempts for communication with objects that the runtime environment has destroyed (which developers typically consider as an error).

Q4: *What monitoring data does the MSS require from the instrumentation?*

We do not require the MSS to perform any specific processing activities on monitoring data and therefore the MSS requires from the instrumentation the same monitoring data we discussed in the answer to the previous question. The MSS simply collects event reports from the instrumented application parts (the instrumentation) and delivers them to the monitors subscribed with it.

Q5: *What measurements does the instrumentation need to perform?*

In order to obtain the required monitoring data, the instrumentation needs to detect the events corresponding to activities happening during an operation invocation. We discussed event detection in Chapter 2, section 2.4.2. To detect the events necessary for monitoring an operation invocation, the instrumentation needs to install proper sensors. In Chapter 3, section 3.4.1, we discussed mechanisms for message reflection. For example, for the CORBA middleware, the interception points provided by the Portable Interceptors specification and the POA allow for detecting the completion of the activities we identified in the answer of question Q3.

The instrumentation needs to associate a timestamp with each event so that monitors can reason about their order (of occurrence). To provide causal precedence among events, the instrumentation needs to implement and maintain a vector clock system. When the instrumentation detects an event, it measures the current progress of its vector clock to generate a vector timestamp. Since vector timestamps provide partial order among events, the instrumentation also measures the physical computer clock to provide an absolute timestamp that allows absolute (though inaccurate) ordering of the events in the system.

When the instrumentation detects an event, it needs to identify uniquely the object(s), component instance(s) and the process (for the vector clock) concerning this event, so that the instrumentation can generate an event report that properly characterizes the event. For this the instrumentation measures the internal representations (e.g., names, pointers or hash codes) of objects, component instances, and processes, and uses these measurements to generate (if necessary) globally (for the whole distributed system) unique identifiers that allow monitors to distinguish among different objects, component instances and processes.

The instrumentation also needs to measure the values of parameters (input parameters and results) of an invocation at the moment of event detection.

In Chapter 8, we further elaborate on Q1, Q2 and Q3 in order to define a generic monitor for the MODOCC system. In this chapter we concentrate on Q4 (which coincides with Q3) and Q5, in order to build the MODOCC system.

7.2 GMS specialization

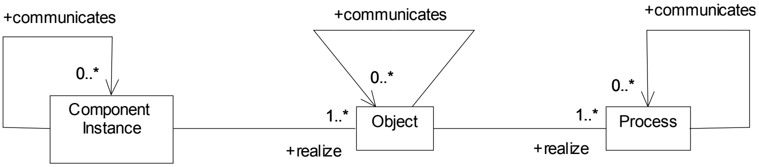
In this section we describe the specialization of the GMS into an MSS for monitoring object and component communication. Stage two of our design methodology suggests that a designer should follow three steps in the specialization of a GMS: definition of a monitoring model, definition of a data structure, and adding of processing components. We follow steps one and two, however we do not follow step three, since we do not require any additional processing of monitoring data in the MODOCC system.

We structure the discussion in the following manner. First we define a MODOCC MM that consists of an entity model and a behavior model. The entity model describes the abstract structure of a monitored application in terms of objects, component instances, processes and the relations among them. The behavior model describes the behavior (within a single application run) of entities (from the entity model) in terms of events, where an event represents the successful completion (see Chapter 2) of a non-distributed communication or lifecycle activity. The behavior model also defines ordering relations (causal precedence and limited realized causality) among events so that using them monitors can reason about operation invocations (including relations among them) in the behavior of the monitored application. Then we discuss a data structure of the monitoring reports for the MODOCC system.

7.2.1 Entity model

Figure 7-1 shows a UML class diagram of the entity model of a middleware-based application.

Figure 7-1 Entity model



Entities

We distinguish three types of entities in a middleware-based application: *component instances*, *objects* and *processes*. We defined the object and component instance concepts in Chapter 3. We assume that a component can offer its functionality through multiple interfaces, while an object can offer its functionality through a single interface. We defined the process concept in Chapter 2.

Entity relations

In Chapter 3 we made the observation that component technology often evolves as a superstructure on object middleware. In this model we use this observation to define that one or more objects *realize* the behavior of a component instance within a single application run. In turn, one or more processes *realize* the behavior of an object by sequentially performing activities during application runtime. Some of these activities result in entities communicating among each other. Component instances and objects communicate via operation invocations and processes via message exchange. In the behavior model we further discuss concrete activities that entities of the entity model can perform.

7.2.2 Behavior model

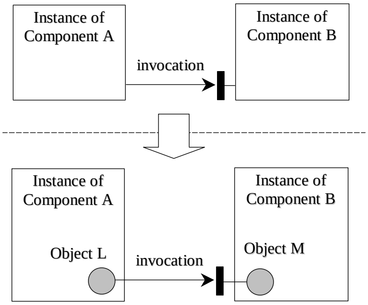
We start by expressing component instance communication using object communication, then object communication using activities that processes perform. We also discuss lifecycle activities for components and objects.

Having identified the necessary activities, we define event types and event attributes for each event type. We then discuss relations among events.

Mapping component communication to object communication

In Figure 7-2, component instance A invokes an operation on an interface provided by component instance B.

Figure 7-2 Mapping of component communication to object communication



Since we assume that objects realize the behavior of components, there exist two objects L and M that realize this particular operation invocation at A and B respectively.

Mapping object communication to process communication

We model an operation invocation between two objects as a sequence of activities. We distinguish three types of operation invocations (Figure 7-3): (a) method call, (b) synchronous operation invocation, and (c) asynchronous (oneway) operation invocation. P_1 and P_2 represent processes involved in performing communication activities. O_1 and O_2 represent the caller and called objects respectively. Circles represent activities that we consider significant. Arrows represent the order in which processes execute activities. The object middleware used to build the monitored application prescribes this order.

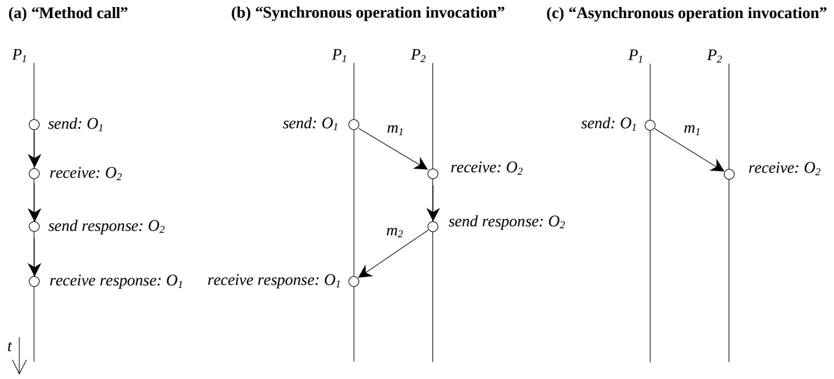


Figure 7-3 Mapping of object communication to process communication

In (a), both O_1 and O_2 represent local objects. We consider (a), because most OO programming languages handle method calls in this way. We model a method call using four activities: *send* represents the calling of the method at O_1, *receive* represents the beginning of the execution of the method body at O_2, *send response* represents the completion of the execution of the method body at O_2, and *receive response* represents the return of results at O_1. Hence we model the execution of a method call with the consecutive execution of the activities *send*, *receive*, *send response* and *receive response*, by a single process P_1.

In (b), O_1 and O_2 represent objects possibly situated at remote physical locations (or different execution environments at the same location). We consider (b), because most contemporary object middleware products handle synchronous operation invocations in this way (similarly to an RPC call). We model a synchronous operation invocation using four activities: *send* and *receive response* associated with object O_1, and *receive* and *send response* associated with remote object O_2. During a synchronous operation invocation, a process P_1 executes *send*, to send a message m_1 to process P_2, which performs *receive* to receive the message. Message m_1 constitutes the forward direction of an operation invocation. After sending the message, P_1 waits for a response message. When the computation of the operation implementation completes, process P_2 performs *send response* to send message m_2 to process P_2 containing the results (if any) from the operation computation. Process P_1 performs *receive response* to accept the message.

In (c), O_1 and O_2 again represent objects possibly situated at remote physical locations (or different execution environments at the same location). We consider (c), because the CORBA middleware handles asynchronous (oneway) communication in this way. An asynchronous operation invocation differs from a synchronous operation invocation in that it does not require a response message.

Lifecycle activities

Regarding the lifecycle of component instances we consider two activities: *create* and *release*, where *create* results in a new component instance and *release* results in the releasing of any system resources associated with an existing component instance. Regarding the lifecycle of an object we again consider two activities: *create* and *release*, with similar meaning as defined above, but applied to objects.

Event types

In the previous sections we have identified communication and lifecycle activities that we find interesting to a monitor. In this section, we associate with each activity an event type. Each time a process successfully performs an activity that we want to monitor, we require the MODOCC monitoring system to detect this condition as an event of the event type associated with that activity. In Table 7-1 we define the required communication event types.

Table 7-1 Communication events

Communication events	**Description**
m_snd	This event occurs when *send* has completed during a method call.
m_rcv	This event occurs when *receive* has completed during a method call.
m_snd_resp	This event occurs when *send response* has completed during a method call.
m_rcv_resp	This event occurs when *receive response* has completed during a method call.
i_snd	This event occurs when *send* has completed during a synchronous operation invocation.
i_rcv	This event occurs when *receive* has completed during a synchronous operation invocation.
i_snd_resp	This event occurs when *send response* has completed during a synchronous operation invocation.
i_rcv_resp	This event occurs when *receive response* has completed during a synchronous operation invocation.
o_snd	This event occurs when *send* has completed during an asynchronous operation invocation.
o_rcv	This event occurs when *receive* has completed during an asynchronous operation invocation.

With $Types_{comm}$ we denote the set of all communication event types.

In Table 7-2 we summarize how the defined event types relate to operation invocations.

Table 7-2 Invocations

Invocation type	Corresponding tuple of events
Method call	(m_snd, m_rcv, m_snd_resp, m_rcv_resp)
Synchronous operation invocation	(i_snd, i_rcv, I_snd_resp, i_rcv_resp)
Oneway operation invocation	(o_snd, o_rcv)

In Table 7-3 we define lifecycle event types.

Table 7-3 Lifecycle events

Lifecycle events	Description
ob_create	This event occurs when *create* has completed during class instantiation.
ob_release	This event occurs when *release* has completed during class instantiation.
ci_create	This event occurs when *create* has completed during component instantiation.
ci_release	This event occurs when *release* has completed during class instantiation.

With $Types_{life}$ we denote the set of all lifecycle event types.

We define function $Type: E \mapsto (Types_{comm} \cup Types_{life})$, such that $Type(e)$ for an event $e \in E$ gives the type of the event, where E represents the set of all events in some application run.

Event attributes

In Chapter 2 we define that an event has three types of attributes: *time*, *address*, and *information*. The time attribute describes the moment of event occurrence. The address attribute describes the place at which the event occurred. The information attribute describes the effect or result of the event. In this section we define concrete attributes for the communication and lifecycle events (Table 7-4).

Table 7-4 Event attributes

Event type	Time attribute	Address attributes	Information attributes
m_snd	lt	o_{caller}, p	o_{called}, op, params
m_rcv	lt	o_{called}, p	o_{caller}, op, params
m_snd_resp	lt	o_{called}, p	o_{caller}, op, params
m_rcv_resp	lt	o_{caller}, p	o_{called}, op, params
i_snd	lt	o_{caller}, p	o_{called}, op, params
i_rcv	lt	o_{called}, p	o_{caller}, op, params, p_{caller}
i_snd_resp	lt	o_{called}, p	o_{caller}, op, params
i_rcv_resp	lt	o_{caller}, p	o_{called}, op, params, p_{called}

o_snd	lt	o_{caller}, p	o_{called}, op, params
o_rcv	lt	o_{called}, p	o_{caller}, op, params, p_{caller}
ob_create	lt	c, p	id
ob_release	lt	c, p	id
ci_create	lt	p	id
ci_release	lt	p	id

Every event type from the MODOCC MM has one time attribute *lt*. The *lt* attribute accepts as a value a timestamp that determines the moment of event occurrence. The MODOCC system uses timestamps issued with the help of a vector logical clock.

Every event type has an address attribute *p* that represents the process that performed this event. Communication events have as an additional address attribute the identifier of the object at which this event occurred (o_{caller} or o_{called}, depending on the event type). Object lifecycle events have as an additional address attribute the identifier of the component this object realizes.

All communication events have three information attributes: o_{caller} or o_{called} (depending on the event type) representing the identifier of the other object that participates in the operation invocation, *op* representing the signature of the operation invocation, and *params* representing the parameters and/or result values of the operation invocation at the moment of the occurrence of the communication event. Furthermore, *i_rcv* and *o_rcv* have an additional p_{caller} information attribute, and *i_rcv_resp* has an additional p_{called} information attribute. These additional attributes contain the identifier of the process that has sent (performed a *send* or *send response* activity) a message to another process as part of performing an operation invocation. Monitors that analyze communication events detected by the MODOCC system use these event attributes in combination with the timestamp to order and match individual events as part of an operation invocation. Lifecycle events have as an information attribute the identifier of the entity (object or component) whose lifecycle they relate to.

With $Attrib_{name}$ we define the set of all attribute names. With $Attrib_{value}$ we define the set of all possible values for an attribute of an event. We define an (infix) accessor operator "." : $E \times Attrib_{name} \mapsto Attrib_{value}$, such that for an event *e*, *e*."attribute name" produces the value of the corresponding attribute of this event. For example, for a communication event $e \in E$, *e.op* gives the name of the operation invocation that this event corresponds to.

Example: The "Hello!" application

To illustrate how the model applies to a concrete application, we consider a simple system consisting of two components: a *client* and a *server*. The server component offers an interface, defined in CORBA IDL as follows:

```
//Hello IDL
interface Server
{
    string hello(in string msg);
};
```

The client component C_1 contains object O_1, and the server component C_2 contains object O_2 (Figure 7-4). In the example we consider the following scenario: "C_1 invokes the *hello* operation on C_2". For this scenario, we assume that process P_1 performs activities within C_1 and process P_2 performs activities within C_2.

Figure 7-4 The model of the "Hello" application

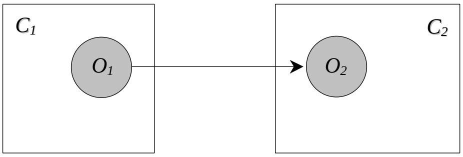


Table 7-5 contains a list of possible events that a monitor receives from the MODOCC system for the selected scenario. We represent each event by its type name followed by a list (between brackets) of its attribute values in the order presented earlier. For simplicity, we disregard the timestamp attribute by using a "_" on the corresponding position.

Table 7-5 List of events for the "Hello!" application

Nr.	Event
1	ci_create(_, P_1, C_1)
2	ci_create(_, P_2, C_2)
3	ob_create(_, C_1, P_1, O_1)
4	ob_create(_, C_2, P_2, O_2)
5	i_snd(_, O_1, P_1, O_2, "hello", "Hello, client here!")
6	i_rcv(_, O_2, P_2, O_1, "hello", "Hello, client here!", P_1)
7	i_snd_resp(_, O_2, P_2, O_1, "hello", "Welcome, client! Server here!")
8	i_rcv_resp(_, O_1, P_1, O_2, "hello", "Welcome, client! Server here!", P_2)
9	ob_release(_, C_2, P_2, O_2)
10	ob_release(_, C_1, P_1, O_1)

11	ci_release(_, P_2, C_2)
12	ci_release(_, P_1, C_1)

First, the application creates two components C_1 and C_2 at the client and the server hosts, respectively (events 1, 2). The application creates O_1 within the context of C_1 (event 3), and O_2 within the context of C_2 (event 4). The process P_1 starts executing the functionality of O_1, which leads to performing of “hello” invocation on O_2 (event 5). O_2 receives the invocation (event 6) in the context of process P_2. After executing the operation functionality, O_2 sends the result (event 7). The result arrives at O_1 in the context of P_1 (event 8). Then the application releases objects and component instances (events 9, 10, 11, 12). Figure 7-5 shows the events altogether and relative to the middleware that handles the distribution details.

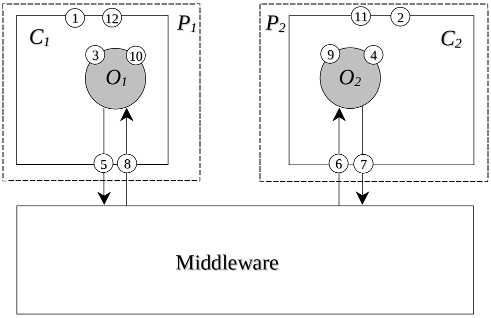


Figure 7-5 Events for the “Hello” application

Events 5,6,7, and 8 represent successfully completed local interactions (between the objects and the middleware) that altogether (in this order) represent a synchronous operation invocation.

Relations among the events

Monitors require from the MODOCC system information about the order of communication events, so that they can restore correctly individual invocations, and further to order invocations relative to each other within a single application run. The lack of global time in low-cost distributed system motivates the use of logical time to order the events in an application run monitored with the MODOCC system.

In Chapter 2 we distinguished two types of causality relations: *potential* and *realized*. In the following sections we describe how the MODOCC MM uses potential and realized causality.

Monitoring potential causality with the MODOCC system

We associate with every event in an application run a logical timestamp attribute. As discussed in Chapter 2, we can define a partial order relation, denoted as “ $<$ ”, on the timestamps issued by a logical clock system. With a strongly consistent logical clock system, the “ $<$ ” relation among timestamps allows monitors to restore the causal precedence relation (denoted by “ $\to$ ”) among events they observe. We shall discuss a particular design for a vector clock in the instrumentation design presented later in this chapter. The causal precedence relation provides potential causality among the events of an application run.

The partial order that causal precedence gives, by itself only means that events occurred in some order in that particular application run. If a designer has some additional knowledge about the model (e.g., original design specification) of the monitored application, he/she may attempt to determine whether the model allows a behavior that produces the observed application run. If the model allows a trace however, the designer cannot say anything about the correctness of the implementation, because a single run does not determine all possible ways to execute the application. The designer may have to check this way a lot of different runs in order to raise his confidence in the implementation. If the model does not allow a run, then we have two possible conclusions: the monitored application implementation does not faithfully implement its model or, it does implement its model faithfully, but it produces a disallowed run because of a factor not considered in the model (e.g., a communication failure caused by cutting a wire). The designer needs to reexamine its implementation in order to distinguish the two cases. To narrow down the work during this reexamination, a designer may want to analyze the causal relationships among events, for example to find the reason for an event he considers as error.

When the designer does not have access to the model of the monitored application, he may try to determine the possible causes of a particular event using the potential causality provided by the causal precedence relation for an application run. Nevertheless, another application run may produce the same event but in different order with the other events in the application run, because the potential causality does not provide the certainty of realized causality. Hence, potential causality only narrows down the choice for a cause to a list of possible causes.

Middleware-based applications have something common in their behavior – the middleware. By definition, the middleware represents a collection of reusable blocks of functionality to provide to the application layer certain transparency from low-level programming details. As a consequence, all middleware-based applications sharing a particular middleware, also share its behavior. For example, a CORBA-based application would use the behavioral patterns of the CORBA middleware for communication among its (potentially) distributed application objects. Hence, all communication would follow the same model – the design of the intended behavior of the CORBA object middleware. All this implies that, since we have the goal to monitor middleware-based applications, our MODOCC system has access to an important asset – the model of the middleware. This model does not change over time and it prescribes all possible types of communication behavior (earlier we limited the discussion to three) between middleware objects. Furthermore, we assume that the middleware implementation used during monitoring correctly implements its model. In the next section, we discuss how the additional information from the middleware model allows us to reason about realized causality relation among events of an application run.

Monitoring realized causality with the MODOCC system

We have already used the model of the middleware when we defined the event types and how events represent a method call, a synchronous operation invocation, and an oneway operation invocation. According to this model, in the scope of a single invocation certain event types precede other event types in terms of realized causality. We define the binary relation $Causality_{types} \subset ((Types_{comm} \times Types_{comm}) \cup (Types_{life} \times Types_{life}))$, as the causal relation between event types. Table 7-6 shows all event type pairs that this relation contains.

Table 7-6 Causal relations between event types according to the middleware model

Event type cause	Event type effect
m_snd	m_rcv
m_rcv	m_snd_resp
m_snd_resp	m_rcv_resp
i_snd	i_rcv
i_rcv	i_snd_resp
i_snd_resp	i_rcv_resp
o_snd	o_rcv
ob_create	ob_release
ci_create	ci_release

We provide the following example application run (Figure 7-6) to illustrate realized causality between events.

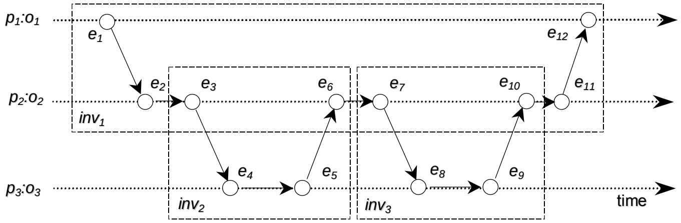


Figure 7-6 Three nested synchronous operation invocations

The application run consists of three nested synchronous operation invocations. Horizontal lines represent the passing of time at an object o_i, $i \in \{1,2,3\}$. Each object o_i has a process p_i associated with it. Each process performs the activities of the corresponding object. Objects perform three synchronous operation invocations that we denote with inv_1, inv_2 and inv_3. The quadruples $(e_1, e_2, e_{11}, e_{12})$, (e_3, e_4, e_5, e_6) and (e_7, e_8, e_9, e_{10}) represent inv_1, inv_2 and inv_3 respectively. According to their timestamps, these events relate with causal precedence in the order provided by the arrows in the diagram. According to the middleware model, the events within each quadruple relate among each other with realized causality, for inv_3, e_1 represents the definite cause of e_2, e_2 represents the definite cause for e_{11}, e_{11} represents the definite cause of e_{12}, and so on for the other two invocations.

For an application run we define the Limited Realized Causality (LRC) relation as a binary realized causality relation between the events of every invocation in that application run. We call it *limited*, because it only involves events resulting from middleware activities that we want to observe: communication and lifecycle.

Let $DC = (E, \rightarrow)$ represent an application run. Let $C(\rightarrow)$ represent the transitive closure of the causal precedence relation $\rightarrow$, and $R(\rightarrow)$ represent its transitive reduction.

Definition 7-6 Limited Realized Causality

$LRC \subset E \times E$, $LRC = \{(e_1, e_2)\text{: } (Type(e_1), Type(e_2)) \in Causality_{types}$ *and* $(((Type(e_1) \in Types_{comm}$ *and* $Type(e_2) \in Types_{comm})$ *and* $((e_1, e_2) \in R(\rightarrow)))$ *or* $((Type(e_1) \in Types_{life}$ *and* $Type(e_2) \in Types_{life})$ *and* $((e_1, e_2) \in C(\rightarrow))$ *and* $(e_1.id = e_2.id)))\}$.

Informally, for two events e_1, e_2 to participate in the LRC, their types need to relate causally in one of the ways defined earlier in Table 7-6. Furthermore, for communication events, the two events should participate in the transitive reduction of the causal precedence relation, i.e., e_1 should directly precede e_2. For lifecycle events, both events should regard the same object or component instance, and the events should participate in the transitive closure of the causal precedence relation, i.e., e_1 should transitively precede e_2. Note that LRC defines a strict partial order on E, and LRC$\subset$ $C(\rightarrow)$, .

The main application of the *LRC* comprises various analysis techniques regarding invocations that need the certainty of realized causality. For example, if a monitor receives only the first two events of an invocation quadruple, this would imply an application hasn't finished preparing the result of the operation invocation (if any). If the last two events of the quadruple never appear at the monitor (provided the monitoring system operates correctly) this would imply an error in a particular application component which the monitor can recognize automatically, without intervention of a human operator. If only the first and the last events appear at a monitor, this would imply an error in the operation of the monitoring system itself, since according to the middleware model this kind of behavior cannot happen. With this additional information, a human operator can concentrate on dealing with the error, instead of spending a lot of time searching for its location.

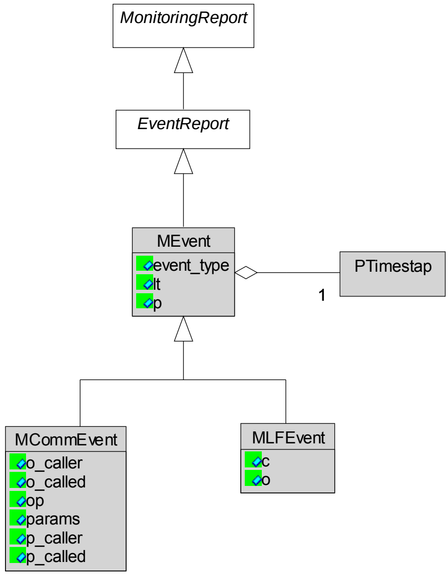
When a monitor receives information about an event (in the form of an event report), it uses the information from the logical timestamp attribute of the event to restore the causal precedence relation. Then, based on the type of the event (and other event attributes in case of lifecycle event), the monitor can restore the limited realized causality relation.

7.2.3 Data structure

Each time the MODOCC system detects a communication or a lifecycle event we require the information about this event to become available to monitors. The MODOCC system makes an event available to monitors by generating an event report. An event report represents a special kind of monitoring report that describes an event in terms of its type and specific attributes.

In this section we define a data structure for the MODOCC MM. This data structure defines (the structure of) the MODOCC event reports (i.e., reports that the MODOCC system can generate). Figure 7-7 shows the structure of MODOCC event reports.

Figure 7-7 The structure of MODOCC event reports



In Chapter 6 we defined a structure for monitoring reports that allows the GMS to work with monitoring data in a generic way. Because we build the MODOCC system on top of the GMS, we derive the definition of MODOCC reports from the EventReport class, where the MEvent class represents a MODOCC event report. We define two additional classes of event reports derived from MEvent: MCommEvent and MLFEvent, representing communication events and lifecycle events respectively. The MEvent, McommEvent, and MLFEvent define specific attributes for every event report corresponding to the event attributes from the MODOCC MM (except for the *event_type* attribute – see below).

The MEvent class defines specific attributes common to both communication and lifecycle events, and an additional attribute *event_type* to distinguish among individual event types (that we defined in the MODOCC MM). The MCommEvent defines attributes specific to communication related events and the MLFEvent defines attributes specific to lifecycle events. The GMS can access all specific attributes in a generic way by enumerating them in the *s_attributes* list attribute of the MonitoringReport base class (see Chapter 6 for the details).

The values of the *event_type* attribute enumerate the event types we defined in the MODOCC MM. The values of all other attributes except *lt*, represent unique identifiers for entities from the MODOCC entity model. The *lt* attribute contains as a value an instance of the PTimestamp class (Figure 7-8).

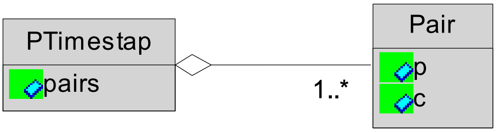


Figure 7-8 The logical timestamp

The MODOCC system uses a vector clock system to generate vector logical timestamps. We discussed the details of the vector clock system in Chapter 2. The value of the *lt* attribute represents a vector of pairs, where every pair has an attribute *p* representing the id of a process, and an attribute *c* representing the progress of the process in *p*, according to the vector clock system. In the next section we further show how we implemented the rules of the vector clock system.

7.3 Instrumentation design

In this section we describe the instrumentation design for the MODOCC system. We consider three types of instrumentation: instrumentation of an object oriented programming runtime (Java) to monitor objects (i.e., objects local to a single execution environment), instrumentation of an object middleware (CORBA) to monitor middleware objects (i.e., objects that allow distributed communication), and instrumentation of a component middleware (DSC) to monitor component instances.

The MODOCC MM defines several event types. For each event type, we need to build software sensors that can detect the completion of the activities represented by the event. We also need to determine where to place these sensors with respect to the monitored application and the middleware.

Middleware technologies provide to application developers tools that facilitate the application development process. During the instrumentation design we also need to provide tools that automate the instrumentation process, so that developers can easily adapt for the purpose of monitoring large numbers of different application objects and components.

The instrumentation should communicate with the GMS to send monitoring data. For this purpose we design an architecture of a CMA that encapsulates all sensors, their management, the packaging of the results of measurements into event reports, and the sending of these event reports to the GMS according to its service definition.

7.3.1 Sensor design and placement

In order to detect the occurrence of an activity corresponding to an event type from the MODOCC MM, we want to put a sensor on the execution path of the process performing the activity, so that the sensor would execute right after the activity has completed its execution. We call this method of detecting events *interception*. We associate with each event type from the MODOCC MM, a sensor that gets executed right after the moment of the completion of the activity. A sensor measures all the various values that constitute the attributes of the event that this sensor detects. The instrumentation has the responsibility to package these values in an event report and to send the report to the GMS.

For communication events, we embed sensors on the communication path between objects in order to detect the completions of local interactions between the middleware and the objects participating in the communication. For lifecycle events, we embed sensors in the middleware object management facilities in order to detect the creation and destruction of objects.

Sensors for Java

The Java platform offers two interfaces that can support installation of sensors for communication and lifecycle events: the Java Virtual Machine Profiler Interface (JVMPI) [JVMPI], and the Java Platform Debugger Architecture (JPDA) [JPDA].

The JVMPI allows for establishing hooks on the path of the JVM class loading mechanism during application runtime. Using JVMPI, designers can monitor various object activities, such as method calling, creation of new objects, synchronization between threads, etc. The JVMPI mainly finds use in application benchmarking and performance testing, as done in JPMT [HQGM02].

The JPDA debugging interface allows for monitoring (and manipulation in terms of stopping, resuming, and step-by-step execution) of a JVM started in debug mode (this includes everything that JVMPI can do). The Java development kit offers the Java Debugging Interface (JDI) as a default Java-based debugger front-end implementation of JPDA. We use the JDI to place sensors that can detect “method call” and lifecycle events for Java objects. We prefer to use the JDI because it presents the cheapest (with respect to development resources) way to implement generic instrumentation for Java. The work described in [BrM02] uses the same approach for formal verification of the behavior of Java programs.

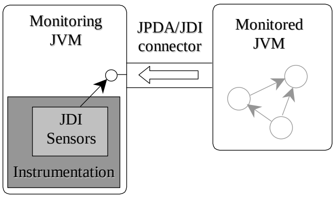


Figure 7-9 JPDA / JDI instrumentation for monitoring of Java object method calls

Figure 7-9 presents the sensor architecture using the JPDA / JDI approach. The instrumentation attaches JDI sensors to the JVM running a monitored application. JDI provides several standard notifications from which we use two: MethodEntryEvent and MethodExitEvent. The JDI sends a MethodEntryEvent notification when an object has just called a method call (but before the method started executing). The JDI sends the MethodExitEvent notification when an object has just completed the execution of a method call.

When a notification occurs, the JDI calls a handler (in this case the JDI sensor we provide) in which we perform the measurements necessary for generating communication events. At the moment of a notification the JDI offers two options for progressing the monitored JVM: suspending all threads in it, or suspending only the thread that carries out the activity that resulted in the notification (the method call). We use the second option as it does not enforce as much intervention in the application execution as the first one – the process (i.e. thread in Java) execute the sensor and then continues with its normal application execution. In contrast, suspending the whole JVM diminishes the concurrency in the monitored application, by making all notifications sequential. After performing the necessary measurements, the notification handler (our sensor) resumes the suspended thread so it can continue with its normal execution.

The MethodEntryEvent and MethodExitEvent notifications allow for monitoring both arbitrary method calls and the creation of objects – the JVM processes object constructors (calls to the "new" operator) and destructors (i.e., calls on finalize() from the Java garbage collecting mechanism) similarly to any other method. Note that since Java has automated (implicit) garbage collection (which may occur upon demand for more memory), we cannot monitor explicit object destruction.

The JDI provides a straightforward way to measure the values of the name, parameters and results (including possible exceptions) of a method. The monitoring system cannot use the standard object reference (the one provided by the programming language runtime) to identify local objects outside the JVM. For this reason, the instrumentation generates globally unique identifiers using a mechanism that guarantees uniqueness. The UUID/GUID [RPC] represents a generic method (that we do not use because Java does not support it) for generation of globally unique identifiers. In the MODOCC monitoring system we use proprietary identifiers based on information about the host, the JVM and the object reference within the JVM. We use a similar mechanism for the identifiers of the processes (threads) involved in object communication. We discuss the measuring of the time event attribute in the next section.

Sensors for the CORBA object middleware

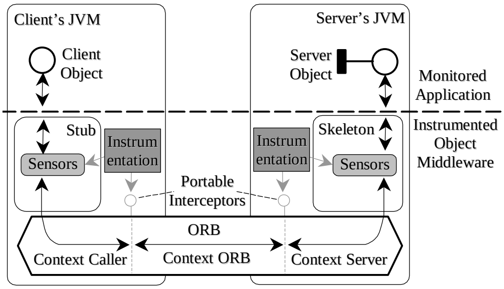
CORBA objects can communicate among each other across the boundary of a single execution environment, by using synchronous operation invocations and oneway operation invocations. We model both types of invocations in the MODOCC MM and therefore we want to monitor both. Furthermore, we want to monitor the lifecycle of CORBA objects too.

CORBA offers two generic interfaces that allow one to monitor (middleware) object lifecycle and communication activities: the Portable Object Adapter (POA) and the Portable Interceptors (PI). We discussed both interfaces in Chapter 3.

For monitoring of lifecycle activities we provide two alternative approaches: using the POA and modifying the JacORB CORBA implementation. Every CORBA-based application uses a reference to a POA to create its CORBA objects. The ORB provides a default POA called “RootPOA”, which we use to create a special MonitoringPOA. We than replace the uses of the “RootPOA” in the monitored application, with our MonitoringPOA, i.e., we modify the original monitored application code. The MonitoringPOA contains sensors, which detect object creation or destruction. The second approach does not require modification of the application source code. In this approach, we embedded the sensors directly in the JacORB implementation of the CORBA standard. This method however, makes the instrumentation specific to the JacORB product. We use the modified JacORB implementation as the default approach in the MODOCC prototype. Designers of monitoring systems, who do not prefer modifications of the ORB or use other ORB implementations, can resort to the POA approach.

For monitoring communication activities we use modified stubs/skeletons and the PI interface, to detect communication evens (Figure 7-10).

Figure 7-10
CORBA sensors



A stub in CORBA represents the client object proxy and a skeleton represents the server object proxy (see Chapter 3 for more details on the proxies in object middleware). We embed sensors on the path of a CORBA invocation in the stub at the client object side and in the skeleton at the server object side. The client sensors detect the communication events that represent local communication interactions between the client object and the middleware – sending an operation invocation and receiving the result. The server sensors detect the communication events that represent local communication interactions between the server object and the middleware – receiving an operation invocation and sending results. The CORBA middleware processes an operation invocation in three separate contexts: the context of the client object, the context of the ORB and the context of the server object. The client sensors have access to the current process context of the client object. In Chapter 2 we discussed that the metrication rules for the vector clock system require access to the current process context in order to store there information about the progress of the clock system. Analogously, server sensors get executed in the context of the server object. The metrication rules of the vector clock system also requires that the client and the server exchange information between the contexts of their corresponding processes. We use the Portable Interceptors interface to install special monitoring interceptors that copy the necessary information between the contexts of the client and the server. This works in the following way. Every time a client object invokes an operation, the monitoring interceptor at the client side copies the current value of the vector clock of the current process from the client object context to the client ORB context. The ORB then transmits this value to the server ORB context together with the request object representing the operation. The server monitoring interceptor copies the value of the clock that arrives with the operation request, into the context of the server object, and merges it with the clock value of the process of the server object that will process the operation (according to the rules of the vector clock system). The whole scheme repeats on the way back with the response of the operation invocation, except for oneway operations (which do not produce a response).

CORBA caller identity problem

The CORBA technology follows closely the client-server paradigm, which leads to one significant problem during monitoring: the target object has a well-defined identity represented by its unique object reference, however, the caller (client) does not have a clearly defined identity (such as an unique reference). This makes it difficult for an external observer (such as a monitor using the MODOCC system) to uniquely determine the identity of the caller object of an invocation. The work reported in [HaSte97] identifies this identity problem of the sender (also called asymmetry between send and receive) in the more generic context of communication among processes. A sender process typically knows about the receiver process, while the receiver process may receive a message without information about the sender, because the communication system puts the focus explicitly on the message content and message handling by the called object.

Caller identity does not present a problem for the operation of the CORBA platform itself. The ORB can perform its task without this knowledge, because for the purpose of receiving invocation results back at the caller the ORB internally maintains proprietary information. We cannot use this information outside of the ORB. We need a generic method of caller identification.

In the solution we propose, objects alternatively switch between the client and the server roles, to cooperate among each other using remote invocations. When a server object receives an invocation, it executes the invocation according to its implementation. If, as a consequence of this execution, the same object needs to make an operation invocation on another remote object, the server object first assumes a client role, and then performs the operation invocation. We can determine the identity of a caller object (one that performs an invocation on another one) as the identity of the last server object within the current process (of the caller) that received an invocation request (Figure 7-11).

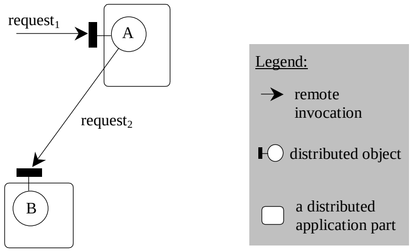


Figure 7-11 The client identity becomes the identity of the object which last had the server role.

According to this solution, object A becomes the caller of request$_2$ on object B, because object A last served request in the context of the same process. Nevertheless, this solution has one exception – active objects. An active object performs an initiating remote call (without any other objects calling it). For example, such active object may represent a human user that interacts with the monitored application. We use two basic default identities in this exceptional case: a GUI and a MAIN. We use GUI as an identity of the caller when an object invocation results directly from the interaction of a human user with the graphical user interface of the monitored application. We use MAIN as the identity of the caller in all other cases, including the start of the application.

Sensors for the DSC component framework

The Distributed Software Components (DSC) [BaBa98] component framework belongs to the family of component middleware. The DSC framework represents a generalization and an implementation of the TINA [HaSte97][Ste97][Kri97] computational object model. In this section we describe the instrumentation of the DSC runtime library that allows for the monitoring of component lifecycle and communication events. In Chapter 8 we describe a monitor developed as part of the DSC framework's testing suite.

The DSC framework defines the rules and constraints that allow component instances to collaborate. DSC uses CORBA as a distributed processing environment. DSC designers use the Java programming platform as an implementation technology.

A DSC component instance has one control interface and zero or more operational interfaces defined in CORBA IDL. The control interface provides common functionality related to various CORBA services, such as persistence and transaction support. The operational interfaces define the specific service that a component offers to its users. DSC component instances communicate with each other in a client/server fashion by using

CORBA remote invocations. In addition to that, DSC components can communicate using component events in a publish/subscribe fashion.

Within the FRIENDS project [FRIENDS] we have developed an instrumentation that allows for the monitoring of lifecycle and communication events of DSC components. We integrated this instrumentation with the DSC runtime library to allow component developers to observe the behavior of their prototype applications for debugging and presentation purposes (presentation of results to the project management body and to the scientific community).

The DSC framework explicitly supports creation and destruction of component instances. We embedded sensors in the routines for creation and destruction in order to detect component lifecycle events.

The MODOCC system supports monitoring of component communication only for CORBA-based invocations among DSC components. We do not consider the event-based component communication. In DSC, a single CORBA object realizes each operational interface of a component. Therefore, to detect communication events we reuse the instrumentation for the monitoring of CORBA invocations between objects presented in the previous section.

In DSC the caller component always has a unique reference associated with it, hence we do not face the caller identity problem.

7.3.2 Design of instrumentation tools

In this section we discuss the instrumentation tools that we have developed to provide developers with an automated instrumentation process. The JDI interface does not require additional tool support to install the instrumentation of the MODOCC system with any Java monitored application. For the CORBA and the DSC instrumentation however, we have developed design-time (see Chapter 5) instrumentation tools.

Modified IDL compiler for CORBA

As we have discussed in Chapter 3, the development of software using CORBA requires the specification of object interfaces in IDL. The CORBA standard provides mappings of an IDL specification to all major programming languages. In Chapter 3 we made an overview of the software development process for object middleware. In CORBA, designers process the IDL specification with an IDL compiler for a specific programming language or platform. The IDL compiler generates a set of source code templates that allow a programmer to use a CORBA object in the role of a server and in the role of a client. Among these templates we find the two types of proxies corresponding to the stub (client) and a skeleton (server), in which we embed the sensors necessary for detecting communication events. The manual modification of the proxies would require from instrumentation developers a lot of effort dedicated solely to instrumentation. We have automated the proxy modification process by providing a tool based on the IDL-to-Java compiler of JacORB.

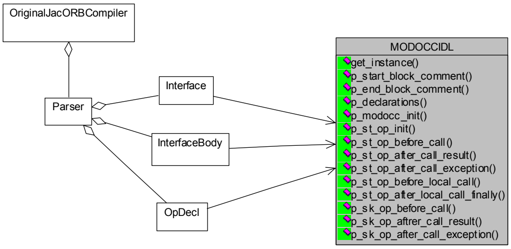


Figure 7-12 The modified IDL compiler

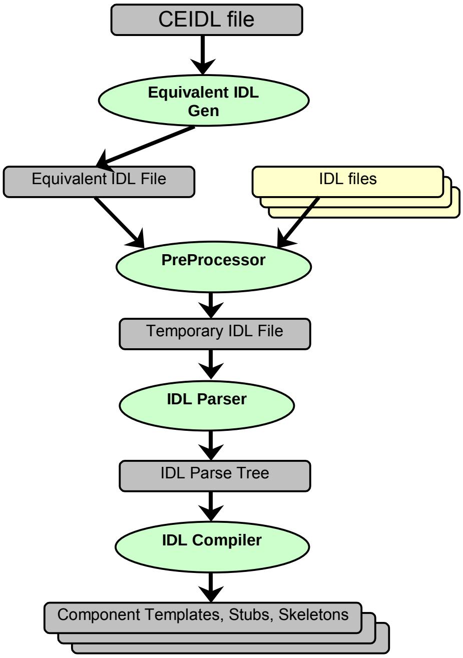
Figure 7-12 shows the architecture of the IDL compiler and the modifications we have made to provide the sensors. The JacORB IDL compiler uses three classes: Interface, InterfaceBody, and OpDecl. The Interface class has the responsibility to generate the proxy declarative parts and the InterfaceBody class has the responsibility to generate the body of the proxies. The OpDecl has the responsibility to generate the code for each individual operation in an interface. We modify these three classes by embedding in them statements that generate sensor code necessary for detecting communication events. We place the code generation routines in a single class called MODOCCIDL.

During the compilation of an IDL specification the modified IDL compiler automatically embeds all the necessary sensors on their appropriate places. Although based on the JacORB product, the tool generates Java stubs and skeletons that comply to the CORBA Portable Stubs and Skeletons specification, which means that designers can in principle use other ORB products than JacORB together with our tool.

Modified component generation facilities

The DSC framework provides to component developers a tool called DscGen. The DscGen tool has the responsibility to process a component specification and to generate the templates of a component described by this specification. Figure 7-13 shows the steps in the process of specification compilation performed by the DscGen tool.

Figure 7-13 Compilation of component specifications with the DscGen tool



Designers use the CEIDL language to create a specification of a DSC component. In this specification, designers define the component interface names, dependencies of the component on the interfaces of other components, all events it generates and events that it can consume. In the first step, The Equivalent IDL Gen compiles a CEIDL specification to a standard CORBA IDL specification called Equivalent IDL. The PreProcessor combines the Equivalent IDL with the IDL files describing the individual operational interfaces of the component. The IDL Parser parses the result and feeds the parse three to the IDL Compiler. The IDL Compiler generates the necessary templates, stubs and skeletons.

Internally, the DscGen tool also uses the JacORB IDL compiler to generate the stubs and skeletons necessary for components to communicate. Therefore, by providing the modified JacORB IDL compiler to the DscGen tool we allow detecting of component communication events.

7.4 CMA design

The Co-located Monitoring Agent (CMA) has the responsibility to encapsulate all instrumentation specific mechanisms and implement the generic interfaces for interaction with the GMS (see Chapter 6 for more details). In this section we present the architecture of the CMA for the MODOCC system.

In the MODOCC system we design a CMA for monitoring of object and component communication. Figure 7-14 shows the CMA architecture in relation to the GMS.

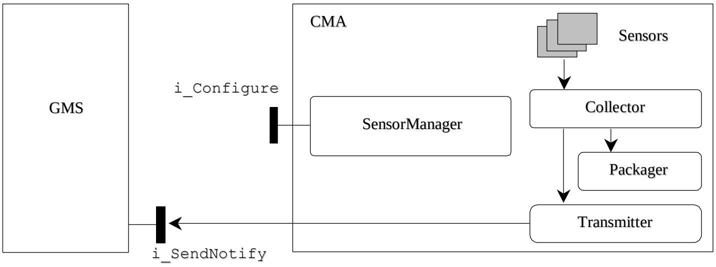


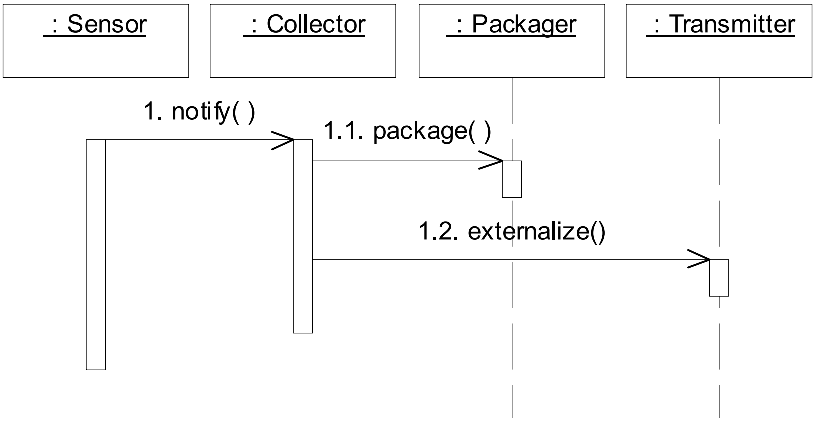
Figure 7-14 A CMA architecture for the MODOCC instrumentation

In Chapter 6 we refined the GMS service into several interfaces that the components of the GMS can implement. According to the GMS software architecture, the instrumentation implements two interfaces i_Configure and i_SendInterrogate. The CMA of the MODOCC system only implements the i_Configure interface. Since the MODOCC system produces only event reports and since we require online monitoring, we consider event-driven report delivery using notifications on the i_SendNotify interface sufficient for providing timely delivery of event reports to monitors. Therefore, the CMA does not implement the i_SendInterrogate interface.

The CMA has five main component types: Sensors, a SensorManager, a Collector, a Packager and a Transmitter. A Sensor represents a software sensor that can detect one particular event type. The SensorManager has the responsibility to manage and configure different sensors. For example, the Sensor manager can turn some sensors on and off based on the monitors' runtime requirements for monitoring data. A Collector has the responsibility to collect the information generated by sensors. The Packager has the responsibility to package information coming from sensors into proper event reports. The Transmitter has the responsibility to send event reports to the GMS.

In Figure 7-15 we show how the CMA components interact during monitoring. When a Sensor executes, it sends measured information to the Collector with a notify() operation. The Collector sends the information from the Sensor to the Packager with a package() operation. The Packager packages an event report and returns the result to the Collector. The Collector then calls the Transmitter with an externalize() operation, in order to prepare the event report for sending to the GMS.

Figure 7-15
Sequence diagram of event generation



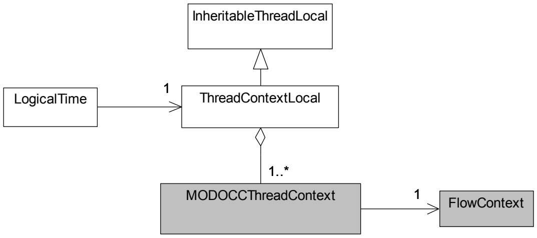
Internally, the Transmitter provides two basic strategies for sending monitoring data to an LMA (of a GMS): *immediate forwarding* and *low-priority forwarding*. The Transmitter performs immediate forwarding in the same process context in which the sensor executed. We consider the major benefit of this strategy the immediate sending of new monitoring data to LMAs. This strategy has the drawback that the application process that executes the sensor blocks until the sending of monitoring data to the LMA completes.

The Transmitter performs the low-priority forwarding strategy in a separate thread that has the responsibility to send monitoring data to the LMA at a processing priority lower than the priorities of monitored application processes. In this strategy, the externalize() operation returns immediately, while a low-priority thread starts sending (in the background) the new monitoring data to the LMA. We consider the major benefit of this strategy the lower computational overhead on the application process resulting from a shorter duration for sensor execution as compared to the immediate forwarding strategy. We consider a drawback of low-priority forwarding the additional delay that monitors may experience before receiving the monitoring data (due to the low priority of the sending thread).

Implementing the vector clock rules

The Packager uses the data structure of event reports that we defined earlier to produce event reports containing all necessary attributes values. Generating the value of the timestamp attribute *lt* however, requires some additional steps. To generate a timestamp for an event, the Packager uses the LogicalTime utility class (see Figure 7-16). The LogicalTime class has the responsibility to consistently perform the rules for updating the vector clock system (we discussed the rules for a vector clock system in Chapter 2). Rule R1 requires access to the process context, in order to maintain a local logical clock for each process. Rule R2 requires access to the context of messages exchanged between processes, i.e., the context of operation invocations used in the CORBA middleware (since we use CORBA for the MODOCC system).

Figure 7-16
Structure of the process context used in the CMA



The LogicalTime class allows access to the MODOCCThreadContext that represents the current thread context for a Java program, and the FlowContext that represents the current invocation context for a CORBA-based application. We associate one MODOCCThreadContext object with every thread in a Java program using a special ThreadContextLocal utility class, which extends InheritableThreadLocal. The Java platform offers the InheritableThreadLocal to allow any subclasses of this class to store and retrieve information to and from the context of the current thread. In the MODOCCThreadContext object associated with a thread, we keep the logical clock for that thread. When the Packager requires a timestamp value for an event report, it first retrieves the MODOCCThreadContext for the current thread and then uses rule R1 to advance the clock from the previous clock value. Then the Packager records the new clock value as the timestamp for the current event report. The InheritableThreadLocal also allows for automatic copying of thread context each time a process creates a new (child) process. We use this to assign to a new process its own version of a vector clock derived from its parent's clock.

We associate every MODOCCThreadContext object with one FlowContext object. We send the FlowContext along with every operation invocation so that we can implement the R2 rule of the vector clock consistently.

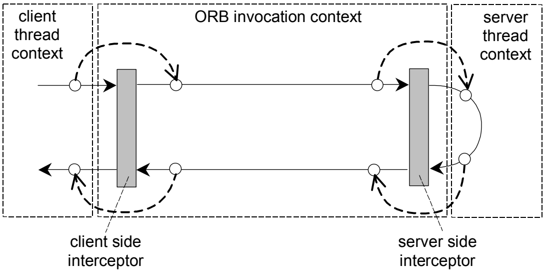


Figure 7-17 Copying FlowContext along the invocation path

Figure 7-17 shows how we use the CORBA Portable Interceptor API to install custom interceptors, which have the responsibility to copy the FlowContext among the three different execution contexts. The ORB executes the custom interceptors in its own ORB context. The client side interceptor has the responsibility to copy the FlowContext object from the client thread context to the ORB invocation context (on the request direction of the invocation), and from the ORB invocation context to the client thread context (on the response direction of the invocation). The server side interceptor has the responsibility to copy the FlowContext object from the ORB invocation context to the server thread context (on the request direction of the invocation), and from the server thread context to the ORB invocation context (on the response direction of the invocation). Furthermore, the interceptors use the ServiceContext field of the CORBA request object to transport the FlowContext information between the two interceptors (because these two may operate at remote hosts).

For more details on the implementation of the mechanism consult the source code of the MODOCC monitoring system available at [MODOCC].

7.5 Performance measurements of the MODOCC prototype

In Chapter 2 we identified two main performance issues during monitoring: monitoring overhead and information consistency. Monitoring overhead concerns the impact of the monitoring system on the behavior of the *monitored* application. Information consistency reflects how faithfully the monitoring system presents information to the *monitoring* application. In this section we discuss the performance of the prototype of the MODOCC monitoring system with respect to the monitoring overhead it introduces to monitored applications. In Chapter 8, we discuss information consistency of the MODOCC system in combination with the MSD Monitor that we present in that chapter.

We provide the results of our performance tests for the CORBA middleware instrumentation. We do not provide performance data for the Java object instrumentation, because it uses the debug mode of the JVM to detect events in the monitored application. The MODOCC monitoring system using the Java object instrumentation has significant overhead, comparable to the overhead of any Java debugger, because the debug mode of the JVM executes the monitored application considerably slower than the normal mode. As a consequence, we consider the Java object instrumentation unusable outside a testing environment. A reimplementation of the Java object instrumentation sensors using the JVMPI has the potential to provide better performance results and make the Java object instrumentation usable during the normal operation of the monitored application [HQGM02].

Furthermore, we do not provide performance data for the DSC component instrumentation either, because it relies entirely on the CORBA middleware instrumentation for monitoring communication events, and thus does not introduce any significant additional overhead.

We measured the performance of the prototype implementation of the MODOCC system with the CORBA middleware instrumentation from two perspectives: (a) computational overhead and (b) communication overhead. Computational overhead characterizes the additional processing that monitoring requires. Communication overhead characterizes the additional communication resources (such as network bandwidth) that monitoring requires. In the next sections we describe the technological decisions we make, the setup for the measurements, the measuring of computational overhead, and the measuring of communication overhead.

7.5.1 Technological decisions

We use Java 2 (v1.3 and above) as the implementation technology for the MODOCC system.

We use the JacORB implementation of the CORBA standard. For building, configuring and deploying the MODOCC system we use the same approach as for the GMS (see Chapter 6), because we build the MODOCC system using the GMS as a basis. In Appendix C we discuss how developers can configure, deploy, and start the MODOCC system.

7.5.2 Setup

The setup consists of a test site including specific hardware and software, and of a measurement scenario, which we run in the test site with several different configurations of the MODOCC system.

Test site

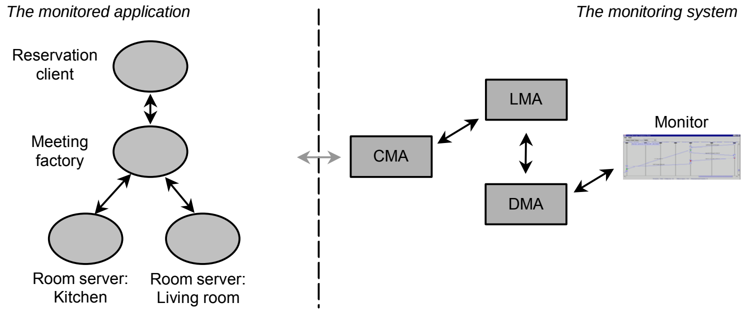
We used two PCs: PC (A) with a Pentium III running at 550 MHz and 256 MB of physical memory, and a PC (B) with a Pentium IV at 1.7 GHz and 512 MB of physical memory.

Both PCs had MS Windows 2000 Professional installed with SUN Java Virtual Machine version 1.4. We also installed Ethereal 0.9 [Eth03] to monitor network traffic.

Measurement scenario

For a monitored application we use the “room booking example” application [VoDu98]. We distribute an implementation of this example together with the MODOCC source code at [MODOCC].

Figure 7-18 Demo setup



The room booking example application allows for making reservations of rooms for conducting meetings (for whatever purpose). A separate *room server* represents each room. A special *meeting factory server* has the responsibility to create and cancel meetings. The *reservation client* application shows to the user an interface that allows him/her to choose a room and a time slot and to create a meeting, or to browse and cancel existing meetings.

On the side of the monitored application, our deployment of the example has two rooms, “Kitchen” and “Living room”, which the user can reserve for meetings. We deploy the two room servers and the meeting factory server on PC (B) and the client application on PC (A). On the side of the monitoring system, our deployment has one LMA per execution environment, one LMA per host, one DMA on PC(A), and one monitor on PC(B).

We define a test scenario in which the client “Books” and subsequently “Cancels” meeting on all time slots of all rooms, and repeats this 100 times.

We ran the same scenario with six different configurations of the room booking example and the MODOCC monitoring system (see Table 7-7).

Table 7-7 Six configurations

Name	Description
unmonitored original	The original application
unmonitored instrumented	The instrumented original, but MODOCC inactive
monitored light deferred	MODOCC active, timestamps generated using physical clock, using caching
monitored light direct	MODOCC active, timestamps generated using physical clock, sending data directly
monitored heavy deferred	MODOCC active, timestamps generated using logical clock, using caching
monitored heavy direct	MODOCC active, timestamps generated using logical clock, sending data directly

The first configuration represents the original application that we want to monitor. This configuration gives us a reference point for estimating the performance of the monitored application when monitoring with the MODOCC system.

The “unmonitored instrumented” configuration represents the instrumented monitored application however, with an inactive monitoring system. This configuration allows for measuring the overhead of an inactive instrumentation on the monitored application.

The next four configurations represent an active monitoring system.

The “monitored light deferred” configuration represents the instrumented monitored application, in which we generate timestamp values using the physical computer clock, and we use the *low-priority forwarding* policy (see section 7.4) for delivering monitoring data to the GMS.

The “monitored light direct” configuration represents the instrumented monitored application, in which we generate timestamp values using the physical computer clock, and we use the *immediate forwarding* policy for delivering monitoring data to the GMS.

The "monitored heavy deferred" configuration represents the instrumented monitored application, in which we generate timestamp values using the logical vector clock system, and we use the low-priority forwarding policy for delivering monitoring data to the GMS.

The "monitored heavy direct" configuration represents the instrumented monitored application, in which we generate timestamp values using the logical vector clock system, and we use the immediate forwarding policy for delivering monitoring data to the GMS.

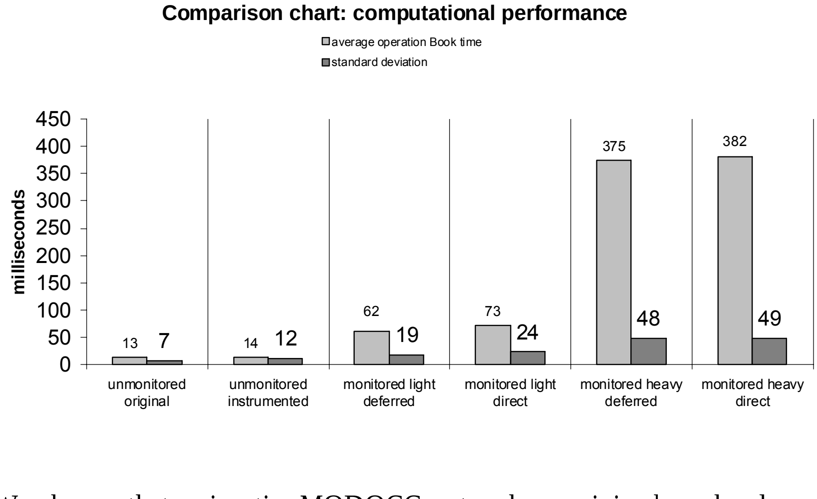
The "light" and "heavy" configurations allow for assessing the difference in overhead between using the inaccurate computer clocks and the logical vector clock for generating timestamps.

The "deferred" and "direct" configurations allow for assessing the difference in overhead between using the two different forwarding policies for sending monitoring data to the GMS.

7.5.3 Computational overhead

We estimated the computational overhead of each configuration by measuring the response times for each "Book" operation at the client (the "Cancel" operation shows similar results). Figure 7-19 shows a summary of these measurements.

Figure 7-19
Computational performance

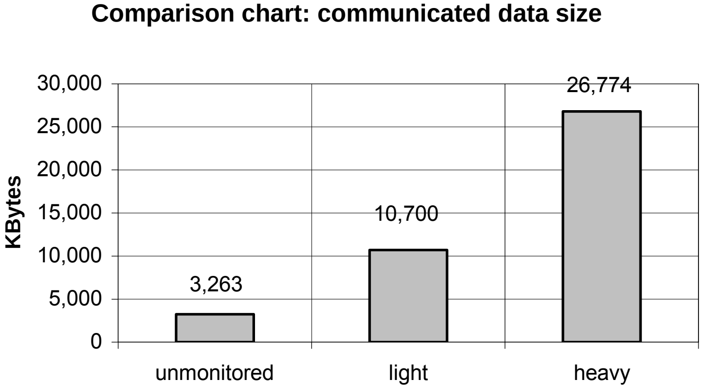


We observe that an inactive MODOCC system has a minimal overhead. When monitoring using timestamps generated from the physical clock however, the MODOCC system starts adding some overhead. Caching the event reports before sending them reduces the overhead a bit (the "deferred" configurations). When the MODOCC system starts using the logical clock however, we can see a significant increase of overhead per operation. This results mainly from the computational complexity of the metrication rules of the vector clock system. Remember that each “book” operation invocation results in four event reports, each of which contains a logical timestamp generated using our vector clock implementation. Nevertheless, we measure a relatively low deviation from the average response time, which means that although we have high overhead, we can at least predict how much this overhead changes within certain boundaries.

7.5.4 Communication overhead

Using the Ethereal network analyzer we measured the size of the data exchanged over the network by the components of the monitored application under the different configurations. We define three cases for measuring communication size. The “unmonitored” case represents the first two configurations. We consider these two configurations together because they do not generate any additional traffic – we do not activate the monitoring system for these configurations. The “light” case represents the second two configurations. We consider these two configurations together because they generate the same traffic – the *low-priority* and *immediate* forwarding policies only concern when the instrumentation sends monitoring data to the GMS but do not change the size of monitoring data. The same applies for the “heavy” case. Figure 7-20 and Figure 7-21 show the results of these measurements.

Figure 7-20 Total size of communicated data



As we can see, executing the same scenario in the three cases generates different amounts of monitoring data. In terms of kilobytes, the traffic of the MODOCC system in the “light” case equals roughly three times the traffic of the unmonitored case. This difference in traffic size comes from the additional invocations that the monitoring system performs during the dissemination of the event reports.

The "heavy" case generates roughly 2.5 times the traffic of the "light" case. The number of event reports stays precisely the same however, in the "heavy" case the size of the logical vector timestamp of each event report changes proportionally with the number of new processes in the monitored application.

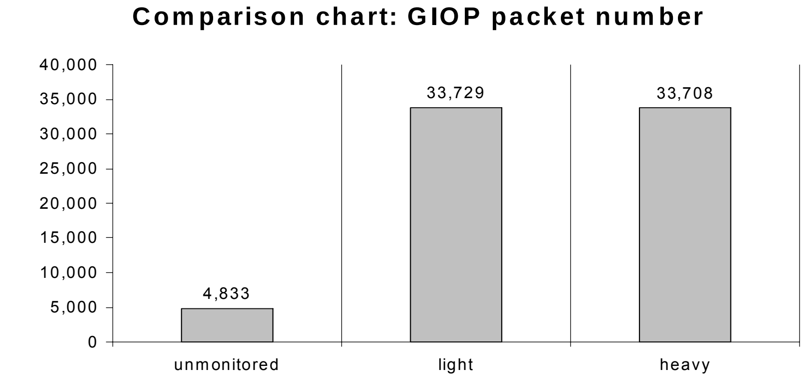


Figure 7-21 Size of CORBA communication (GIOP packets)

In this setup of the testing environment, each invocation performed by the monitored application (for its behaviour) or the monitoring system (for disseminating event reports), roughly fits into a single GIOP packet, almost without the need for GIOP fragmentation (one invocation to span over two packets). This diagram shows that the number of GIOP packets (and hence operation invocations) does not change between the "light" and the "heavy" cases. The small difference of 21 GIOP packets comes from connection reestablishments with the naming server resulting from some connection timeouts. We disregard these as having insignificant effect on the measurements.

7.5.5 Concluding remarks

The current prototype of the MODOCC system produces high overhead in our "toy" test application. This requires careful consideration about how to use our monitoring system with a larger and more complex "real life" application.

Monitoring of "real" applications

A "real" application may have a large number of remote objects that frequently perform operation invocations on each other.

Monitoring using the physical computer clock produces (relatively) constant computational overhead per event report, because generating the event report's timestamp from the physical computer clock takes constant time. The communication overhead per event report also stays relatively constant because the average size of an event report does not change over time (for simplicity we assume that a communication event report does not encapsulate the parameters of an operation). Therefore, the overall overhead of the monitoring system stays proportional to the number of invocations (communication event reports) and new objects (lifecycle event reports). Therefore, we expect that the performance of a large monitored application will deteriorate proportionally to its growth in terms of new remote objects and an increasing average of the number of invocations per second in the application.

Monitoring using the logical clocks adds additional computational overhead to the previous case of using physical clocks. In this case, the monitoring system requires additional processing that leads to a significant increase of computational overhead per event report, because of the algorithms required for maintaining a logical clock system. In contrast to the previous case however, the communication overhead per event report does not stay constant but grows with the number of processes in the system – every new process (typically at least one per remote object) increases the dimension of the vector clock system by one. Furthermore, the logical clock system requires sending of system clock information along with every operation (piggybacking). Therefore, we expect that the performance of a large monitored application will deteriorate at an rate, proportional to the number of processes in the system.

To help reason about how one can alleviate monitoring overhead we look at the lifecycle of monitored applications.

Monitoring during testing

We observe that in a testing environment, testers can tolerate certain delay resulting from the computational overhead of the MODOCC system, as opposed to the normal environment in which users have stricter requirements on application response time. Therefore, testers can afford to use the accurate logical timestamps to provide partial order and causal information among event reports occurring in the monitored application.

Monitoring during usage

Monitoring during the normal operation of the monitored application may become impossible because of the potentially drastic performance deterioration resulting from the monitoring overhead introduced by the monitoring system.

We consider several ways to alleviate monitoring overhead during normal application usage: periodical monitoring, partial monitoring, offline monitoring, and push-filtering.

In periodical monitoring, an administrator (or a tester) activates the monitoring system only for a limited period of time, e.g., based on the current load of the monitored application, habitual intervals of its users during which they do not use the monitored application that much (meals, evenings, weekends), etc. Periodical monitoring allows monitoring of the monitored application at times when the overhead would not lead to breaking any service contracts with application users. The MODOCC monitoring system can dynamically turn on and off generation of monitoring data.

In partial monitoring, the monitoring system only observes a part of the monitored application. For this, an administrator activates the instrumentation in some interesting parts of the application. The selection of which part to monitor may depend again on the current load of that part, but also on some high-level knowledge about, for example, the potential source of an error. Partial monitoring allows monitoring of suspected application parts, so that monitoring overhead does not affect the whole application. The MODODCC prototype supports flexible configuration of its instrumentation so that only parts of the application produce monitoring data.

We discussed offline monitoring in Chapter 2. In offline monitoring, the monitoring system does not immediately transport event reports to monitors. Instead, after generating an event report, the monitoring system saves it locally. Later, at a convenient moment (e.g., during the night or during times of low load) the monitoring system transports the accumulated data to the monitors. This removes (or delays) the communication overhead resulting from sending event reports around, but in the case of using logical clocks, the logical clock system still requires piggybacking of system information with every invocation so that newly generated event reports get correct logical timestamps. Note that the MODOCC system prototype can support offline monitoring through re-configuring the way the instrumentation sends the monitoring data to the GMS.

In push-filtering, the monitoring system allows monitors to express their interest in certain events in the monitored application, using a filtering language. The monitoring system then treats a specification of monitor interest as configuration information, which it pushes to the instrumentation. The instrumentation then activates only the sensors that produce relevant notifications. Note that although the GMS that we use takes into account specification of interest, the current prototype of the

MODOCC system does not support it. We plan to introduce this functionality in the future. Push-filtering reduces monitoring overhead by making sure that the monitoring system produces only necessary information.

Note that one can combine the mechanisms for alleviating monitoring overhead we discussed so far in order to reduce overhead further when compared to using only one mechanism. For example, combining offline monitoring, with partial and periodical monitoring has the potential to yield low overhead, which makes the monitoring system useful during the normal operation of a monitored application.

Furthermore, we have not optimized the current prototype for performance. We believe that a re-implementation of the vector clock system for performance may give some reduction of the computational overhead of the monitoring system.

In Chapter 2 we pointed out several approaches for reducing the size of logical timestamps. While these approaches provide some reduction of the communication overhead, they do not solve the major problem that the size of the timestamps stays roughly proportional to the number of processes in the monitored application.

Chapter 8

A monitor and monitoring applications

This chapter presents the design and the implementation of the Message Sequence Diagram (MSD) monitor for the MODOCC monitoring system. The MSD monitor provides a basis for the development of variety of monitoring applications. We design the MSD monitor following the design approach presented in Chapter 5.

We evaluate the information consistency provided by the MSD monitor and the MODOCC system from the perspective of monitor users using the method presented in Chapter 2, section 2.8. In order to validate our approach to monitoring, we present three applications of this monitor and the MODOCC monitoring system. We close the chapter with some concluding remarks.

8.1 MSD monitor

This section presents the design and the implementation of the Message Sequence Diagram (MSD) monitor for the MODOCC monitoring system.

8.1.1 Functional requirements

In Chapter 7 we defined a MODOCC monitoring model that describes single application runs in terms of objects, component instances, processes and the communication and lifecycle activities they perform. The MODOCC MM defines a set of event types that the MODOCC monitoring system can detect and offer to monitors. We would like to develop a generic MSD monitor that uses events of these types to allow application developers analyze the communication among object and among component instances during application runtime, for the purpose of testing and debugging application behavior. Based on requirements identified in Chapter 4 and on the MODOCC MM defined in Chapter 7, we phrase the following functional requirements for the MSD monitor:

1. The MSD monitor should allow online monitoring, i.e., it should depict new information as soon as it becomes available to the monitor;
2. The MSD monitor should support offline monitoring in the following way: it should provide a persistent storage for accumulated monitoring data, so that users can analyze monitoring data at a later moment;
3. The MSD monitor should provide interoperability with other communication analysis tools at the level of the data format used for persistent storage, such as based on the ITU-T Z.120 [Z.120] or XMI [XMI];
4. The MSD monitor presents information coming from the MODOCC system in the following way: it offers a diagram similar to the UML message sequence diagram and the ITU-T Z.120 Message Sequence Chart. The diagram depicts the entities and their communication as the monitoring application executes;
5. The MSD monitor should provide means for inspection of all parameters of the event reports coming from the MODOCC system;
6. The MSD monitor should properly restore the partial order of the causal precedence relation (Chapter 2, section 2.2.3) from the timestamps attribute of the event reports. The MSD monitor should use this partial order to position events and the invocations in which they participate, in a message sequence diagram. The MSD monitor orders unrelated events using the physical timestamp (the one generated using the computer clock) of the monitoring report, to approximate the (possibly inaccurate) relative position of the events in the diagram;
7. The MSD monitor should properly correlate the events constituting an individual invocation using the LRC relation (Chapter 7, section 7.2.2), and should properly visualize the dynamics of an invocation's during runtime: its initiation, its execution and the returning of a result;
8. The monitor should allow one to query the partial order topology imposed on the events in an application run by using the causal precedence relation. Possible queries include the generation of the set of possible causes of an event and the set of possible effects of an event. The monitor can depict these sets using alternative coloring for the event in the set.

8.1.2 Monitor implementation

Figure 8-1 shows the architecture of the MSD Monitor.

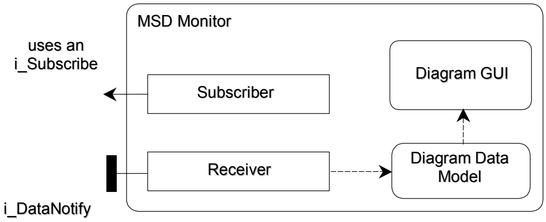


Figure 8-1 The architecture of the MSD Monitor

The MSD Monitor subscribes for monitoring data with the MODOCC system through its Subscriber component. The Subscriber component uses the i_Subscribe interface offered by a DMA of the MODOCC system (see Chapter 6, section 6.3.2). To establish a successful subscription, the Subscriber registers with the MODOCC system an i_DataNotify interface offered by the Receiver component. The MODOCC system then uses this interface to send event reports to the Receiver component.

We structure the rest of the monitor using the Model-View-Controller (MVC) design pattern [BuMe + 96], where the Receiver component represents the Controller responsible for the input (new event reports from the MODOCC system), the Diagram Data Model component represents the Model responsible for processing and storing all monitoring data in the MSD monitor, and the Diagram GUI component represents the View that presents monitoring data to monitor users. When the Receiver component receives new event reports, it passes them to the Diagram Data Model component for processing. The Diagram Data Model component performs all necessary processing activities and updates the Diagram GUI component to show any new information on screen. In the next section we discuss each of the components.

The Receiver component

The Receiver component uses an algorithm developed in [RST91] to deliver event reports to the Diagram Data Model component according to their causal precedence order. Basically, this algorithm delays the delivery of an event report until the Receiver component has delivered all causally preceding event reports. This makes sure that the Diagram Data Model component maintains a consistent view of the monitored system at any moment. An example of a similar approach constitutes the work described in [Logean00] where the monitor performs event report reordering as part of testing the prototypes of telecommunication services.

The Diagram Data Model component

The Diagram Data Model component (DDM) maintains an *ordered* collection of event reports, which represents an (partial) execution of the monitored application. Since the use of a linear array suggests an absolute order, but the vector timestamp that the MODOCC system uses provides only partial order, we use the physical clock timestamp that every event report also has (because event reports derive from monitoring reports of the GMS – consult Chapters 6 and 7) to determine the order among causally unrelated events.

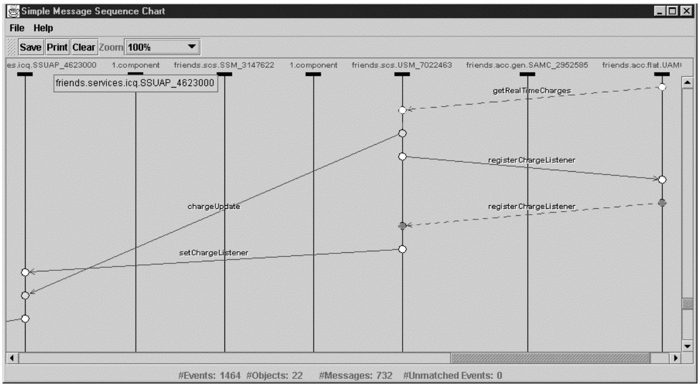
The DDM maintains a list of the objects and a list of the components in the current application execution.

The DDM also maintains a list of the messages exchanged between the entities of the model. The DDM uses two types of messages, a *request* message represents the successful receiving of an invocation (synchronous or oneway) at the server side, and a *response* message represents the successful receiving of a response (synchronous invocation) at the client side. According to the MODOCC MM, a synchronous invocation corresponds to a pair of request and response messages and an oneway invocation corresponds to a single request message. To match events into messages, the DDM restores the LRC relation (see Chapter 7 for the definition).

The DDM updates the Diagram GUI component in the following cases: it has received a new communication or lifecycle event, and it has matched two communication events in a new message.

The Diagram GUI component

The Diagram GUI component offers a diagram (a graphical representation) of the information maintained in the DDM. The diagram has an organization similar to an UML message sequence diagram and the ITU-T Z.120 Message Sequence Chart (Figure 8-2).

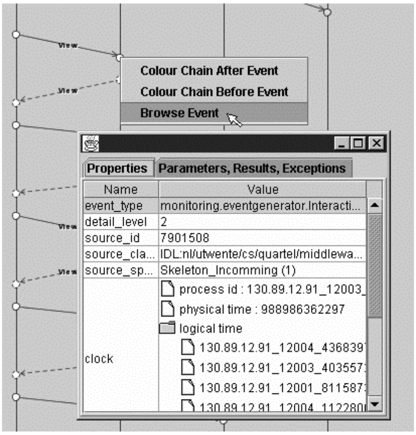


#Events: 1464 #Objects: 22 #Messages: 732 #Unmatched Events: 0

Figure 8-2 The diagram

The horizontal axis of the diagram represents the entities in the current application execution. For CORBA-based applications, the diagram shows CORBA objects. For DSC-based applications the diagram shows component instances. The vertical axis of the diagram represents time. Each entity has a vertical line representing its lifecycle. The diagram represents an event with a circle on the lifecycle line of the entity where this event has occurred. A white circle represents communication events participating in a synchronous invocation. A red (in the figure – dark grey) circle represents a communication event of an invocation response with exception. A green (in the figure – grey) circle represents an event participating in an oneway invocation. A solid arrow represents a request message. A dashed arrow represents a response message. We depict the name of the invocation operation in the middle of a message arrow.

The user of the MSD Monitor can interact with the diagram. Besides scrolling to view portions of a large diagram, the user can click the right mouse button on an event to open a popup menu that offers three options: “color chain after event”, “color chain before event” and “browse event” (Figure 8-3). By selecting the “color chain after event” the user highlights the set of events representing possible effects of the selected event. By selecting “color chain before event” the user highlights the set of events that represent the possible causes of the selected event. The line color of highlighted events the arrows between them (if any), changes to purple to illustrate the chain of cause or effect starting from the selected event. By selecting “browse event”, a new window opens showing a table of all event attributes. With this option, developers can inspect the parameters, result or exceptions (if any) at any monitored moment of the execution of an invocation.



Name	Value
event_type	monitoring.eventgenerator.Interacti...
detail_level	2
source_id	7901508
source_cla...	IDL:nl/utwente/cs/quartel/middlewa...
source_sp...	Skeleton_Incomming (1)
clock	process id : 130.89.12.91_12003_ physical time : 988986362297 logical time 130.89.12.91_12004_436839 130.89.12.91_12003_403557 130.89.12.91_12001_811587 130.89.12.91_12004_112280

Figure 8-3 Event inspection

Furthermore, the user can zoom the diagram to fit more information onto one screen, clear the diagram from old events and messages, print the diagram in a pane format to allow assembly of large diagrams, save the diagram for later, and open a saved diagram to finish postponed work.

The MSD Monitor can export the diagram to a file in the ITU-T Z.120 standard for message sequence charts. This feature allows for analysis of a monitoring trace into other tools compatible with this standard. In section 8.2.2 we shall present an application that uses this feature of the monitor.

Technology decisions

We implemented the MSD Monitor in Java. The MSD Monitor uses the JacORB CORBA product to interact with the MODOCC system.

8.1.3 Information consistency

In Chapter 7 we evaluated the performance of the MODODCC monitoring system from the point of view of the users of the monitored application. In this section we quantify the information consistency of the MODOCC monitoring system from the point of view of the users of the monitor – we call these users *observers*. We use the evaluation method presented in Chapter 2, section 2.8.

Minimal correctness

The MODOCC monitoring system detects all events that we have identified in the MODOCC monitoring model and reports them to the MSD Monitor accordingly. Hence $E_C = E_{M,C} = E_{O_M}$, where E_C represents the set of events in the original unmonitored and un-instrumented distributed computation, $E_{M,C}$ represents the set of events in the monitored instrumented distributed computation, and E_{O_M} represents the set of events in the distributed computation that the MSD monitor presents to the observer. Therefore, our monitoring system fulfils the *minimal non-interference* and *minimal accuracy* and hence *minimal correctness* properties.

Total accuracy

Our monitoring system uses a *strongly consistent* vector clock system for event timestamping, which allows for restoring the causal precedence relation among events monitored in the instrumented application. This makes our monitoring system *totally accurate*.

Better than minimal correctness

In Chapter 7 we defined the *LRC* relation for some object middleware model that governs how objects and component instances communicate. Since the selected evaluation method works with communication events only, we consider from the *LRC* relation only the relations among communication events – we define $LRC_{comm} = LRC \setminus \{(e_1, e_2) : Type(e_1) \in Types_{life} \text{ and } Type(e_2) \in Types_{life}\}$. From the definition of the *LRC* relation we have that $R(LRC_{comm}) \subset R_C$, where $R(LRC_{comm})$ represents the transitive reduction of the LRC_{comm} relation. We designed the MODOCC system to preserve the *LRC* relation. This means that the relations among events participating in an invocation do not change because of monitoring. Therefore $R(LRC_{comm}) \subset R_{M,C}$, and from the total accuracy property ($R_{M,C} = R_{O_M}$) we have that the $R(LRC_{comm}) \subset R_{O_M}$. This effectively means that the MODOCC monitoring system preserves some part of the relations in the original application behavior (the ones between events from the middleware), all the way to the observer. Therefore, we can say the MODOCC monitoring system provides better than *minimal non-interference* (but not *strong non-interference*), and from the total accuracy property follows that the MODOCC system provides better than *minimal correctness* (but not *strong correctness*).

Discussion on strong and total correctness

Strong non-interference and strong accuracy imply strong correctness, and total non-interference and total accuracy imply total correctness. We demonstrated that a monitoring system can offer *total accuracy* by employing a strongly consistent system of vector clocks to describe the order the events in the monitored application execution.

In order to provide strong non-interference, the monitoring system needs to provide realized causality among events that belong to different individual invocations. This means that the monitoring system must have access to an accurate and detailed model of the monitored application behavior, so that it can enforce the correct order among events in the monitored computation.

In order to provide total non-interference, the monitoring framework should not introduce any detectable monitoring overhead, in addition to the requirements for strong non-interference. One can achieve zero monitoring overhead only if designers use specialized hardware to detect the necessary events without introducing any additional delay to the monitored application. We consider this approach achievable, although developing hardware instrumentation generally involves higher costs than software instrumentation. Furthermore, hardware instrumentation falls out of our scope.

8.2 Concrete monitoring applications

In this section we present the use of the MODOCC monitoring system and of the MSD Monitor in three different monitoring applications.

8.2.1 Monitoring for testing and debugging of middleware-based applications

Component developers and system integrators used the MODOCC system and the MSD monitor within the FRIENDS project [FRIENDS] to improve the quality of application prototypes.

Overview of the FRIENDS project

The FRIENDS project has as a goal to develop a flexible and extensible software platform that provides an integrated solution to deployment, creation, and usage of services and applications for next-generation networks [FRIENDS].

The FRIENDS service platform architecture represents a component-oriented implementation of the Telecommunications Information Networking Architecture (TINA) [Kri97]. The FRIENDS service platform architecture consists of various layers of components, ranging from service session control components, to network control components. The

FRIENDS project builds on the results of the Multimedia services on the Electronic Super Highway (MESH) project [MESH].

Monitoring DSC

The Distributed Software Component (DSC) framework provides the building blocks for the FRIENDS services platform. The DSC framework, together with its development environment supports the development of component-oriented services [VBM00]. Within the DSC framework, we used the MODOCC system to build a testing environment that allows service integrators to analyze service behavior in terms of how different service components interact with each other [DiBa + 00][DSQ00].

The testing environment uses the DSC component instrumentation (introduced in Chapter 7) and the MSD monitor (introduced earlier in this chapter). For FRIENDS services, the MSD diagram shows component instances as entities on its horizontal axis.

In addition to the MSD diagram, for monitoring of FRIENDS services, the MSD monitor provides an additional diagram – the Dynamic Service Deployment (DSD) diagram (Figure 8-4). The DSD diagram uses the additional information from the TINA-based FRIENDS service architecture, in order to show how TINA-specific component instances interact during runtime.

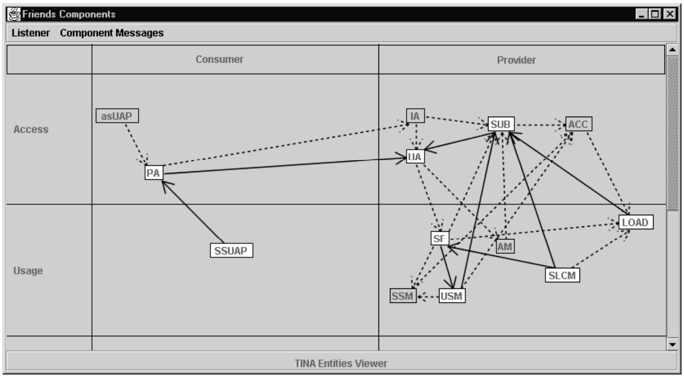


Figure 8-4 Dynamic service deployment diagram

The DSD diagram shows TINA service component types as rectangles with text inside, and their communication dependencies as arrows connecting the rectangles. The diagram's horizontal axis groups component types according to (a) their role in the TINA architecture: consumer or provider, while the diagram's vertical axis groups the components according to (b) their participation in TINA service access and usage sessions. When a component instance of a certain type communicates with a component instance of another type, an arrow appears on the diagram and the rectangles representing the communicating component types become white. The diagram shows live animation: an arrow between component types, the instances of which have not interacted for some time, fades to a dashed line arrow, and the rectangle of a component type that hasn't communicated with any other component type for a while changes its color from white to gray. When the DDM matches two events in a message, it notifies the DSD diagram the same way it notifies the MSD diagram. Hence the DSD diagram presents to the observer a different view on the same application execution.

FRIENDS service developers used the DSD diagram to visually represent the dynamics of their prototypes in terms of interacting TINA components. We used the DSD diagram for presentation purposes only.

Component developers and service integrators in FRIENDS used the MSD diagram to achieve detailed visualization of the interactions among various component instances.

The diagrams produced by the MSD monitor helped developers discover errors in the implementation of the service platform and the services build for this platform: (1) by finding an operation invocation that did occur, but under the circumstances shouldn't have occurred, (2) finding that wrong order of calling operations results in the use of variables with incorrect values, and (3) designers could trace (by using the coloring causality chain option) operation invocations that caused an erroneous invocation in order to examine the parameters of these operation invocations and determine the reasons for the error.

Furthermore, the diagrams produced by the MSD monitor helped service integrators to check whether the components of the different service prototypes behaved according to the sequence of interactions defined in the TINA architecture.

8.2.2 Semi-automatic conformance testing

Researchers within the FRIENDS project have also developed a model-based approach to service creation [TeQua01]. This approach allows one to check the conformance of a service prototype to its formal model.

Introduction to service creation in FRIENDS

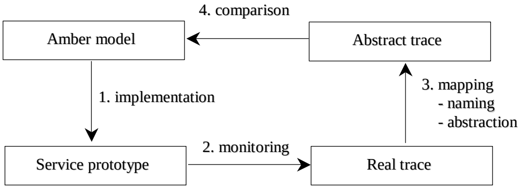
The FRIENDS services platform provides integrated support for service creation in a so-called service creation environment. The service creation environment enables a service developer to design and implement the requested service in an efficient and cost-effective way. The service creation environment promotes a model-based approach, which allows modeling of the complete external behavior of each of the components that contribute to a service, defining both the operations of its interfaces and the operations invoked by this component on interfaces of other components, as well as the relationships between these operations and their parameters [TeQua01].

Using the model-based approach, the FRIENDS service creation platform supports *design time* and *runtime* analyses of service properties. The Testbed Studio [FraJa98] constitutes an essential tool in the service creation platform, which supports various *design time* analyses. Testbed Studio supports the editing of service specifications in the formal specification language AMBER [EJ + 99], including syntax and semantics checking, and adds a number of analysis tools, such as step-wise simulation, quantitative analysis, integrated use of the model checker SPIN [Holz97], and several kinds of generated views on a model of a service. The tools for *runtime* analysis of service properties in the FRIENDS service creation platform allow for testing the conformance of a service prototype behavior to its behavior model.

Using monitoring for testing service behavior

Figure 8-5 depicts how FRIENDS testers perform conformance testing.

Figure 8-5 A method for conformance testing



1. Developers use an AMBER model to implement a prototype;
2. Testers perform a particular scenario of service use on the resulting prototype in the FRIENDS testing environment. This results in a *real trace*. The MSD monitor saves the real trace in the ITU-T Z.120 file format, in order to allow an independent (from the monitoring system) analysis tool to further analyze the trace. The real trace contains the request and response messages exchanged among component instances;
3. An analysis tool transforms the real trace into an *abstract trace* that designers can check against the AMBER model. The transformation involves rising the level of abstraction from the implementation level to the model level, e.g. by removing interactions that appear too detailed for the model level. Furthermore, the transformation clears any naming

inconsistencies, so that names of trace entities uniquely correspond to names of the model entities;

4. Using the simulator of the Testbed Studio, designers check whether the AMBER model allows each message from the abstract trace in a step-wise manner. We say that the abstract trace conforms to its model, if the AMBER model allows all messages in the order indicated by the abstract trace.

Tools support the activities in the whole process. The mapping of a real trace to an abstract trace may require an amount of manual work, depending on the additional refinement during the development of the AMBER model into a software implementation of the service prototype.

By testing the conformance of multiple traces produced by various use scenarios, testers can increase their confidence in the correctness of a service prototype.

8.2.3 Validation of the UMTS Application Platform

Lucent Technologies Bell Labs carried out the UMTS Application Platform project. This project produced a software platform that uses monitoring of interactions among CORBA objects for testing and presentation purposes [WPU01].

Overview of the UMTS Application Platform

The UMTS Application Platform project has as a goal the design and implementation of a testbed for the development and deployment of end-user applications for the third generation of mobile communication systems – UMTS.

The UMTS application platform has to provide support for the standardized interfaces required to enable interoperability and portability of end-user applications within the telecommunication's business domain. The project considers as important standards in this respect the Open Services Access standard [3GOSA] and the Presence and Availability Management standard [PAM].

Figure 8-6 Software architecture of the UMTS application platform

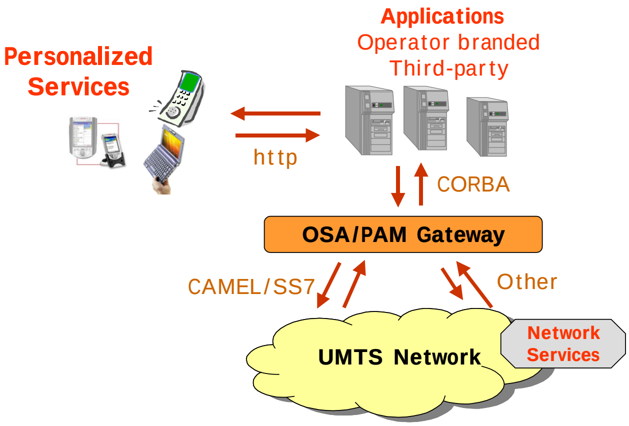


Figure 8-6 shows the architecture of the UMTS application platform. The OSA/PAM Gateway constitutes the main component of the UMTS application platform architecture. The OSA/PAM Gateway offers abstraction from the UMTS network layer by implementing the OSA standard and the PAM standard as well as several other telecom standards. The UMTS application platform designers implemented the OSA/PAM Gateway using Java and CORBA.

Monitoring the UMTS platform

The Java-based CORBA instrumentation provided by the MODOCC system allowed seamless integration with the UMTS platform prototype. The automated tool for generation of the object instrumentation minimized the effort from platform developers to prepare the platform prototype for monitoring.

The designers of the UMTS application platform used the MODOCC system to enhance the testing process during the platform development. The UMTS application platform project uses telecommunication standards that the standardization bodies have specified using, among other descriptive techniques, UML message sequence diagrams. The testers compared the message sequence diagrams generated by the MSD monitor with the UML message sequence diagrams from the standard, in order to determine whether their prototype correctly implements the standards.

In the occasion testers discovered discrepancies between the standard and the implementation, they further looked in the detailed information generated by the MODOCC monitoring system to find out about concrete errors, this way cutting down the time for delivery of a working prototype compliant to international telecom standards.

8.3 Summary and conclusions

In this chapter we have presented the MSD monitor for the MODOCC system. Furthermore, we have described three applications of the MSD monitor. The first application integrated monitoring into a component development platform to enhance the testing phase of component-based software development. The second application illustrated how designers can use monitoring at the level of application components for semi-automated conformance testing of prototype behavior against models of their intended behavior. The third application showed the usefulness of monitoring for increasing the quality of critical and large business applications, such as an UMTS application platform that has to conform to international telecommunication standards.

With these applications we have shown that designers can use the MODOCC systems for monitoring of different middleware-based applications without additional development effort, investment in monitoring software, and research in the area of monitoring. The MSD monitor allows developers to examine the behavior of a prototype in order to locate and remove implementation errors, check conformance with a model, analyze causal relationships, and animate object and component communication on screen for presentation purposes.

Chapter 9

Conclusions

In this thesis we presented an approach for monitoring the communication behavior of distributed applications built with object and component middleware. In this chapter we present conclusions about the work done and the results achieved. First we summarize the major research contributions of this thesis. Then we identify some remaining problems and directions for future work in the area.

9.1 Contributions

This work makes the following contributions to the area of monitoring distributed applications.

9.1.1 Design approach

We propose a design approach for building monitoring systems (Chapter 5). Compared to existing approaches, our design approach offers the following unique combination of features:

- Separation of concerns about generic monitoring functionality from concerns about functionality specific to the domain of the monitoring or the domain of the monitored application;
- Separation of concerns about the monitoring application from the concerns about the monitored application;
- Explicit modeling of the monitoring aspects of the monitored application execution, required by the monitoring application;
- Stepwise development of a complete monitoring system.

Following this approach, designers increase the “openness” of their monitoring system architecture, reduce development costs and possibly achieve a more efficient monitoring system.

9.1.2 MODOCC monitoring model

We define a model for monitoring communication in applications built with object and component middleware (Chapter 7). Our model has the following unique combination of features, compared to existing models:

- Support for monitoring of the communication behavior of objects, component instances and processes;
- An event-based model of communication behavior;
- Support for reasoning about causal relations among events in terms of both causal precedence and limited realized causality.

This model allows the application of existing formal methods to the analysis of concurrent execution at the level of object and component communication, such as conformance testing and verification.

9.1.3 A monitoring system for arbitrary middleware-based applications

We propose the MODOCC monitoring system for applications built with object and component technology (Chapter 7). The MODOCC monitoring system has the following unique combination of features compared to existing monitoring techniques:

- Independence of middleware software vendors and particular object-oriented programming languages;
- A flexible and open-ended architecture for generic monitoring systems with a clearly defined system service (see also Chapter 6);
- An instrumentation and a set of automatic instrumentation tools for Java, CORBA, and the DSC framework.

Using the MODOCC system designers can concentrate on analysis, instead of spending too much time to prepare their applications for monitoring.

9.1.4 Limited Realized Causality (LRC) relation

We define a new type of causal relation – LRC (Chapter 7, section 7.2.2). The LRC has the following unique features compared to other (partial) order relations and potential causality relations:

- The LRC relation allows reasoning about application behavior from post-execution traces with the certainty of realized causality;
- The MODOCC monitoring system allows for restoring the LRC relation. We demonstrate that the LRC relation allows the MODOCC monitoring system to satisfy the stronger than minimal non-interference

property and thus stronger than minimal correctness (Chapter 8, section 8.1.3).

Although limited to events in the middleware, the LRC relation constitutes a powerful tool for analysis of application behavior.

9.1.5 A Message Sequence Diagram (MSD) monitor

We have designed and implemented the MSD monitor (Chapter 8). The MSD monitor has the following unique combination of features, compared to existing monitors:

- Visually represents the distributed execution of a middleware-based application as a collection of communicating objects or component instances;
- Uses a new visual notation, based on the UML Message Sequence Diagram and the ITU-T Z.120 Message Sequence Chart, but extended with additional dynamic features for online animation of object and component communication;
- Orders events in the system according to the causal precedence relation (partial order);
- Can match two corresponding communication events to a request or response message and corresponding request and respond messages to invocations using the LRC relation;
- Uses alternative coloring to visualize the chains of possible causes or effects for a selected event in a distributed execution.

Software developers have successfully used the MSD monitor to produce more reliable applications within the following research projects: MESH [MESH], FRIENDS [FRIENDS], and UMTS Application Platform [WPU01].

9.2 Future work

We identify the following directions into which the presented work can further expand:

9.2.1 Design approach

We believe that designers of monitoring systems can benefit from our design approach. Nevertheless, in its current state our design approach provides guidelines but not enough generic building blocks (such as architectural patterns) that designers can use to assemble a monitoring system – something necessary for a design methodology. Defining a design methodology for building monitoring systems based on our design approach constitutes an important original work in the area of monitoring distributed applications. This work should also provide sufficient evidence that the methodology provides the promised benefits, by applying it to the development of several different monitoring systems.

9.2.2 Monitoring model

We assume that the middleware used by the monitoring system faithfully implements its middleware model (i.e., “no debugging of the debugger”). Since the middleware model dictates which event definitely causes which other event, a correct implementation of the model allows us to restore the LRC relation among events representing communication activities in the middleware. We consider as a possibility for future research the extension of the LRC relation beyond the middleware layer and into the application layer. For this to work we have two requirements:

- Formal modeling of at least a part of the application behavior using a suitable design language. In Chapter 8, section 8.2.2, we demonstrated that a complete application model allows one to perform conformance testing in order to improve a prototype implementation of the model;
- A faithful implementation of that model (achieved by some means, e.g., conformance testing). This assumption allows the designer to distinguish the two possible sources of an observed erroneous behavior: (1) problems in the model (e.g., some issue not considered in the model, or a wrong model), (2) an implementation that does not conform to the model because of implementation error.

Using such an extended LRC relation, a monitor can analyze the causes of an event in the application model with the certainty of realized causality. This gives designers a powerful tool for understanding and improving their models.

9.2.3 The logical clock system

In Chapter 2 we point out the drawbacks of using vector clocks. Furthermore, in Chapter 7 we give evidence about the large overhead produced by the algorithms of the logical system. Possible future work constitutes the reduction of this overhead, by implementing a more efficient logical clock system than the vector clock.

In line of this work, one can also experiment with existing logical clocks systems that promise lower overhead, such as in clustering [Ward01]. Such approach however, needs to reevaluate the usefulness of the causal precedence relation for monitoring purposes, since the monitoring system can reconstruct causal precedence only for events belonging to the same cluster but not to different clusters.

9.2.4 The instrumentation implementation

We also consider an original future research the making of our instrumentation implementation more efficient. For the Java execution environment, an instrumentation based on the JVMPI interface will probably produce better results than an instrumentation based on the JDI interface, since the JVMPI does not require running the Java virtual machine in debug mode in order to install the proper monitoring sensors.

An implementation of the CORBA instrumentation in a programming language, such as C + +, that compiles to native binary code may reduce the overhead for middleware objects.

9.2.5 Experiment with source code modification

In Chapter 5 we discuss that source code modification provides the ultimate means for monitoring any aspect in the behavior of a monitored application. Instrumenting the source code by a third party designer however, may require a lot of knowledge about the monitored application. For this reason we consider as original future work the use of instrumentation tools for fast and correct source code instrumentation. In this respect, embedding sensors in the source code using an aspect-oriented approach, such as composition filters [Berg01], seems to us a promising direction for building instrumentation tools for source code instrumentation.

Appendix A

IDL interfaces of the GMS

File: modocc_types.idl

```
#ifndef modocc_types_idl
#define modocc_types_idl

module modocc
{
   //structures of the monitoring data
   struct _Attribute
   {
      string name;
      any value;
   };
   typedef sequence<_Attribute> AttributeList;

   struct _Time
   {
      string physical_timestamp;
   };
   struct _Address
   {
      string host_name;
      string host_ip_address;
   };
   struct _System
   {
      long detail_level;
   };

   valuetype MonitoringReport
```

```
    {
        public _Time time;
        public _Address address;
        public _System system;
        public AttributeList s_attributes;
    };
    typedef sequence<MonitoringReport> MonitoringReportList;

    valuetype Event : MonitoringReport
    {
        public string name;

        factory create();
    };
    valuetype Status : MonitoringReport
    {
        public string name;

        factory create();
    };
    //structures of the availability information
    enum ReportTypes { report_event, report_status };
    enum TypesUpdateStatus { data_added, data_removed, data_changed };

    struct NamedOutput
    {
        string name;
        string type;
    };
    typedef sequence<NamedOutput> NamedOutputList;

    struct MonitoringReportType
    {
        string name;
        ReportTypes type;
        NamedOutputList attributes;
    };
    typedef sequence<MonitoringReportType> MonitoringReportTypeList;

    typedef sequence<string> StringList;
    typedef sequence<StringList> StringListList;
};
```

```
#endif
```

File: msap.idl

```
#ifndef msap_idl
#define msap_idl

#include <modocc_types.idl>

module modocc
{
   module msap
   {
      //forward declarations
      interface i_BrowseUpdate;
      interface i_DataNotify;

      //GMS interfaces
      interface i_Browse
      {
         /**
          * The request of the following synchronous operation represents
          * the "Browse interrogate req" service primitive.
          * The response of the following synchronous operation represents
          * the "Browse interrogate cnf" service primitive.
          */
         modocc::MonitoringReportTypeList interrogate(
            in string search_criteria);

         /**
          * The request of the following synchronous operation represents
          * the "Browse subscribe req" service primitive.
          * The response of the following synchronous operation represents
          * the "Browse subscribe cnf" service primitive.
          */
         boolean subscribe(in string monitor_id,
            in i_BrowseUpdate types_notification_reference);

         /**
          * The following oneway operation represents the
          * "Browse unsubscribe req" service primitive.
          */
         oneway void unsubscribe(in string monitor_id);
```

```
};
interface i_Subscribe
{
    /**
     * The request of the following synchronous operation represents
     * the "Request Data subscribe req" service primitive.
     * The response of the following synchronous operation represents
     * the "Request Data subscribe cnf" service primitive.
     */
    boolean subscribe(in string monitor_id,
        in string specification_of_interest,
        in i_DataNotify data_notification_reference);

    /**
     * The following oneway operation represents the
     * "Request Data unsubscribe req" service primitive.
     */
    oneway void unsubscribe(in string monitor_id);
};
interface i_Interrogate
{
    /**
     * The request of the following synchronous operation represents
     * the "Request Data interrogate req" service primitive.
     * The response of the following synchronous operation represents
     * the "Receive Data interrogate cnf" service primitive.
     */
   void interrogate(in StringList data_selection_criteria,
        in i_DataNotify data_notification_reference);
};

//Monitor interfaces
interface i_BrowseUpdate
{
    /**
     * The following oneway operation represents the
     * "Browse update ind" service primitive.
     */
    oneway void update(
        in modocc::TypesUpdateStatus update_status);
};
interface i_DataNotify
{
```

```
            /**
             * The following oneway operation represents the
             * "Receive Data notify ind" service primitive.
             */
            oneway void notify(
                in modocc::MonitoringReportList monitoring_data);
        };
    };
};
#endif
```

File: isap.idl

```
#ifndef isap_idl
#define isap_idl

#include <modocc_types.idl>

module modocc
{
    module isap
    {
        //GMS interfaces
        interface i_Announce
        {
            /**
             * The request of the following synchronous operation represents
             * the "Announce register req" service primitive.
             * The response of the following synchronous operation represents
             * the "Announce register cnf" service primitive.
             */
             boolean register(in string instrumentation_id,
                in string specification_of_availability);

            /**
             * The following oneway operation represents the
             * "Announce unregister req" service primitive.
             */
             oneway void unregister(in string instrumentation_id);
        };
        interface i_SendNotify
        {
            /**
```

```
             * The following oneway operation represents the
             * "Send Data notify req" service primitive.
             */
            oneway void notify(
                in modocc::MonitoringReportList monitoring_data);
        };

        //CMA interfaces
        interface i_Configure
        {
            /**
             * The request of the following synchronous operation represents
             * the "Configure configure ind" service primitive.
             * The response of the following synchronous operation represents
             * the "Configure configure rsp" service primitive.
             */
             boolean configure(in StringList configuration_specification);
        };
        interface i_SendInterrogate
        {
            /**
             * The request of the following synchronous operation represents
             * the "Send Data interrogate ind" service primitive.
             * The response of the following synchronous operation represents
             * the "Send Data interrogate rsp" service primitive.
             */
            modocc::MonitoringReportList
            interrogate(in StringList data_selection_criteria);
        };
    };
};
#endif
```

File: gms_internal.idl

```
#ifndef gms_internal_idl
#define gms_internal_idl

#include <modocc_types.idl>
#include <msap.idl>

module modocc
{
```

```
module repository
{
    interface ii_Repository
    {
        /**
         * Given a list of event report names, this operation
         * returns a list of their descriptions.
         */
        boolean get_event_description1(in StringList event_list,
            out MonitoringReportTypeList descriptions);

        /**
         * Given a event report name pattern, this operation
         * returns a list of descriptions of events
         * matching the pattern.
         */
        boolean get_event_description2(in string event_pattern,
            out modocc::MonitoringReportTypeList descriptions);

        /**
         * Given a list of status report names this operation
         * returns a list of their descriptions.
         */
        boolean get_status_description(in StringList status_names,
            out MonitoringReportTypeList descriptions);

        /**
         * Given a list of event report names, this operation returns
         * a structure containing the unique identifies of the
         * instrumentation instances that can generate these
         * events.
         */
        void get_ids_for_events(in StringList event_names,
            out StringListList instrumentations);

        /**
         * Given a list of status report names, this operation returns
         * a structure containing the unique identifies of the
         * instrumentation instances which allow measuring of
         * these statuses.
         */
        void get_ids_for_statuses(in StringList event_names,
            out StringListList instrumentations);
    };
};
module dissemination
{
    interface ii_ConfigureInstrumentation
    {
        /**
         * Given a list of event report names, this operation
         * configures the instrumentation to start
         * producing events of the required types.
         */
        boolean switch_on(in string monitor_id,
            in StringList event_names);

        /**
         * Given a list of event report names, this operation
         * configures the instrumentation to stop
         * producing events of the required types.
         */
        boolean switch_off(in string monitor_id);
    };
    interface ii_ConfigureDelivery
    {
        /**
         * This operation adds a monitor to the event delivery
         * mechanism
         */
        void add_monitor(in string monitor_id,
            in modocc::msap::i_DataNotify data_notification_reference);

        /**
         * This operation removes a monitor from the event delivery
         * mechanism
         */
        void remove_monitor(in string monitor_id);
    };
    interface ii_DeliverEvent
    {
        /**
         * This operation delivers a list of events to a list of
         * interested monitors
         */
        oneway void deliver_events(in modocc::StringList monitor_ids,
                in modocc::MonitoringReportList event_list);
        };
    };
    module filtering
    {
        interface ii_ConfigureFiltering
        {
            /**
             * This operation adds to the filtering engine a
             * filter corresponding to a monitor
             */
            void add_filter(in string monitor_id, in any filter_tree);

            /**
             * This operation removes from the filtering engine a
             * filter corresponding to a monitor
             */
            void remove_filter(in string monitor_id);
        };
    };
};
#endif
```

Appendix B

How to use the GMS prototype

In this appendix we present several tasks important for the proper usage of the GMS prototype: building, configuration and deployment, and startup.

Building the prototype

Although no international standardization organization has published any formal standards for setting up of an open source project, the open source community has come with informal recommendations, such as a publicly available source tree, binary distributions, list of dependent technologies, and a minimal documentation that allows developers to build and run an open source project. We consider SourceForge [SF] a good source with information on starting such projects. We use SourceForge to host the development of the GMS prototype as a part of the MODOCC system (see Chapter 7).

For providing the building process we have chosen the Ant technology [ANT]. Ant stands for an extensible build tool entirely written in Java. It has the portability and platform independence of Java. Furthermore, Ant uses an XML-based specification for describing builds. This specification does not depend on the features of a particular software execution platform (except Java). Hence it does not rely on command shell scripting or other OS features. The Ant allows for building of the GMS on any platform with available Java 2 virtual machine.

We have made available to the general public the whole project source code and documentation as part of the MODOCC project [MODOCC]. Executing of the ANT XML specification automatically takes care of the proper order of compilation, building and packaging of system components. Designers can easily adapt the XML script to their needs if they want to modify or reuse parts of the GMS in other projects.

Configuration and deployment

Designers can perform the system configuration of the MODOCC system using a special configuration specification that describes the deployment of GMS monitoring agents. This specification reflects the hierarchical distribution model presented earlier. The specification itself represents an XML document validated by this XML Schema. The following XML Schema defines the structure of an configuration specification XML document:

conf_spec.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="conf_spec">
        <xs:annotation>
            <xs:documentation>Comment describing your root
element</xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:sequence maxOccurs="unbounded">
                <xs:element ref="realm"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="realm">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="prop" type="property" minOccurs="0"
maxOccurs="unbounded"/>
                <xs:element ref="domain" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="domain">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="prop" type="property" minOccurs="0"
maxOccurs="unbounded"/>
                <xs:element ref="host" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="host">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="prop" type="property" minOccurs="0"
maxOccurs="unbounded"/>
                <xs:element ref="unit" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="unit">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="prop" type="property" minOccurs="0"
maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="property">
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="value" type="xs:string" use="required"/>
    </xs:complexType>
</xs:schema>
```

An example of a valid configuration specification XML document we provide in Appendix C.

During runtime, a software component called `ConfSpecMgr` processes the configuration specification and provides monitoring agent with configuration information. Suffice to say, the `ConfSpecMgr` provides access to a logically centralized repository for configuration information within the GMS. The configuration information organizes properties (name/value pairs) into a hierarchy of domains (corresponding to the domains from the hierarchical distribution model) containing hosts (corresponding to locality regions), which contain units (corresponding to co-location regions). Physical components of the monitoring system running in an unit can use the properties defined at the level of the unit, and the properties defined at the level of the host that executes the unit, and the properties defined at the level of the domain to which that host belongs. Designers can override property values. For example, a property defined in a domain, can have its value overridden by a property with the same name defined in a host, contained in this domain. Analogously, a property defined in a host, can have its value overridden by a property with the same name defined in a unit on that host. This allows for a convenient organization of the properties needed by many components belonging to the same region, but allows the definition of regions of exceptions within the containing region.

Startup

Designers cannot execute a GMS by itself because they first need to build an application of it for a particular monitored application. We present an application process for our GMS prototype in the next chapter. After having created a specialization of the GMS, the developers define a configuration specification, and then run the monitored application together with the monitoring system.

Designers perform the startup process using the hierarchical order of the configuration: first they need to instantiate the CORBA naming service, the DMAs and the DMA-DMA channels (CORBA notification service), than the LMA on each application host, then the instrumented parts (with the co-located CMAs inside) of the monitored application. In the future versions of the GMS prototypes, we plan to provide a deployment tool that uses the configuration specification to install and startup the application automatically.

Discovery of running agents presents an important issue in a distributed environment. The Naming Service represents a central repository for storing references of remote CORBA object. Each monitoring agent of the GMS contains several remote objects each implementing one interface of the agent. The Naming service manages the references to these objects so that other agents can access them. We consider this centralized approach for discovery of running instances of the system inappropriate for use in large distributed environments, where a centralized Naming Service may become a bottleneck and a single point-of-failure. Designers who desire a more flexible solution may combine an elaborated approach for the communication of the DMA, e.g. by using the JXTA [Wil02] peer-to-peer infrastructure, to perform dynamic and distributed discovery of other DMAs using the peer discovery protocol part of the JXTA. Designers can do the same for the discovery of LMA instances, however, we consider this unnecessary in most cases, because designers can organize LMAs in relatively small administrative domain regions each using its own instance of the Naming Service. For small domains the peer-to-peer protocols do not provide a significant benefit.

Appendix C

How to use the MODOCC prototype

Configuration and deployment

In Chapter 6 we discussed the GMS configuration specification. We defined the specification in XML and its structure using the XML Schema standard. Using this definition, the MODOCC system further defines several concrete properties, which designers can use to configure the MODOCC monitoring system (in addition to the basic configuration for the GMS). Bellow we list an example configuration file that describes a (static) deployment of a simple CORBA application prepared for monitoring with the MODOCC system. In the listing we highlight the important MODOCC specific properties.

roombooking.xml

```
    <?xml version="1.0" encoding="UTF-8"?>
    <conf_spec xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="./conf_spec.xsd">
        <realm id="modocc_testbed">
            <domain id="room_booking_session">
                <prop name="modocc.active" value="yes"/>
                <prop name="modocc.forwarding.policy" value="deferred"/>
                <prop name="modocc.externalization.policy" value="SendToLMA"/>
                <prop name="modocc.mode" value="heavy"/>
                <prop name="modocc.ntp" value="enabled"/>
                <prop name="modocc.ntp.server" value="timehost.cs.utwente.nl"/>
                <prop name="modocc.ntp.offset.refresh.time" value="3600000"/>
                <host id="10.10.1.3">
```

```
                    <prop name="modocc.externalization.resource"
value="LMA_ref"/>
                    <prop name="modocc.oa.port" value="5555"/>
                    <unit id="meeting_factory"/>
                    <unit id="room_server_1"/>
                    <unit id="room_server_2"/>
                    <unit id="room_booking_lma">
                        <prop name="modocc.lma.service" value="DMA"/>
                    </unit>
                    <unit id="room_booking_dma">
                        <prop name="modocc.oa.port" value="5554"/>
                    </unit>
                </host>
                <host id="10.10.2.2">
                    <prop name="modocc.externalization.resource" value="LMA_ref"/>
                    <prop name="modocc.oa.port" value="5555"/>
                    <unit id="room_client"/>
                    <unit id="room_booking_lma">
                        <prop name="modocc.lma.service" value="DMA"/>
                    </unit>
                </host>
            </domain>
        </realm>
    </conf_spec>
```

The following Table 12-1 describes each property. Some of these properties have implications on the performance of the MODOCC system and we will use them in section 7.6.

Table 12-1 MODOCC configuration properties

Property name	Description
modocc.active	Possible values: {yes, no*[9]}. Indicates whether the MODOCC system should monitor or not
modocc.forwarding.policy	Possible values: {deferred*, direct}. Indicates whether the CMA caches the monitoring data it needs to send to the LMA (deferred) in order to return the control to the monitored application as soon as possible or, sends the data immediately (direct) and waits for a response indicating successful receiving

[9] A * after a value indicates that the MODOCC system uses this value as a default in the case a configuration specification does not define the property for some reason.

modocc.externalization.policy	Possible values: {SaveToFile, SendToLMA*}. Indicates whether the CMA saves monitoring data locally to a file or sends it to the LMA
modocc.externalization.resource	In case the previous property indicates saving to a file, this property indicates the name of that file
modocc.mode	Possible values: {light*, heavy}. Indicates how the MODOCC system generates event timestamps: from the physical computer clock (light) or using the MODOCC vector clock implementation (heavy)
modocc.ntp	Possible values: {enabled, disabled*}. Indicates the use of the NTP protocol for synchronization of the physical computer clocks in the distributed system monitored with MODOCC
modocc.ntp.server	In case the previous property allows use of NTP, this property holds an URL to the NTP server (e.g., a LAN time server, or a server somewhere on the Internet hooked to an atomic clock)
modocc.ntp.offset.refresh.time	This property indicates the time between two subsequent synchronizations using NTP. Normally, computer clocks do not drift away from each other too fast so MODOCC uses a default value of one hour
modocc.oa.port	This property indicates the CORBA port for MODOCC internal CORBA objects. Default value may differ between ORB vendors

For the correct deployment of the MODOCC system designers need to provide a command line parameter “-Dmodocc.config.file” to the Java interpreter running every instrumented application part that indicated the file or URL where every CMA instance can load the configuration specification. Furthermore, designers have to make sure they have configured the ORB to install the CMAORBInitializer as an orb initializer so that it can install the necessary monitoring interceptors, using the ORB’s Portable Interceptors API.

Starting the MODOCC system

To start the MODOCC system, designers first need to start the monitoring agents of the GMS in the sequence indicated in Appendix B. Then they can start the parts of the monitored application. This will automatically instantiate a CMA at each location. Then, the MODOCC system becomes operational and users can start a monitor to perform some analysis and presentation on information coming from the MODOCC system.

References

On some occasions, we provide URLs for easy access to online versions of works cited. For the full text of some of the cited works, you may need an account for the ACM Digital Library or the IEEE Computer Society Digital Library.

[.NET] Microsoft .NET technology. http://www.microsoft.com/net/, 2002

[.NETST] Visual Studio .NET. http://msdn.microsoft.com/vstudio/, 2002

[3GOSA] 3rd Generation Partnership Project, Technical Specification Group and System Aspects. "Virtual Home Environment / Open Service Architecture" (Release 1999). 3GPP TS 23.127 V3.3.0 (2000-12)

[AdSi95] Adelstein, F., Singhal, M. "Real-Time Causal Message Ordering In Multimedia Systems". In the Proceedings of the International Conference on Distributed Computing Systems, pp. 36-43, 1995.

[AkTri88] Aksit, M., Tripathi, A. "Data Abstraction Mechanisms in Sina/st", ACM Object-Oriented Programming Systems, Languages and Applications Conference Proceedings, September, 1988, pp. 267-275.

[AMIDST] http://amidst.ctit.utwente.nl/, 1999-2002.

[ANT] The Jakarta ANT project, http://jakarta.apache.org/ant/index.html, 2003

[APPC] AppCenter. http://www.borland.com/appcenter/index.html, 2002

[Arbab95] Arbab, F. Coordination of massively concurrent activities. CWI report CS-R9565, 1995.

[ARM98] "Systems Management: Application Response Measurement (ARM) API". Open Group Technical Standard C807, The Open Group, 1998. http://www.opengroup.org/onlinepubs/009619299/.

[BaBa98] Bakker, J.L., H. Batteram. "Design and evaluation of the Distributed Software Component framework for distributed communication architectures". 2nd Intl. Workshop on Enterprise Distributed Object Computing (EDOC'98), San Diego, USA, Nov. 3–5, 1998, pp. 282-288.

[Bach86] Bach, M.J. "The design of the UNIX Operating System". Prentice Hall, 1986.

[Bates85] Bates, P. "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior", ACM Transactions on Computer Systems, Vol. 13, Issue 1, February, 1995, pp. 1-31

[BeAb02] Benslimane, A., Abouaissa, A. “Dynamical grouping model for distributed real time causal ordering”. In Computer Communications, Volume 25, Issue 3, pp. 288-302, 2002

[BeHa + 00] Bergmans, L., van Halteren, A., Ferreira Pires, L., van Sinderen, M., Aksit, M. A “QoS-Control Architecture for Object Middleware”. Proceedings of the IDMS’2000 workshop, Springer Verlag, 2000,pp.117-131.

[Berg01] Bergmans, L., Aksit, M., “Composing crosscutting concerns using composition filters”, Communications of the ACM, October 2001, Vol. 44, No.10, pp. 51-57.

[Bertin99] Bertin, J. “Sémiologie Graphique”. Les Re-impressions des Editions de l'Ecole des Hautes Etudes En Sciences Sociales, 1999

[Bi + 00] Birman, K., Constable, R., Hayden, M., Kreitz, C., Rodeh, O., van Renesse, R., Vogels, W. “The Horus and Ensemble Projects: Accomplishments and Limitations”. Proc. of the DARPA Information Survivability Conference & Exposition (DISCEX '00), South Carolina, USA, January, 2000.

[BiJo87] Birman, K., Joseph, T. “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.

[BiJo97] Birman, K.P., Joseph, T.A. “Reliable communication in the presence of failures”. In ACM Transactions of Computer Systems, Volume 5, Issues 1, pp.47-76, 1987.

[Blair98i] Blair, G. S., Coulson, G., “The case for reflective middleware”, report.nr. MPG-98-38, Distributed Multimedia Research Group, Lancaster University, 1998.

[Blair98ii] Blair, G. S., Coulson, G., Robin, P., Papathomas, M. “An Architecture for Next Generation Middleware”, In the proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Lake District, UK.

[BMC00] “Application Service Management: Reducing Complexity in Today’s Distributed Environmemt” — White Paper, March 2000. http://www.bmc.com/products/whitepaper.html.

[Booch91] Booch, G. “Object-Oriented Design with Applications”. The Benjamin/Cummings Publishing Company, Inc, 1991.

[Bour91] Bourland, D., “To Be or Not: An E-Prime Anthology”, Intl Society for General Semantics, October, 1991.

[BrM02] Brörkens, M., Möller, M. “Dynamic Event Generation for Runtime Checking using the JDI”. In the workshop on Runtime Verification’2002, Copenhagen, Denmark, July, 2002.

[Buch98] Buchanan, M. “Quantum Teleportation”, New Scientist, 14 March, 1998.

[BuMe + 96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. “Pattern-Oriented Software Architecture, A System of Patterns”. John Wiley & Sons Ltd., 1996

[CCM] CORBA Component Model Specification. http://www.omg.org/cgi-bin/doc?ptc/2001-11-03, 2001

[CES] CORBA Event Service, http://www.omg.org/cgi-bin/doc?formal/2001-03-01

[ChKo00] Chen, G. & Kotz, D. “A Survey of Context-Aware Mobile Computing Research”, Technical report TR2000-381, Dept. of Computer Science, Dartmouth College, November, 2000.

[CLSLDR] http://java.sun.com/j2se/1.4/docs/api/java/lang/ClassLoader.html, 2002

[CNS] CORBA Notification Service, http://www.omg.org/cgi-bin/doc?formal/2002-08-04

[COM+] Kirtland, M. “Object-Oriented Software Development Made Simple with COM+ Runtime Services”, Microsoft Systems Journal, 1997. http://www.microsoft.com/msj/1197/complus.aspx

[CoMa91] Cooper, R., and Marzullo, K. “Consistent Detection of Global Predicates”. In the proceedings of ACM orkshop on Parallel and Distributed Debugging, Santa Cruz, California, pp. 163-173, May 1991.

[CORBA] *http://www.omg.org/technology/documents/formal/corba_2.htm*, CORBA/IIOP Specification 2.6, 2001

[CORBA3] CORBA 3.0. http://www.omg.org/cgi-bin/doc?formal/02-06-33, 2002

[Crow96] Crowcroft, J. “Open Distributed Systems”. Artech House, February, 1996.

[CTRACE] CorbaTrace. http://corbatrace.tuxfamily.org/

[DCE] Distributed Computing Environment. http://www.opengroup.org/dce/, 2001

[DCOM] Distributed Component Object Model. http://www.microsoft.com/com/tech/DCOM.asp, 2002

[DiBa+00] Diakov, N.K., Batteram, H. J., Zandbelt, H., Sinderen, M. J., “Design and Implementation of a Framework for Monitoring Distributed Component Interactions”, Proceedings of the 7th International Workshop, IDMS'2000, pp. 227-240, Enschede, The Netherlands, October 17-20, 2000

[Diet00] Dietrich, F. “Modeling Object-Oriented Communication Services with Temporal Logic”. PhD Thesis, Thesis No. 2141, Swiss Federal Institute of Technology, Lausanne, 2000.

[DMTF] Distributed Management Task Force. www.dmtf.org, 2002

[DSQ00] Diakov, N.K., Sinderen, M. J., Quartel, D., “Monitoring Extensions for Component-Based Distributed Software”, Proceedings of the Protocols for Multimedia Systems, PROMS'2000, pp. 417-424, Cracow, Poland, October 22-25, 2000

[EfCh+01] Efstratiou, C., Cheverst, K., Davies, N., Friday, A. “An Architecture for the Effective Support of Adaptive Context-Aware Applications”. In the proceedings of the Second International Conference on Mobile Data Management (MDM’2001), Springer- Verlag (LNCS 1987), pp.15-209, Berlin Heidelberg, Germany, 2001, pp. 15-27.

[Gu+95] Gu, W., Eisenhauer, G., Kraemer, E., Stasko, K., Vetter, J., and Mallavarupu, N. “Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs," in Proc. FRONTIERS'95, pp. 11-19, February 1995.

[EJ+99] Eertink, H., Janssen, W.P.M., Oude Luttighuis, P.H.W.M, Teeuw, W.B., Vissers, C.A. “A Business Process Design language”, Proceedings World Congress on Formal Methods (FM'99), 1999.

[EJB] Enterprise Java Beans specification. http://java.sun.com/products/ejb/docs.html, 2002

[Eth03] The Ethereal Network Analyzer. http://www.ethereal.com/, 2003

[FaDi00] Guareis de Farias, C., Diakov, N. “Component-based Groupware Tailorability using Monitoring Facilities”. Proceedings of CBG2000, the CSCW2000 workshop on Component-Based Groupware, Telematica Instituut, The Netherlands, 2000, pp. 16-21

[FelEr89] Feldkuhn, L. Erickson, J. “Event Management as a Common Functional Area of Open Systems Management”. Proceedings of the IFIP Symposium on Integrated Network Management, Boston, North-Holland, 1989, pp. 265-276.

[FiMi82] Fischer, M.J., Michael, A. “Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network”. In the proceedings of ACM Symp. Principle Database Systems, ACM Press, New York, 1982, pp.70-75.

[FoZw90] Fowler, J., Zwaenepoel, W. “Causal Distributed Breakpoints”. In the proceedings of the 10th International Conference on Distributed Computing Systems, 1990, pp.134-141.

[FraJa98] Franken, H.M., and Janssen, W. “Get a grip on changing business processes Results from the Testbed-project”, Knowledge & Process Management (Wiley), vol. 5, no. 4, December 1998, p. 208-215.

[FRIENDS] http://www.telin.nl/Middleware/FRIENDS/ENindex.htm, 1999-2000.

[GEC98] Gershon, N., Eick, S. G., Card, S. “Information Visualization”, Interactions, Vol. 5, Issue 2, ACM Press. New York. March/April, 1998.

[Geist+94] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V. “PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing”. MIT Press, Cambridge, Massachusetts, 1994.

[GeRo97] Gennaro, R., Rohatgi, P. “How to Sign Digital Streams”. In the proceedings of “Advances in Cryptology” - CRYPTO'97, Lecture Notes in Computer Science , vol. 1295, Springer Verlag, August, 1997, pp. 180-197

[GiCo00] Gilfix, M., Couch, A. “Peep: Monitoring Your Network With Sound”. In the Proceedings of the 14th Systems Administration Conference – LISA’2000, December, 2000, pp. 109-117

[Giere98] Giere, R. N. “Understanding Scientific Reasoning”. Holt Rinehart & Winston, 4th edition, January, 1998.

[Gold84] Goldberg, A. “Smaltalk-80: The Interactive Programming Environment”. Reading, MA, Addison-Wesley, 1984.

[Google] www.google.com, 2002

[Halt03] Aart T. van Halteren. “Towards an adaptable QoS aware middleware for distributed objects”, CTIT PhD.-thesis series, ISSN 1381-3617 nr 02-46, ISSN 1388-1795 No. 008, ISBN 90-75176-35-X, University of Twente, The Netherlands, January 2003.

[HaSte97] Hamada, T., and F. Steegmans (ed.). “TINA-C, Network Components Specification”. TINA-C consortium, Redbank (NJ) USA, December 1997.

[HeBr89] Helmbold, D., Bryan, D. “Design of Run Time Monitors for Concurrent Programs”. Technical Report No. CSL-TR-89-395, Program Analysis and Verification Group, Computer Science Laboratory. Standford, California, USA, October, 1989.

[Heis27] Heisenberg, W. “The physical content of quantum kinematics and mechanics.” Translation by Wheeler, J.A. and Zurek, W.H., eds. Quantum theory and measurement. Princeton: Princeton University Press. 1983, pp. 62--84.

[HeMcD96] Helmbold, D.P., McDowell, C.E., “Race Detection - Ten Years Later”. in: Simmons, M.L., Hayes, A.H., Brown, J.S., Reed, D.A., "Debugging and Performance Tuning for Parallel Computing Systems", IEEE Computer Society Press, Los Alamitos, CA, USA, 1996, pp. 101-126.

[Hoare78] Hoare, C.A.R. “Communicating Sequential Processes”. Communications of the ACM, Vol. 21, Nr. 8. August, 1978.

[Hof+94] Hofmann, R., Klar, R., Mohr, B., Quick, A., Siegle, M. “Distributed performance monitoring: Methods, tools, and applications”. IEEE Transactions on Parallel and Distributed Systems, vol. 5, num. 6, pp. 585-598, 1994.

[Hoff94] Hoffner, Y., “Monitoring in Distributed Systems”. ANSA project architecture report, http://www.ansa.co.uk/, December 1994.

[Hold89] Holden, D. “Predictive Languages for Management”. In the proceedings of the IFIP symposium on Integrated Network Management, Boston, North-Holland, pp. 585-596.

[Holz97] Holzmann, G.J., “The model checker SPIN”. IEEE Trans. on Soft. Eng., vol. 23, no. 5, May 1997.

[HQGM02] Harkema, M., Quartel, D., Gijsen, B. M. M., van der Mei, R. D. “Performance Monitoring of Java Applications”. in Proc. of the 3rd Workshop on Software and Performance (WOSP 2002), ACM Press, Rome (Italy), July 2002, pp. 114-127.

[HriWo96] Hrischuk, C. E. & Woodside. C. “Proper time: Causal and temporal relations of a distributed system”. Technical Report SCE-96-04, Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, Mar. 1996. Submitted to IEEE Transactions on Parallel and Distributed Systems.

[HSV99] Hasan, M., Sugla, B., Viswanathan, R. “A Conceptual Framework for Network Management Event Correlation and Filtering Systems”. Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Management. May, 1999.

[IEEE98] “IEEE Standard for Software Test Documentation”, ANSI/IEEE Std. 829-1998, IEEE Standard Collection Software Engineering, 1998

[IONAMS] IONA Administrator Management Service. www.iona.com, 2002

[ISO87] Information processing systems - Open Systems Interconnection - Service conventions. Technical Report ISO TR 8509, ISO, 1987

[ISO90] ISO 10165-1 OSI Management Information Services - Structure of Management Information - Part 1, Management Information Model, 1990

[ISO92] ISO 10040: OSI Information Processing Systems, Management Overview, 1992

[ISO98] ITU/ISO, “Quality of Service – Framework”, ISO/IEC CD 13236, 1998.

[ISO9000] ISO standard 9000-2000. http://www.iso.org/, 2002

[ISOVOC] ISO standard 8402-1986 “Quality-Vocabulary”, 2002

[JACORB] The JacORB CORBA ORB implementation for Java, http://jacorb.org, 2003

[JaJou94] Jard, C., Jourdan, G.C., “Dependency Tracking and Filtering in Distributed Computations”. In Brief Announcements ACM Symposium on Principles of Distributed Computing, ACM Press, New York, 1994.

[JBDER] Borland JBuilder. http://www.borland.com/jbuilder/, 2002

[JBR99] Jacobson, I., Booch, G., Rumbaugh, J. “The Unified Software Development Process”. Addison-Wesley Pub Co, 1st edition, February, 1999.

[JD] Java Debugger, http://java.sun.com/j2se/1.4/docs/tooldocs/tools.html, 2001

[JLSU87] Joyce, J. Lomow, G., Slind, K., Unger, B. “Monitoring Distributed Systems”. ACM Transactions on Computer Systems, Volume 5, Issue 2, pp. 121-150, May 1987.

[JMS] Java Messaging Service API. http://java.sun.com/products/jms/, 2002

[JMX] Java Management Extensions. http://java.sun.com/products/JavaManagement/, 2002

[JORAM] JORAM. http://www.objectweb.org/joram/doc/index.html, 2002

[JPDA] Java Platform Debugger Architecture, http://java.sun.com/j2se/1.4.1/docs/guide/jpda/index.html

[JVMPI] Java Virtual Machine Profiler Interface, http://java.sun.com/j2se/1.4.1/docs/guide/jvmpi/index.html

[Kath00] O. Kath, A. v. Halteren, F. Stoinski, M. Wegdam, Mike Fisher, “Integrated Middleware Platform Management based on Portable Interceptors”, in Proc. 11th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 2000), LNCS 1960, Dec. 2000, Austin, Texas, USA.

[KBTB97] Kunz, T., Black, J.P., Taylor, D. J., Basten, T. “POET: TargetSystem -Independent Visualisations of Complex Distributed-Application Executions”. The Computer Journal, Volume 40, Issue 8, pp.499-512, 1997.

[KH + 01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G. “An Overview of AspectJ”. In Proceedings of the European Conference on Object-Oriented Programming, Budapest, Hungary, 18-22 June 2001, Lecture Notes in Computer Science, Volume 2072, pp 327-355, 2001.

[KhoCo86] Khoshafian, S., and Coperland, G. “Object Identity”. SIGPLAN Notices, vol. 21 (11), November, 1986.

[KQS92] Klar, R., Quick, A., Sötz, F. “Tools for a Model–driven Instrumentation for Monitoring”. In the proceedings of the 5th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Elsevier Science Publisher B.V., pp. 165–180, 1992.

[KR + 00] Kon, F., Roman, M., Liu, P., Mao, J., Yamane, T., Magalhães, L. C, Cambell, R. “Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB”. Middleware 2000 --- IFIP/ACM International Conference on Distributed Systems Platforms, volume 1795 of Lecture Notes in Computer Science, pages 71-87, Springer, April, 2000.

[Kranz00] Kranzlmüller, D. “Event Graph Analysis for Debugging Massively Parallel Programs”. PhD dissertation, GUP Linz, Joh. Kepler University Linz, Austria, September, 2000.

[Kranz97] Kranzlmüller, D., Grabner, S., and Volkert, J. “Debugging with the MAD environment”. Journal of Parallel Computing, 23, issue 1–2, pp. 199–217, April 1997.

[KraWis98] Krawczyk, H., Wiszniewski, B. “Analysis and Testing of Distributed Software Applications”, Research Studies Press Ltd., Baldoc, Hertfordshire, England, 1998.

[Kri97] Kristiansen, L. (ed.). “TINA-C, Service Architecture”. TINA-C consortium, Redbank (NJ) USA, June 1997.

[Lamp78] Lamport, L. “Time, Clocks, and the Ordering of Events in a Distributed System”, Communications of the ACM, Vol. 21, No. 7, July, 1978, pp. 558-564

[LBDK01] Laumay, P., Bruneton, E., De Palma, N., Krakowiak, S. “Preserving Causality in a Scalable Message-Oriented Middleware”. In the Proceedings of Middleware’2001, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, pp. 311-317, Germany, November 12-16, 2001.

[LDKK98] Logean, X., Dietrich, F., Karamyan, H., Koppenhöfer, S. “Run-time Monitoring of Distributed Applications”. Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing - Middleware’98, England, 15-18 September, 1998, pp. 459-473.

[LGPL] GNU Lesser GPL. http://www.gnu.org/copyleft/lesser.html

[Lieb86] Liebermann, H. “Using prototypical objects to implement shared behaviour in object oriented systems”. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1986, pp. 214-223.

[LiJo89] Lim, J., and Johnson, R. “The Heart of Object-Oriented Concurrent Programming”. SIGPLAN Notices, vol. 24 (4), April, 1989.

[Logean00] Logean, X. “Run-time Monitoring and On-line Testing of Middleware Based Communication Services”. PhD dissertation, Thesis No. 2137, Swiss Federal Institute of Technology, Lausanne, 2000.

[LWSB97] Ludwig, T., Wismüller, R., Sunderam, V., Bode, A. “OMIS - On-line Monitoring Interface Specification (Version 2.0)”. Technical University Munich, Report TUM-I9733, SFB-Bericht Nr. 342/22/97, Munich, Germany, 1997.

[Mao99] Mao, J. “Monitoring and Analyzing Method Invocations in the 2K Operating System”. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May, 1999.

[Marc86] Marcotty, M. & Ledgard, H. “The World of Programming Languages”, Springer-Verlag, Berlin 1986.

[McDH89] McDowell, C.E., Helmbold, D.P., “Debugging Concurrent Programs”. ACM Computing Surveys, Volume 21, Issue 4, pp. 593-622, December 1989.

[MESH] http://www.telin.nl/Mesh/, 1996-1999.

[Miller56] Miller, G. “The Magical Number Seven, Plus Minus Two: Some Limits on Our Capacity for Processing Information”. The Psychological Review, vol. 63 (2), March, 1956

[Mills92] Mills, D.L. “Network Time Protocol (Version 3) specification, implementation and analysis”. Network Working Group Report RFC-1305, University of Delaware, March 1992, 113 pp.

[MODOCC] The MODOCC project, http://modocc.sourceforge.net/, 2003

[M-W] http://www.m-w.com/. Merriam-Webster, Incorporated, January 2002.

[NTP] Network Time Protocol (Version 3), Network Working Group, rfc1305, http://www.faqs.org/rfcs/rfc1305.html, 2002

[Nutt75] Nutt, G.J. “Tutorial: Computer System Monitors”. IEEE Computer, vol. 8, nr. 11, pp. 51–61 November, 1975.

[OLT03] IBM Distributed Debugger and Object Level Trace Version 9.1 Documentation, online at: http://www-3.ibm.com/software/webservers/appserv/doc/v40/aee/index.html, 2003

[ORBacus] http://www.iona.com/products/orbacus_home.htm

[ORBIX] Orbix Web ORB. www.iona.com, 2001

[OsGl96] Oskarsson, Ö., Glass, R.L. “An ISO-9000 Approach To Building Quality Software”. Prentice-Hall, Inc. 1996.

[PAM] Presence and Availability Management (PAM) Forum, http://www.pamforum.org/, 2003.

[Parnas85] Parnas, D. “Software Aspects of Strategic Defense Systems”. Communications of the ACM, vol. 28 (12), 1985.

[Pras95] Pras, A. “Network Management Architectures”. Ph.D. thesis, Twente University, Enschede, 1995

[Pratt86] Pratt, V. “Modelling Concurrency with Partial Orders”. Int. Journal of Parallel Programming, Vol. 15, No. 1, pp. 33-71, February, 1986.

[PTM97] Protic, J., Tomaevic, M., Milutinovic, V. “Distributed Shared Memory: Concepts and Systems”. IEEE Computer Society, August, 1997.

[Put01] Putman, J. R. “Architecturing with RM-ODP”. Prentice Hall PTR, 2001

[Rack01] Rackl, G. “Monitoring and Managing Heterogeneous Middleware”. PhD thesis. Technischen Universität Munchën, 2001.

[Rack99] Rackl, G. “Multi-Layer Monitoring in Distributed Object Environments”. In the proceedings of the Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS’99), Kluwer Academic Publishers, Helsinki, 1999, pp. 265-270.

[Rait00] Raitalaakso, T. “Dynamic Visualization of C++ Programs with UML Sequence Diagrams”. Master Thesis, Tampere University of Technology, Tampere, Finland, March, 2000.

[RaLe97] Rathmayer, S., Lenke, M. “A Tool for Online Visualization and Interactive Steering of Parallel HPC Applications”. Proceedings of 11th International Parallel Processing Symposium, pp. 181-186, 1997.

[RaLu01] Rackl, G., Ludwig, T. “A Methodology for Efficiently Developing On-Line Tools for Heterogeneous Middleware”. In the proceedings of the HICSS-34 Conference, January, 2001.

[RaSi96] Raynal, M., M. Singhal. “Logical time: Capturing causality in distributed systems”. IEEE Computer 29, Feb. 1996, pp. 49-57.

[RFC2768] RFC2119, http://www.ietf.org/rfc/rfc2119.txt, 2002

[RFC2768] RFC2768, ftp://ftp.isi.edu/in-notes/rfc2768.txt, 2002

[RLRS00] Rackl, G., Lindermeier, M., Rudorfer, M., Süss, B. “MIMO --- An Infrastructure for Monitoring and Managing Distributed Middleware Environments”, Middleware 2000 --- IFIP/ACM International Conference on Distributed Systems Platforms, volume 1795 of Lecture Notes in Computer Science, pages 71-87. Springer, April 2000

[RMI] Java Remote Method Invocation. http://java.sun.com/products/jdk/rmi/, 2002

[Rosen96] Rosenberg, J.B. “How Debuggers Work: Algorithms, Data Structures, and Architecture”. John Wiley & Sons, New York, 1996.

[Royce87] Royce, W.W. “Managing the development of large software systems: concepts and techniques”. Proceedings of the Western Electronic Show and Convention (WESCON), pp. 1-p, 1970. Reprinted in: Proceedings of the 9th International Conference on Software Engineering (ICSE’87), pp. 328-338, 1987.

[RPC] ISO/IEC 11578:1996, “Remote Procedure Call”, Information technology, Open Systems Interconnection

[RST91] Raynal, M., Schiper, A., Toueg, S. “The causal ordering abstraction and a simple way to implement it”. Information Processing Letters 39, Elsevier Science Publishers, September, 1991, pp. 343-350.

[Rumb88] Rumbaugh, J. “Relational Database Design Using an Object-Oriented Methodology”. Communications of the ACM, vol. 31 (4), April, 1988.

[Sahai98] Sahai, A., Morin, C. “Towards Distributed and Dynamic Network Management”. In the proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS), New Orleans, USA, Feb 15-20, 1998.

[Sam95] Mansouri-Samani, M. “Monitoring of Distributed Systems”. PhD thesis. University of London, Department of Computing, 1995

[S3DR204] Surface 3D Release 2, 2004. http://www.traxxdale.de/

[SchMa94] Schwarz, R., Mattern, F. “Detecting causal relationships in distributed computations: in the search of the Holy Grail”. Distributed Computing, 7(3), pp. 149-174, 1994.

[SchSch01] Schantz, R. E., and Schmidt, D. C. “Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications,” Encyclopedia of Software Eng., Wiley & Sons, New York, 2001; also available at http://www.cs.wustl.edu/~schmidt/PDF/middleware-chapter.pdf.

[SF] The Source Forge, “A Home of Open Source Development”, http://sourceforge.net, 2003

[SHA95] SHA-1: Secure Hashing Algorithm, Federal Information Processing Standards Publication 180-1, April, 1995

[Shaer98] Al Shaer, E. S. “A Hierarchical Filtering Based Monitoring Architecture For Large Scale Distributed Systems”. PhD thesis. Old Dominion University, Norfolk, 1998

[Shaw81] Shaw, M. “ALPHARD: Form and Content”. New York, NY, Springer-Verlag, 1981.

[SiKs92] Singhal, M., Kshemkalyani, A. “An Efficient Implementation of Vector Clocks”. Information Processing Letters, Vol. 43, August, 1992, pp. 47-52.

[SILK] SILK software, www.segue.com, 2002

[Simon82] Simon, H. “The Sciences of the Artificial”. Cambridge, MA, MIT Press, 1982.

[SLC99] Schmidt, D.C., Levine, D.L., Cleeland, C. “Architectures and Patterns for High-performance”, Real-time ORB Endsystems, Advances in Computers, Academic Press, Ed., Marvin Zelkowitz, Volume 48, July 1999.

[Slo95] Sloman, M. “Management Issues for Distributed Services”. In the proceedings of IEEE Second International Workshop on Services in Distributed and Networked Environments (SDNE’95), Whistler, British Columbia, Canada, 5-6 June 1995, IEEE Computer Society Press, pp 52-59.

[SloMo89] Sloman, M., Moffett, J. “Managing Distributed Systems”. Imperial College, London, 1989.

[Smith80] Newton-Smith, W. “The structure of Time”. Routledge & Kegan Paul, London, July, 1980.

[SMST] SmartStubs. http://www.white-park.freeserve.co.uk/

[Snir+99] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J. “MPI: The Complete Reference”. Vol. 2, 2nd edition, MIT Press, Cambridge, Massachusetts, 1999.

[SNTP] Simple Network Time Protocol (Version 4), Network Working Group, rfc2030, http://www.faqs.org/rfcs/rfc2030.html, 2002

[SPS02] Stallman, R. M., Pesch, R., Shebs, S. “Debugging with GDB: The GNU Source-Level Debugger”. Free Software Foundation, January 2002.

[SQL99] ISO/IEC 9075-1:1999, “Information technology - Database language - SQL - Part 1: Framework” (SQL/Framework), 1999.

[SSRB00] Schmidt, D., Stal, M., Rohnert, H., Buschmann, F., “Pattern-Oriented Software Architecture”, Volume 2, Patterns for Concurrent and Networked Objects. John Wiley & Sons Ltd., 2000

[Ste97] Steegmans, F. (ed.). “TINA-C, Network Resource Architecture”. TINA-C consortium, Redbank (NJ) USA, February 1997.

[Stitt92] Stitt, M. Debugging – “Creative Techniques and Tools for Software Repair”. John Wiley & Sons, New York, 1992.

[STK96] Schade, A., Trommler, P., Kaiserswerth, M. “Object Instrumentation for Distributed Applications Management”, in “Distributed Platforms”, Chapman & Hall, pp. 173-185, London, 1996

[Sum92] Summers, J. A. “Precedence-preserving abstraction for distributed debugging”. Master’s thesis, University of Waterloo, Ontario, 1992.

[Szy98] Szyperski, C. “Component software: beyond object-oriented programming”. Addison-Wesley, USA, 1998.

[Tan92] Tanenbaum, A. S. “Modern Operating Systems”. Prentice Hall, 1992

[TeQu01] Teeuw, W. B., & Quartel, D. A. C. “Model-based service creation in the Friends project”. Proceedings 6th International Conference on Protocols for Multimedia Systems (PROMS 2001), October 17-19, Enschede, The Netherlands, 2001. Springer- Verlag (LNCS 2213), pp.192-209, Berlin Heidelberg, Germany, 2001

[TeQua01] Teeuw, W. B., Quartel, D. A. C. “Model-based service creation in the Friends project”. Proceedings 6th International Conference on Protocols for Multimedia Systems (PROMS 2001). Springer-Verlag (LNCS 2213), pp.192-209, Berlin Heidelberg, Germany, 2001

[TIVOLI] TIVOLI software from IBM, www.tivoli.com, 2003

[Tufte83] Tufte, E. “The Visual Display of Quantitative Information”. Graphics Press, Chelshire, 1983.

[UML1.4] Unified Modeling Language (UML), version 1.4, http://www.omg.org/technology/documents/formal/uml.htm, 2002

[VBM00] Verhoosel, J. P. C., Batteram, H. J., Millian, R. S. “The FRIENDS Platform: Conquering Complexity using Distributed Software Components”. Technical report, Lucent technology Software Symposium, April 2000.

[ViPi + 00] Vissers, C.A., Pires, L.F., Quartel, D.A.C., van Sinderen, M.J. “The Design of Telematics Systems”. Lecture notes, Twente University, Enschede, The Netherlands, November 2000.

[VoDu98] Vogel, A., Duddy, K. “Java programming with Corba”. John Wiley & Sons Inc, 2nd edition, March 1998.

[Ward01] Ward, P. A. S. “A Scalable Partial-Order Data Structure For Distributed-System Observation”. PhD dissertation, University of Waterloo, Ontario, Canada, 2001.

[WAS] WebSphere Application Server. http://www.ibm.com/software/webservers/appserv/, 2003

[Weg03] Wegdam, M. “Dynamic Reconfiguration and Load Distribution in Component Middleware”, PhD thesis, CTIT research series No. 03-50, Telematica Instituut Research Series No. 9., Enschede, The Netherlands, 2003

[Weide01] Van der Weide, Th. P. “Information Discovery. Lecture notes on Information Retrieval and Hypertext”, University of Nijmegen, April, 2001.

[WeiKu98] Weinreich, R., Kurschl, W. “Dynamic Analysis of Distributed Object-Oriented Applications”. In the Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS-31), Software Technology, Big Island of Hawaii, USA, January 6 - 9, 1998, IEEE Computer Society Press, pp. 386-399, 1998.

[WHAT] http://whatis.techtarget.com/. TechTarget, 2002

[Wil02] Wilson, B. J. “JXTA”. New Riders Publishing, June, 2002.

[WMI] Windows Management Instrumentation (WMI). www.microsoft.com, 2002

[WPU01] Wegdam, M., Plas, D-J., Unmehopa, M. “Validation of the Open Service Access API for UMTS Application Provisioning”. In Proceedings of the 6th International Conference on Protocols for Multimedia Systems (PROMS 2001), published by Springer as LNCS 2213, pp. 210-221, ISBN 3-540-42708-2, October 17-19, 2001, Enschede, The Netherlands.

[XMI] XML Metadata Interchange specification. http://www.omg.org/cgi-bin/doc?formal/2002-01-01, 2002

[Yas92] Yasuhiko, Y. “The Apertos Reflective Operating System: The Concept and Its Implementation”, In the proceedings of OOPSLA'92, October 1992.

[Z.120] ITU-T Z.120. Message Sequence Chart (MSC). ITU-TSS, Oct. 1996.

[Zor00] Zoraja, I. “Online Monitoring in Software DSM Systems”. PhD thesis, Technische Universität München, 2000.

Table of Figures

Index

Samenvatting

Dit proefschrift presenteert ons werk op het gebied van het observeren van gedistribueerde software applicaties (DSAs voor Distributed Software Applications). We leveren drie hoofdresultaten op: (1) een ontwerpbenadering voor het bouwen van “observatie” systemen, (2) een systeemontwerp voor *MOnitoring Distributed Object and Component Communication* (MODOCC) in applicaties die op middleware gebaseerd zijn en (3) een “proof-of-concept” implementatie van dit ontwerp.

Het observeren van de uitvoering van DSAs speelt een essentiële rol in het verbeteren van de kwaliteit ervan in termen van gebruikersverwachtingen, performance en betrouwbaarheid. Bijvoorbeeld: het observeren van de communicatie tussen onderdelen van een DSA levert informatie op die gebruikt kan worden voor het ontdekken van fouten en hun oorzaken, voor storing- en prestatieanalyses en voor het balanceren van de systeembelasting over de DSA onderdelen.

Ontwerpers en programmeurs bouwen vaak observatie-hulpsystemen om de test-, operatie- en onderhoudsfases van de levenscyclus van een DSA te ondersteunen. Een observatie systeem moet daarvoor modellen en mechanismen gebruiken die een consistente weergave van de DSA uitvoering behouden, en die waar nodig informatie verschaffen over de executie van de applicatie.

Dit proefschrift richt zich op het observeren van DSAs die gebouwd zijn met object- en componenttechnologieën, en vooral op de executie aspecten van objecten en componenten, zoals inter-object en inter-component interactie.

Het manuscript heeft de volgende structuur:

Hoofdstuk 1 introduceert het onderzoeksgebied en geeft een gedetailleerde uitleg over onze motivatie voor dit werk en stelt onze doelen vast.

Hoofdstukken 2 en 3 introduceren terminologie en concepten, die door het hele proefschrift gebruikt worden.

Hoofdstuk 2 presenteert de basisterminologie en fundamentele concepten op het gebied van het observeren van gedistribueerde software.

Hoofdstuk 3 geeft een overzicht van object- en component middleware technologiëen.

Hoofdstuk 4 beschrijft en evalueert enkele bestaande observatie systemen, en met name observatie systemen die geschikt zijn voor object en component middleware ondersteunen. Als resultaat van deze evaluatie definiëren we een aantal eisen voor ons observatie systeem.

Hoofstuk 5 beschrijft een ontwerpbenadering voor observatie systemen. Deze ontwerpbenadering bestaat uit vier fases: Algemeen Observatie Systeem (GMS voor Generic Monitoring System) ontwerp, GMS specialisatie, instrumentatie ontwerp en observatie ontwerp.

Hoofdstukken 6, 7 en 8 volgen onze ontwerpbenadering met als doel een systeem te bouwen voor het observeren van op middleware gebaseerde applicaties.

Hoofdstuk 6 stelt een architectuur van een GMS voor. Het GMS bevat alleen algemene eisen voor observatie. Dit hoofdstuk bevat tevens een rapportage over een prototype GMS dat we hebben gebouwd.

Hoofdsuk 7 presenteert het ontwerp van een MODOCC systeem. Het ontwerp omvat een model voor het observeren van de communicatie tussen objecten en componenten en de aanpassingen aan de middleware die nodig zijn het observatie proces. Tevens wordt hier de implementatie van het prototype van deze aanpassingen beschreven.

Hoofstuk 8 beschrijft het ontwerp en de implementatie van een basale monitor die de communicatie tussen objecten en componenten visualiseert. Verder bespreken we het gebruik van het MODOCC systeem voor drie verschillende observeringsapplicaties.

Hoofdstuk 9 bevat een samenvatting van dit proefschrift en bedisscussieerd mogelijke richtingen voor vervolgonderzoek.

Предговор

Тази дисертация третира проблеми, свързани с наблюдаването на разпределен софтуер. Основните резултати са три: (1) методология за проектиране на системи за наблюдение, (2) дизайн на система за наблюдаване на комуникацията в обектно-ориентиран и компонентно-ориентиран разпределен софтуер, и (3) реализация на тази система за наблюдение.

Наблюдаването на поведението на разпределен софтуер по време на неговото изпълнение играе съществена роля при подобряването на софтуерното качество от гледна точка на потребителски очаквания, бързодействие, и надеждност. Например, наблюдаването на комуникацията между физически разпределени софтуерни части предоставя информация, която може да се използва за откриване на грешки и за локализиране на техните източници, за анализ на повреди, за анализ на системно бързодействие, а също така и за балансиране на натоварването на отделни системни компоненти.

Дизайнери и програмисти често използват ситеми за наблюдение при тестване, употреба и поддръжка на разпределен софтуер. Те се нуждаят от системи, които предоставят консистентен модел на изпълнението на разпределения софтуер и, когато е необходимо, могат да показват тази информация по време на самото изпълнение на наблюдавания софтуер.

Тази дисертация фокусира върху наблюдаването на обектно- и компонентно-ориентиран разпределен софтуер и по-конкретно, върху наблюдаването на взаимодействия между обекти и взаимодействия между компоненти.

Дисертацията има следната структура:

Глава 1 въвежда читателя в изследователската област, посочва нашата мотивация и дефинира в детайл нашите изследователски цели.

Глави 2 и 3 въвеждат терминологията, която използваме в тази дисертация.

Глава 2 въвежда в областта на наблюдаване на разпределен софтуер.

Глава 3 запознава читателя с обектно-ориентираните и компонентните технологии.

В Глава 4 изследваме няколко от съществуващите системи за наблюдение. В резултат на това изследване дефинираме изисквания към нашата система за наблюдение.

Глава 5 описва нашата методология за построяване на системи за наблюдение. Тя се състои от четири фази: дизайн на системи с широко приложение, специализация на системи с широко приложение към определена област, дизайн на инструменти за измерване и дизайн на софтуерни монитори.

Глави 6, 7 и 8 следват фазите на нашата методология.

Глава 6 описва архитектура на система за наблюдение с широко приложение. Тази система задоволява единствено тези от нашите изисквания, които позволяват широко приложение. Тази глава също докладва реализацията на прототип на система за наблюдение с широко приложение.

Глава 7 представя дизайн на система за наблюдение на обектна и компонентна комуникацията в разпределен софтуер, базиран на обектно-ориентирани и компонентни технологии и дизайн и реализация на инструментите за измерване тази комуникация.

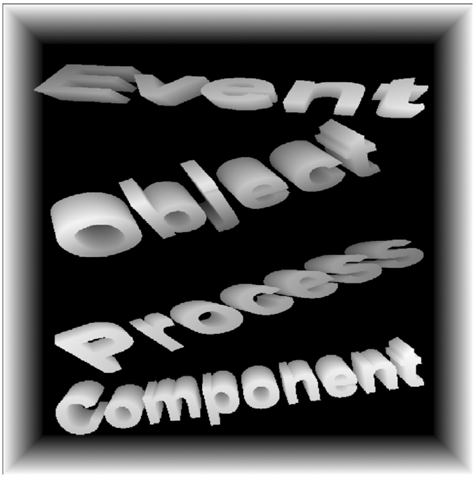
Глава 8 описва дизайн и реализация на софтуерен монитор, служещ за визуализиране на обектна и компонентна комуникацията. Тази глава представя също прилагането на нашата система за наблюдение и на софтуерния монитор към три различни проекта.

Глава 9 обобщава научния принос на тази дисертация и дискутира възможностите за продължаване на тази изследователска работа.

За Корицата

Изображението на корицата представлява стереограма: гледането на стереограма по специален начин разкрива тримерна сцена. Принципът, използван в стереогамите е формулиран за пръв път от физика Сър Чарлз Уетстоун през 1833. Според този принцип, когато човек гледа някаква сцена в околната среда, всяко око вижда отделно изображение на същата сцена. Човешкият мозък сглобява тези две изображение в нещо, което ние възприемаме като тримерно изображение. Стереограмите използват този принцип, за да скрият тримерна сцена в двумерно изображение.

Изображението на корицата съдържа следната тримерна сцена:



Следващите инструкции съставляват техника за гледане на стереограми:

1. Приближете стереограмата много близо до лицето си;
2. Отпуснете очите си и гледайте през стереограмата с очи, фокусирани зад равнината на стереограмата;

3. Бавно отдалечавайте стереограмата, като се старaете да не местите погледа си и да не променяте фокуса му.

Може да отнеме известно време, за да видите тримерната сцена. Също така отбележете, че само 85-90% от хората са в състояние да виждат тримерната сцена на една стереограма.

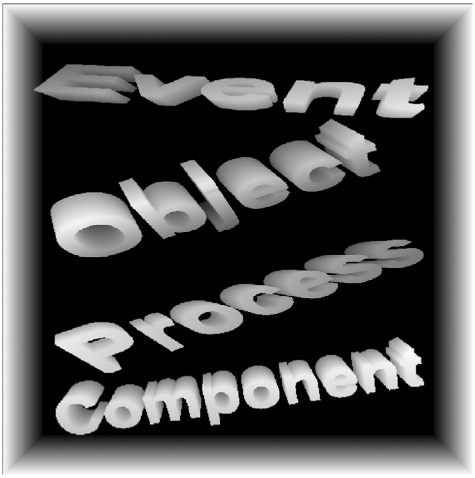
Поставих тази стереограма на корицата на дисертацията си, за да подчертая ролята на системите за наблюдение при производството на софтуер. Софтуерът сам по себе си не "съществува" в същия смисъл както една лъжица съществува физически. При това отбележете, че разпечатана програма "съществува" по-същия начин както и думата "лъжица" написана на лист хартия. Най-реалното нещо от един софтуер са електрическите заряди, намиращи се в компютърната памет или в компютърния харддиск. Изпълнението на една софтуерна програма може да се оприличи на вихрушка от електрически сигнали, които се разкарват из частите на компютъра. "Реалният" резултат от една програма в повечето случаи представлява фотони излъчени от монитор към очите на потребителя. При производството на софтуер, дизайнерите често използват абстрактни концепции като обекти, компоненти и процеси. В ролята на наблюдатели, ако искаме да наблюдаваме софтуерно изпълнение в термините на същите абстрактни концепции, както и при неговия дизайн, ние имаме нужда от система за наблюдение. Такава система за наблюдение би ни разкрила великолепния свят на взаимодействащи обекти, процеси и компоненти, и как те довеждат програма до желан (или нежелан) край.

За генерирането на стереограмата от корицата използвах програмата Surface 3D Release 2 [S3DR204].

About the Cover Page

The image on the cover has certain special properties: when viewing it in a special way one can see a tri-dimensional scene emerge out of it. We call this type of images *steregorams*. The principle behind stereograms dates back to 1833, when the physicist Sir Charles Wheatstone recognized that when humans look at an object each eye sees a separate picture of that object. The brain takes these two pictures and creates what we perceive as a three-dimensional image. A stereogram exploits this feature of the brain to hide a tri-dimensional scene in a two-dimensional image.

On the cover I have put an image, containing the following scene:



The following technique helps the viewer to see the hidden scene in a stereogram:

1. Put your face close to the stereogram;
2. Allow your eyes to relax, and stare right through the stereogram as if your eyes were focused at a point behind the surface of the stereogram;

3. Slowly move away from the stereogram without changing the position of your eyes.

You won’t see the emerging tri-dimensional scene immediately, but keep trying. Please note that only 85-90 percent of people can see stereographic pictures in a three-dimensional way.

I have created this particular stereogram to illustrate the role of a monitoring system in software development. Software does not exist in the same meaning of the word “exist” when applied to things such as potatoes. Note that a print out of some program source code becomes as real as the word “potato” written on a piece of paper. The electrical charges captured within the computer memory or on the plates of the computer hard drive represent the closest to “real” that software gets. When executing software, its “real” execution constitutes a whirlwind of electrical impulses moving around within the computer parts. The photons that user’s eyes detect coming from the computer screen represent the closest to “real” output of a computer program execution. We model software and what they do using abstract (ones that do not “really” exist in the physical world) concepts such as objects, processes, components and events, and if we want to “see” what actually happens in this same terms during the “storm” of electrical charges constituting a program’s execution, we need to use a monitoring system. The monitoring system would reveal to us a fascinating world of interacting objects, processes and components, which seem to govern how the computer produces desired (or undesired) output.

To create the stereogram I used the free tool Surface 3D Release 2 [S3DR204].

Acronyms and Abbreviations

Short form	Expanded form
API	Application Programming Interface
CCM	CORBA Component Model
CMA	Co-located Monitoring Agent
COM	Common Object Model
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DCE	Distributed Computing Environment
DCOM	Distributed Common Object Model
DDM	Diagram Data Model
DMA	Domain Monitoring Agent
DSA	Distributed Software Application
DSC	Distributed Software Components
DSD	Dynamic Service Deployment
EBNF	Extended Backus-Naur Form
EJB	Enterprise Java Beans
FIFO	First In First Out
GMS	Generic Monitoring System
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IDL	Interface Definition Language
IPC	Inter-Process Communication
JDI	Java Debugging Interface
JDK	Java Development Kit
JMS	Java Messaging Service
JMX	Java Management eXtensions
JNI	Java Native Interface

JPDA	Java Platform Debugger Interface
JSP	Java Server Pages
JVM	Java Virtual Machine
JVMPI	Java Virtual Machine Profiler Interface
JXTA	Juxtapose, the project
LMA	Local Monitoring Agent
LRC	Limited Realized Causality
MA	Monitoring Agent
MLM	Multi-Layer Monitoring
MM	Monitoring Model
MODOCC	MOnitoring Distributed Object and Component Communication
MSD	Message Sequence Diagram
MSS	Monitoring Support System
NTP	Network Time Protocol
OMG	Object Management Group
OO	Object-Oriented
ORB	Object Request Broker
OS	Operating System
OSA	Open Services Access
PAM	Presence and Availability Management
PI	Portable Interceptors
POA	Portable Object Adapter
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAP	Service Access Point
SHA	Secure Hash Algorithm
SNMP	Simplified Network Management Protocol
SNTP	Simple Network Time Protocol
SQL	Structured Query Language
SSIL	Simple Specification of Interest Language
STSL	Simple Type Search Language
TINA	Telecommunication Information Networking Architecture
UML	Unified Modeling Language
WAS	WebSphere Application Server
WWW	World Wide Web
XMI	XML Metadata Interchange
XML	eXtensible Markup Language